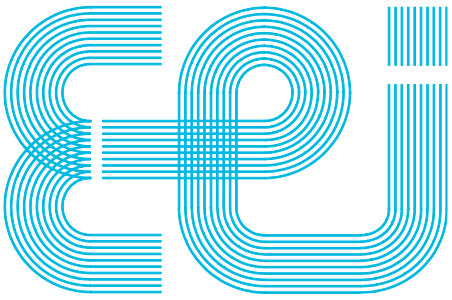


Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

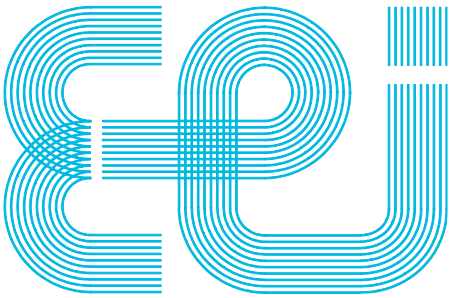
Mestrado en (ver apartado 1.1)

ALUMNO: Nome e Apelidos

DIRECTORES: Nome e Apelidos

Nome e Apelidos

Universidade Vigo



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Mestrado en (ver apartado 1.1)

Documento

MEMORIA

Universidade Vigo

CONTENTS

FIGURE INDEX

TABLE INDEX

GLOSSARY

1 INTRODUCTION

1.1 Problem definition

1.1.1 Motivation

This academical project answers to the desire of its Author to explore the field of applied machine vision and thus, aims at solving a challenging computer vision problem from a hands-on perspective, consisting on the detection and localization of items within a given environment.

This interesting application of autonomous vision systems is yet a current challenge in the world of robotics. While this type of task is usually performed combining a series of sensors for higher detection reliability, an achievable relaxation by using only camera data is still a problem of interest. Such relaxation still requires solving the following problems:

- Object detection: Which nowadays is typically performed with pipelines based on convolutional neural networks. Object detection consists on finding objects in 2D images and drawing square bounding boxes around them.
- Item localization: Which can be performed by the simultaneous localization of the camera with respect to a cordinate frame of reference plus the localization of an item with respect to a camera.
- Autonomous device control: Which is left out of the scope of this project. Effective control for an application like this can be performed by simple control loops provided the localization problem is solved.

Object detection is performed with convolutional neural networks, data-driven models that learn to identify a set of items in pictures from example labelled data. Exploring this field requires a solid understanding on the basics of machine learning and its more specific subfield of deep learning. The whole paradigm on approaching and solving a data-driven problem needs to be addressed, perviously requiring deep research on relevant object detection models to get an understanding on how said detection is performed from a low-level perspective. The design of a quality object detection solution is unattainable for a project this size, and thus the more practical approach of benchmarking suitable models and keeping the best for a custom case will be the one to follow.

Item localization, on the other hand, is a problem closer to the field of robotics. Though typically solved by GPS, finding a solution the localization problem is sometimes a non-trivial task, yet essential to most autonomous robotic applications.

The completion of these goals will provide a series of competitive skills in the field of applied machine vision. From an industry perspective, robot vision problems usually take place in controlled environments, in which solutions of utmost interest such as automated defect detection, item inventory, visual vehicle guidance, etc. can easily be approached with the set of skills obtained after the completion of these more challenging tasks.

1.1.2 DARPA Subterranean Challenge

As it has been stated, this work aims to solve an example task of detection and localization of items within a given environment. The particular realization of this task to be carried out is aligned with that of the DARPA Subterranean Challenge.

The DARPA Subterranean Challenge 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.

1.1.3 DARPA artifact detection from images

As stated in [], not all items are supposed to be detected via images. By only using a camera, the detection of the following items show several difficulties:

- CO_2 source: 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.

And thus these are removed from this work's scope. 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.

1.1.4 Item localization

Item localization aims to produce a computer program capable of drawing a likelihood map for the position of detected items in a real environment, exclusively from camera data. The detection to come from an object detection pipeline predicting a bounding box in a 2D image. Because the availability of the DARPA artifacts to this work's author is limited, this stage is to be performed for any set of sample items rather than the ones depicted above.

Table 1: DARPA SubT Artifact Detection Challenge tangible 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		Typicall 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	N/A	A CO_2 emmitter device that simulates a range of hazardous air quality conditions.

Source: []

1.2 Scope & goals

1.2.1 Goals

The main goals of this project are initially stated as follows:

- Provide the author with knoweledge on the Computer Vision techniques supported on convolutional neural networks that had led the development of the state-of-the-art.
- 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.

After completion of the former, an open-end approach was followed in pursue of the following additional milestones:

- Approach the localization problem of a vision device in an underground setting.
- Develop a pipeline for localization of items in a 3D real envionment, under the conditions of an underground setting, using consumer technology.

1.2.2 Scope

1.3 Methodology

1.4 Notation aspects

In equations, products and convolutions

2 BACKGROUND

2.1 Artificial Intelligence

2.1.1 Artificial Intelligence

Artificial intelligence is the "effort to automate intellectual tasks normally performed by humans" [1], 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 [1]:

- 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 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. 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 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, 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 [1]:

- 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 Leaning 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 [1], typically learned by models called artifical neural networks (ANNs).

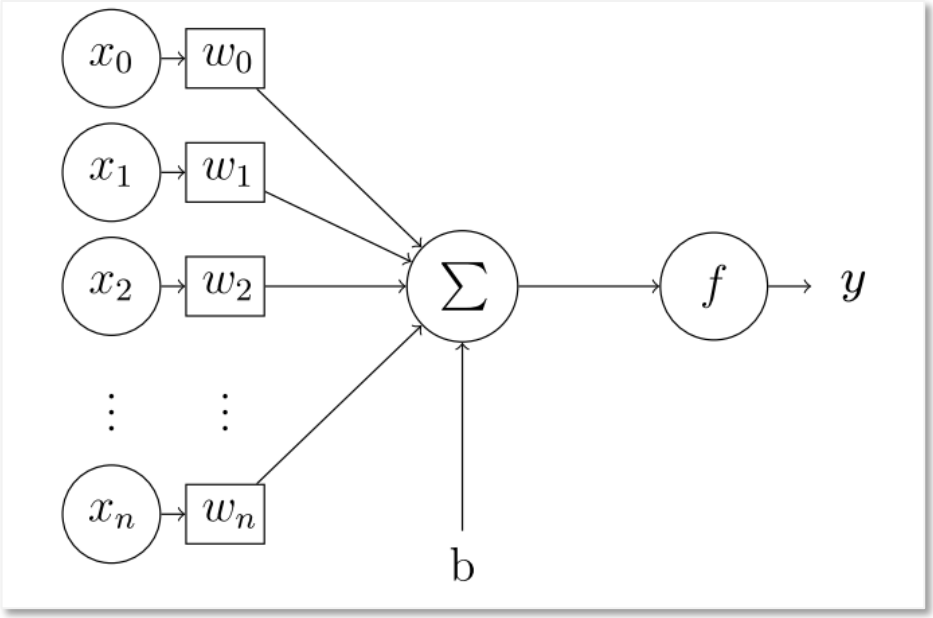


Figure 2: The perceptron
Source: [2]

Neural networks are generally comprised by stacks of neuron layers. The Perceptron [3] is a simple mathematical model for the neurons in an ANN, a representation of which can be seen in FIGURE. 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:

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

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 $\boldsymbol{y_{training}}$ 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 [1]:

- 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.

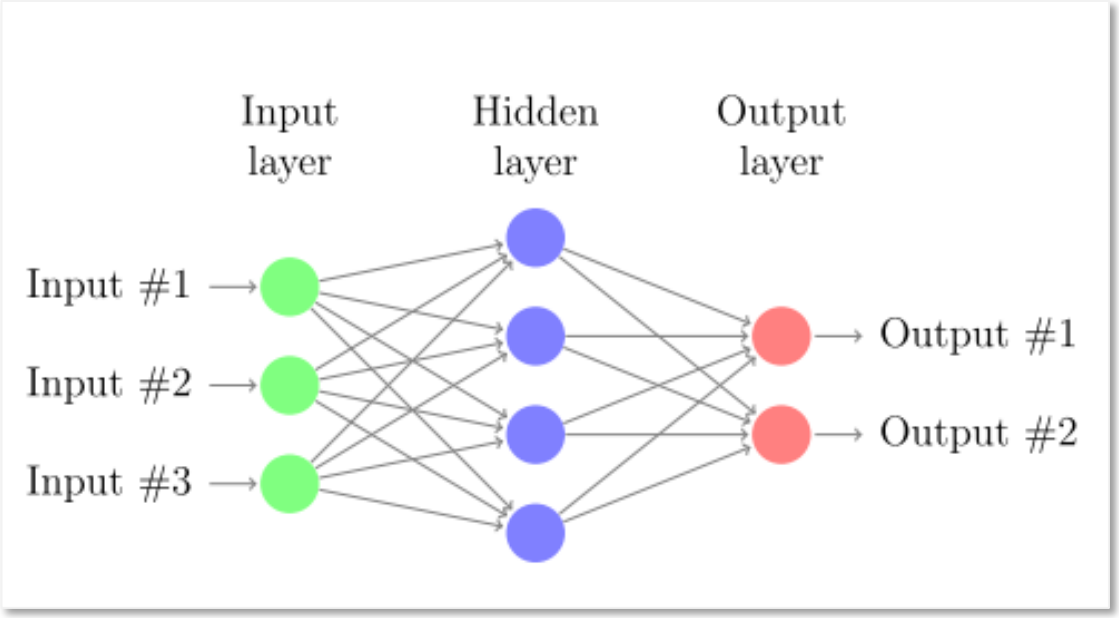


Figure 3: A very simple ANN architecture
Source: **[2]**

FIGURE 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, 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, 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 [2]. By adopting the Perceptron model for a densely-connected layer, the ANN in FIGURE can be mathematically expressed as:

$$\boldsymbol{h_1} = f_1(\boldsymbol{x} * \boldsymbol{w_1} + \boldsymbol{b_1})$$
$$\boldsymbol{y} = f_2(\boldsymbol{h_1} * \boldsymbol{w_2} + \boldsymbol{b_2})$$

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 neccesary 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, opimizers update the parameters of a network so that the loss $\boldsymbol{\ell}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ is minimized [1].

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:

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

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* [1].

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 [1]. Examples of other opmitizers are *Adagrad, RMSProp*, etc.

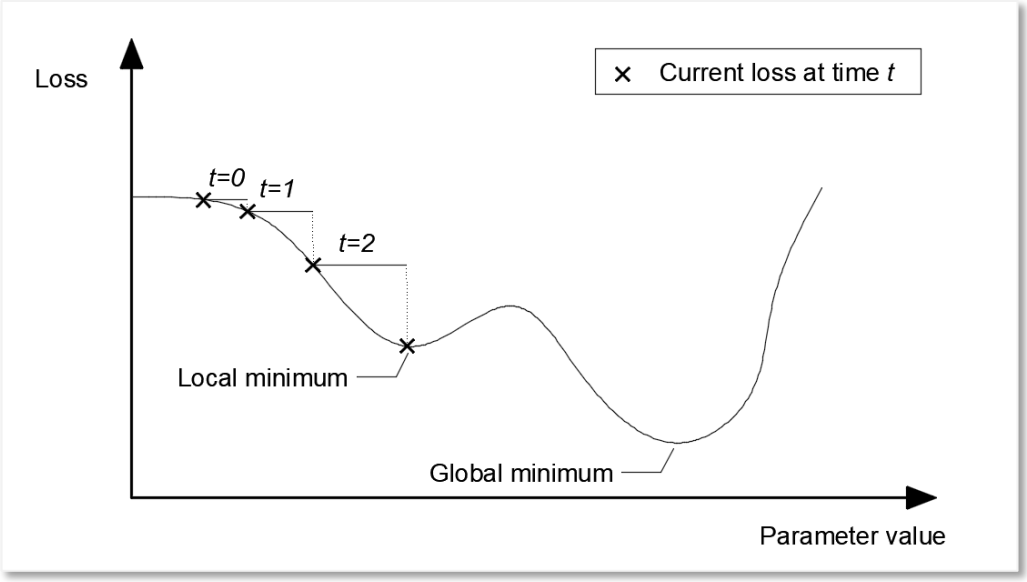


Figure 4: Single-dimensional loss during training
Source: Self-made

FIGURE 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. The Backpropagation algorithm 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 [1]. FIGURE 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. As pointed in the example in FIGURE, 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.

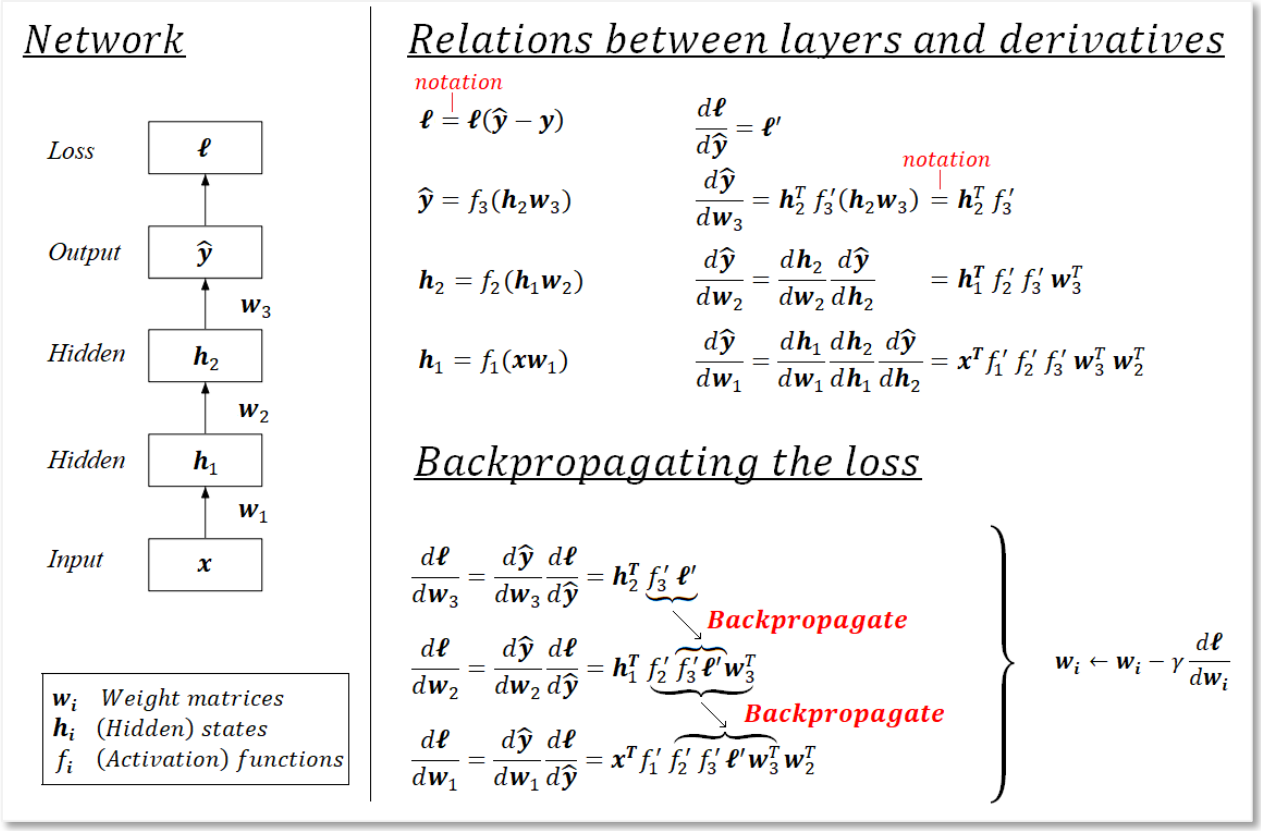
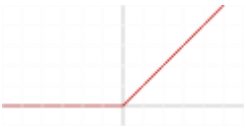
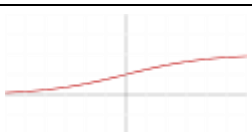
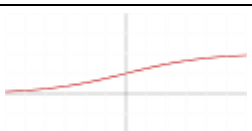


Figure 5: Backpropagation illustrated
Source: Self-made

Table 6: 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: [4]

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 [5]. 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 [1]. Towards this aspect, good choices of intermediate activation functions are, for example: *rectified linear unit* (ReLU), *leaky ReLU*, etc [6].

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 functions 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 [1]:

- *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 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:

- 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 amount 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 perrformance 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* [1] 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?".

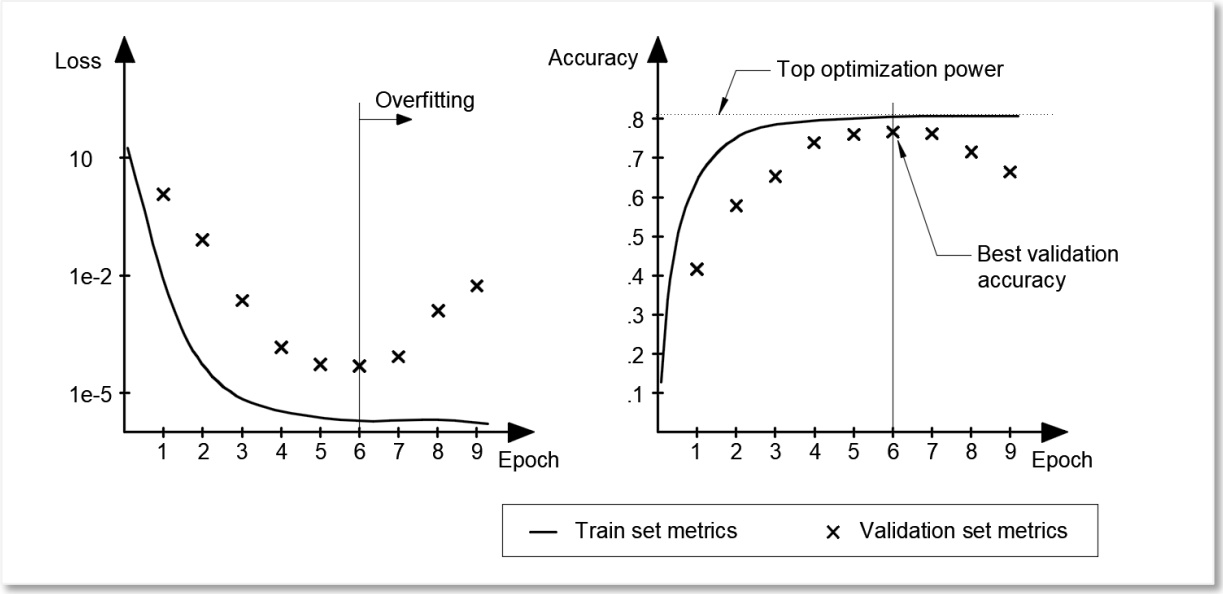


Figure 7: Training metrics for validation and test set
Source: Self-made

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, 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 [1]:

- *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

Two plots of a hypothetical training run for an image classification problem are shown in FIGURES, 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 amount 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 [1]. 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}.
- Design 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 [1]:

- 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 less 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 [1]:

- 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 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 graydcale 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.

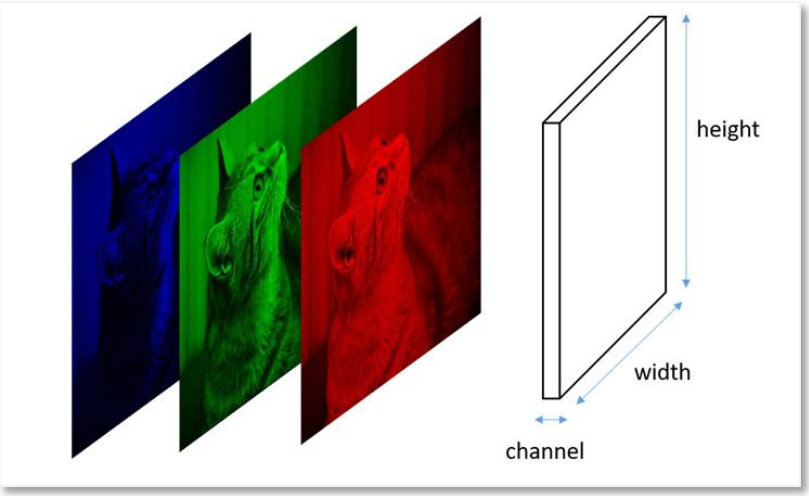


Figure 8: RGB channel decomposition of an image
Source: https://stats.stackexchange.com/questions/427680/

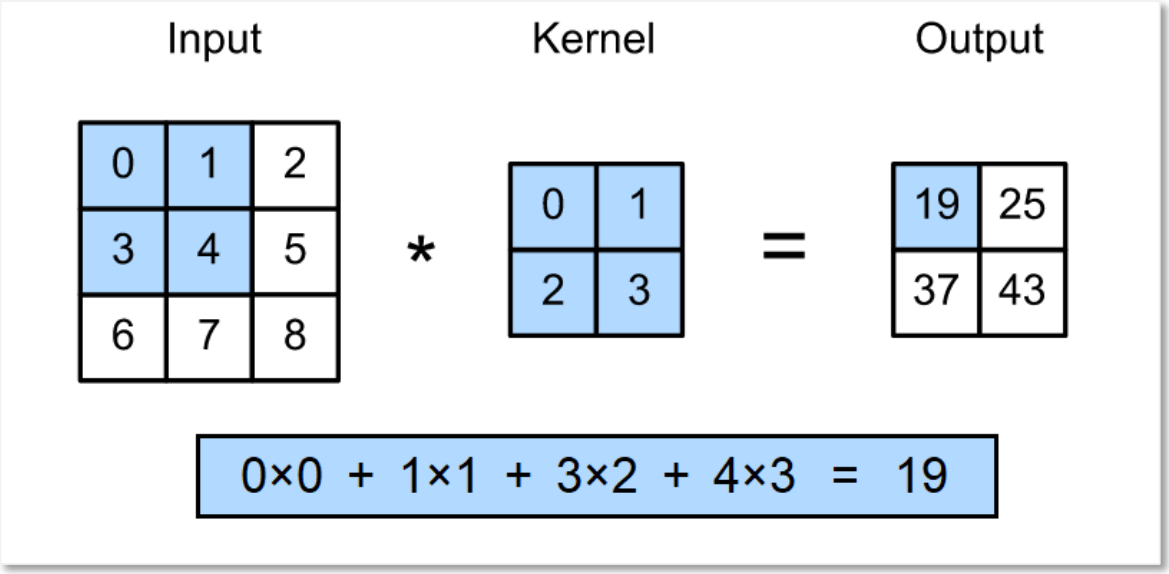


Figure 9:
Source: https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html

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:

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

- x : Being the input image undergoing the convolution operation.
- 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.
- 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. 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, 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.

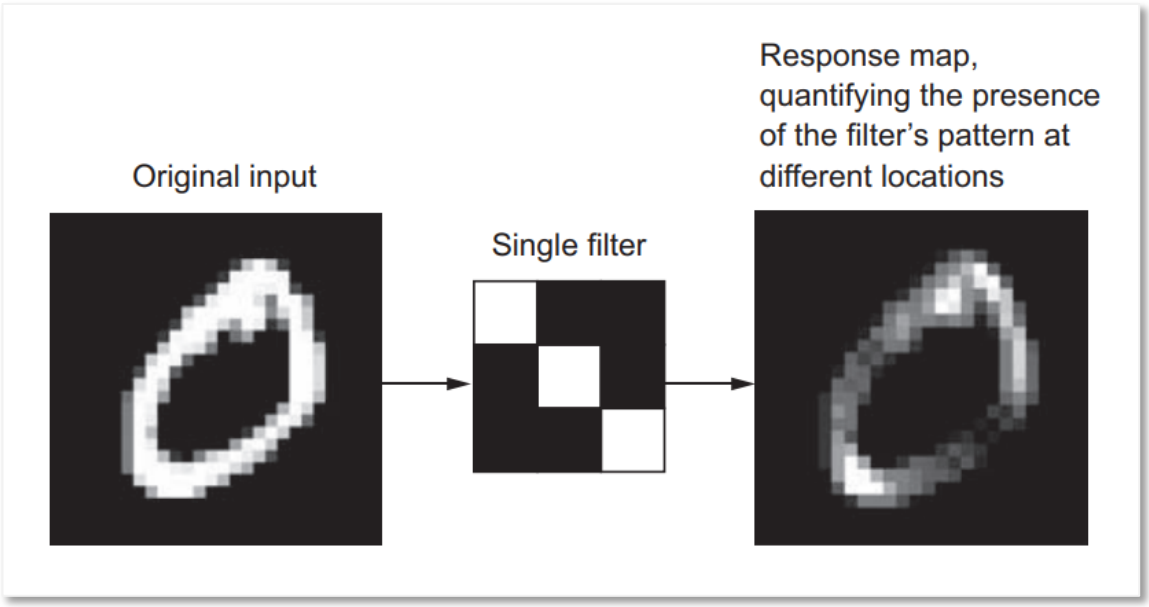


Figure 10: Training metrics for validation and test set
Source: [**1]**

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 [1]. 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 convolution 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;

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

- 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 [1]. 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 shows an example of some augmentations.

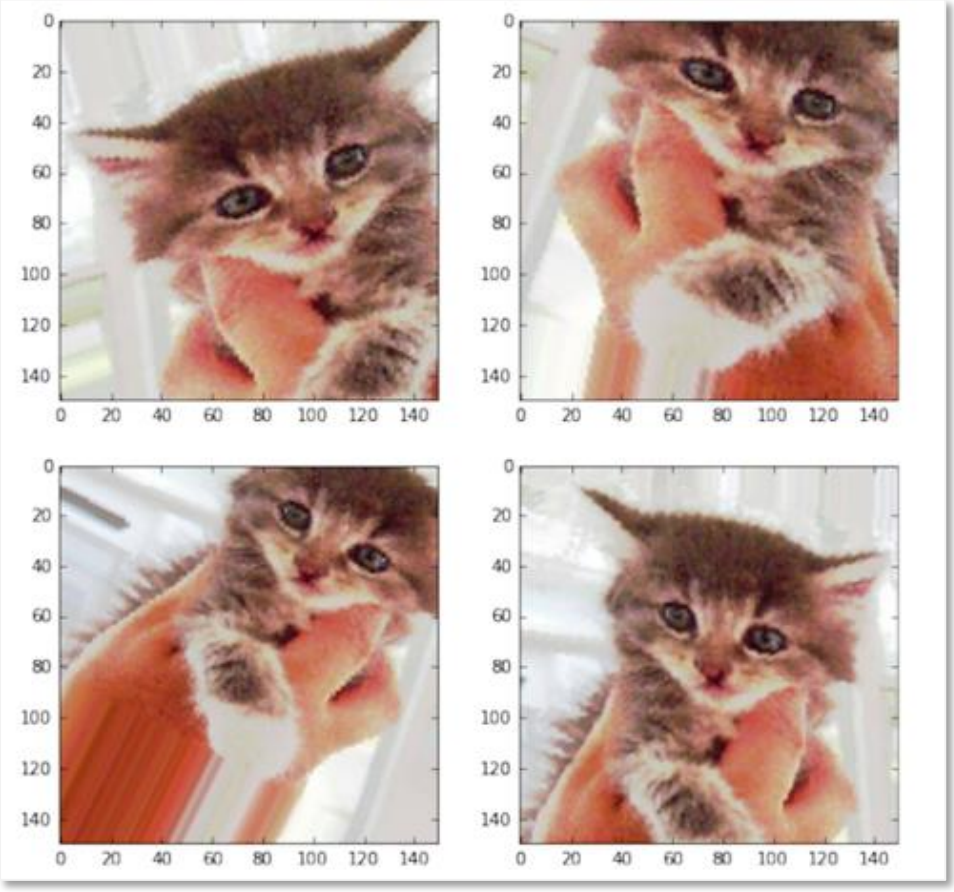


Figure 11: Image augmentation by warping and mirroring
Source: [1]

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 [2]. 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 [7]:

- *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, specially 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 [8].

2.2.8 Alexnet. Image classification

Alexnet [6] 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 [6].

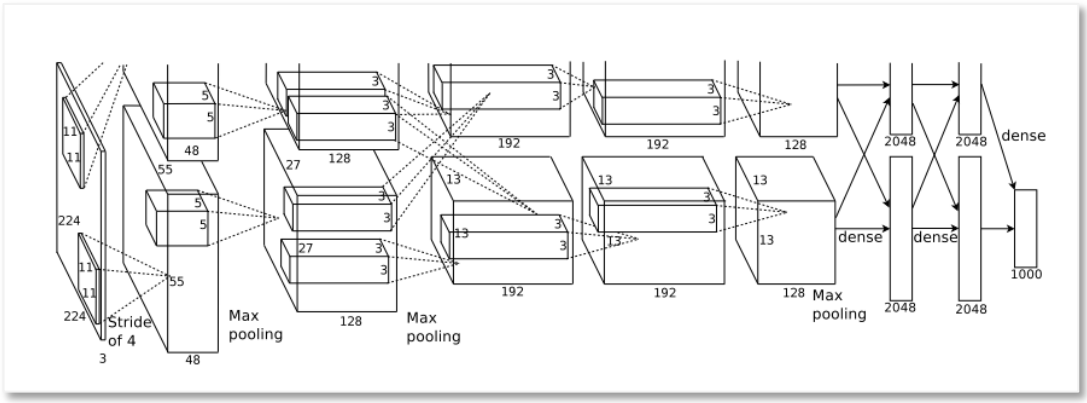


Figure 12: Alexnet architecture
Source: [6]

The architecture of Alexnet is shown in FIGURE. 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 [1], 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. Some applications of object detection lay in the fields of automated visual inspection, robot vision, autonoumous driving, video surveillance, etc [9]. Video-based object detection pipelines also exist, though they are overall less popular than their image counterpart, as they have only emerged more recently [10]. 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.

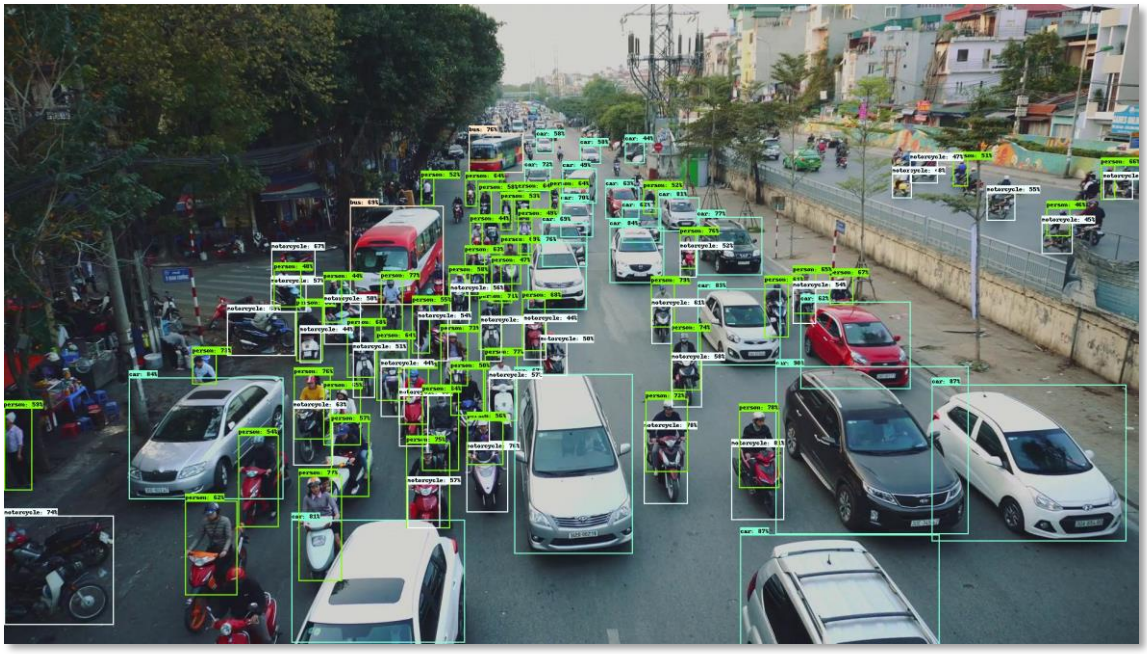


Figure 13: Showcase image of object detection with EfficientDet
Source: https://github.com/google/automl/tree/master/efficientdet

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 [6]. 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 [11] and PASCAL VOC [12].

Object detection methods can be divided in two categories depending on their structure [9]:

- *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 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.
- *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*): 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* [2]. 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 falls 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:

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

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

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

- 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 $\boldsymbol{Precision}$ & $\boldsymbol{PrecisionRecallCurve}$.

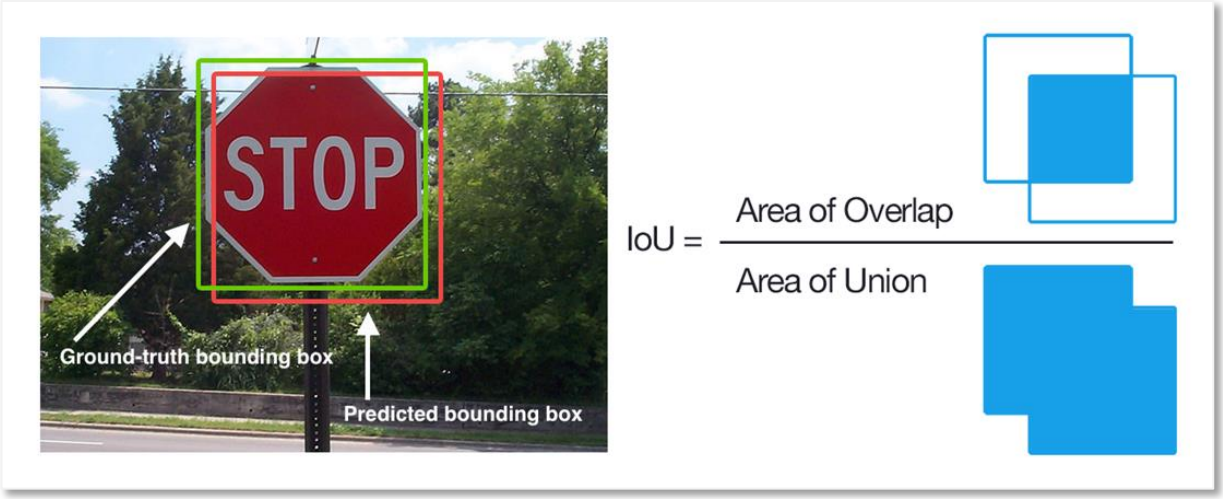


Figure 14: Jaccard Index or IoU
Source: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

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 [11] 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. 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 15: MS COCO metrics
Source: [13]

The metric named AP in FIGURE is the same metric that was defined as mAP in SECTION. The MS COCO team states [13] 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 [13], while others provide only additinoal 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:

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

There have been records of some discrepancies on how these metrics may be computed, depending on the platform used to do so [14]. These origin, among others, on the fact that some authors like to flatten the precision-recall curves so that they are monotonously decreasing [15]. The MS COCO team has published a library for computing their standardised metrics, which can be found in the COCO API repository at [16].

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 [16], 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* [8] 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, 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* [17] and *Faster R-CNN* [18], an overview of each shown in FIGURE. 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 [8]: In the original implementation, region proposal is performed by a non-data-driven *selective search* [19] 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 [17]: 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.

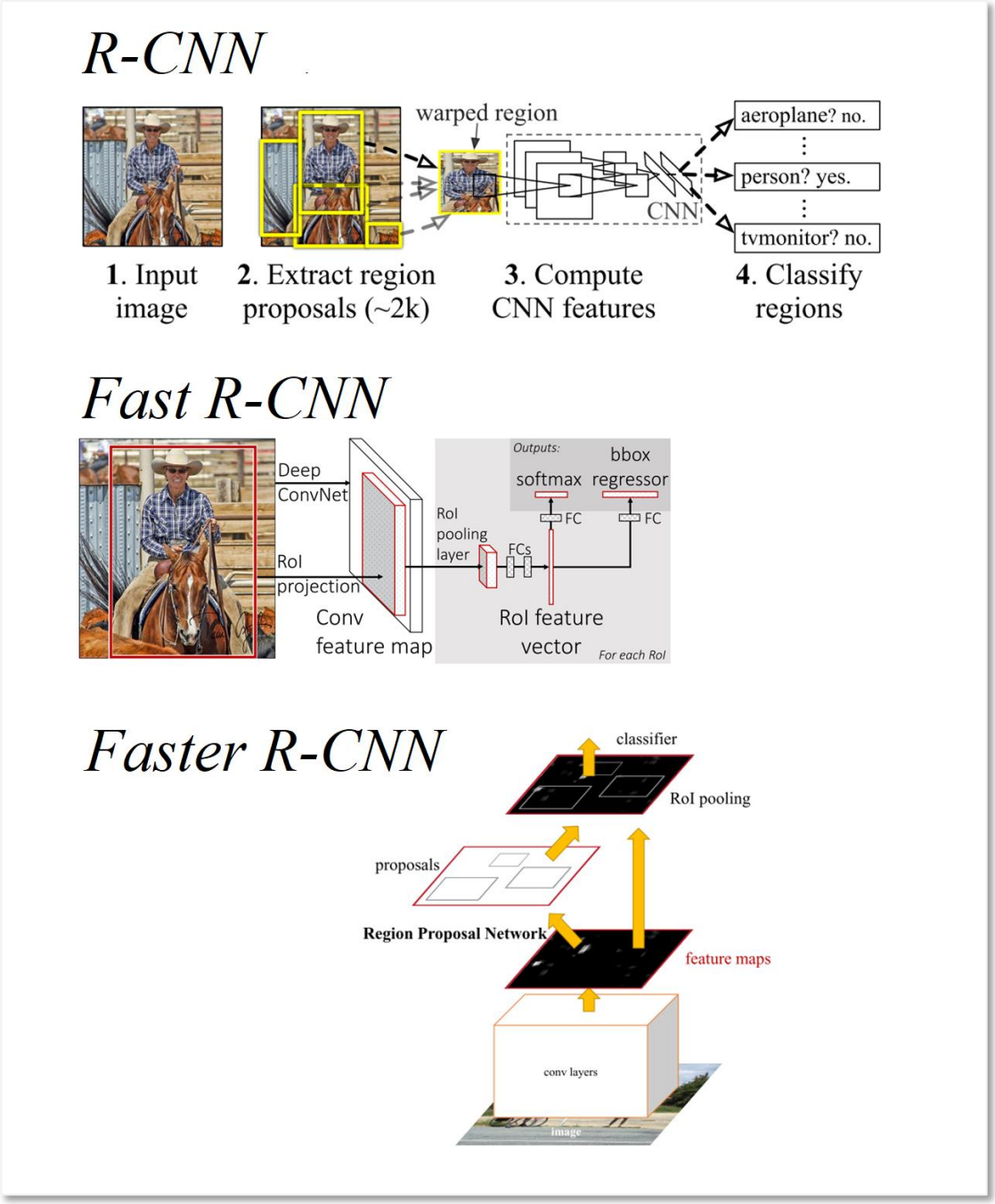


Figure 16: R-CNN, Fast R-CNN and Faster R-CNN overview
Source: [8], [17], [18]

Faster R-CNN [18]: 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*) [20] family (see also [21], [22]) of object detection pipelines is a set of single-stage models initially developed with the aim of maximizing detection inference speed. The original YOLO [20] 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* [23], an open-source neural network library written in C, still influential today.

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 [20].

The main limitations of this model [20] 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.

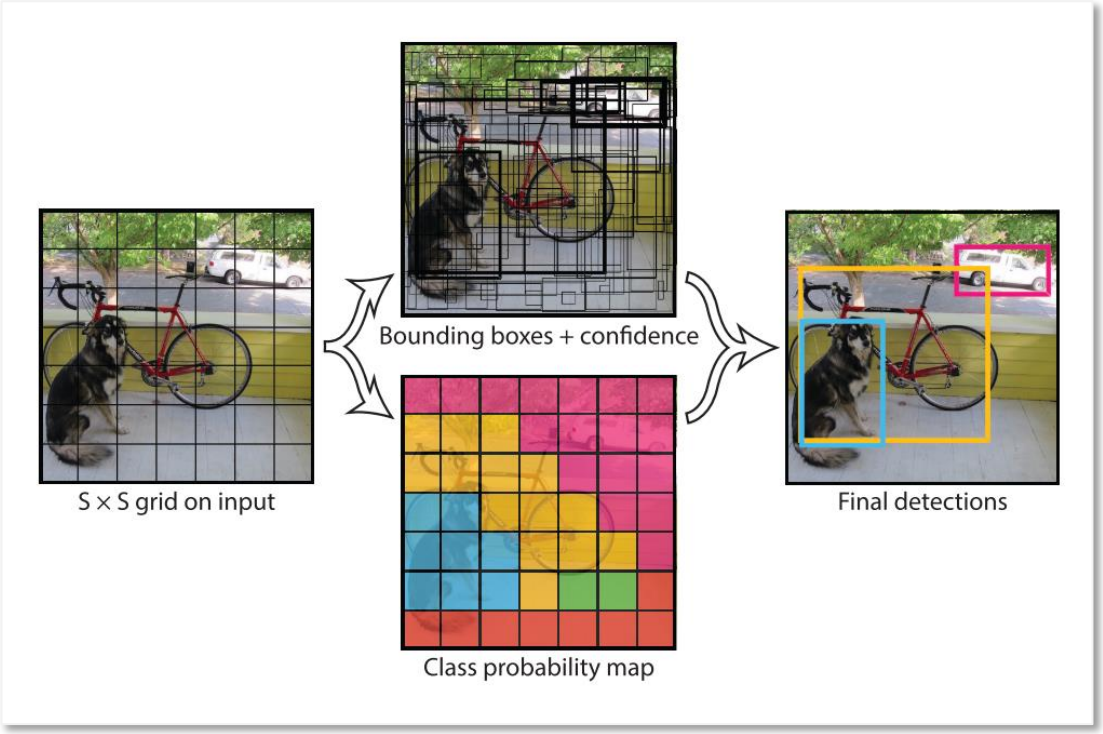


Figure 17: YOLO intuition
Source: [20]

2.4.3 SSD (2015)

The single-stage *single shot detector* or *SSD* [24] 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. Having 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 [fasterrcnn]. 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 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), 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 [24], these two are highlighted:

- Hard negative mining: With this approach, most of the default boxes are discarded as negatives, introducing a significant data. 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 more sure 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 [24].

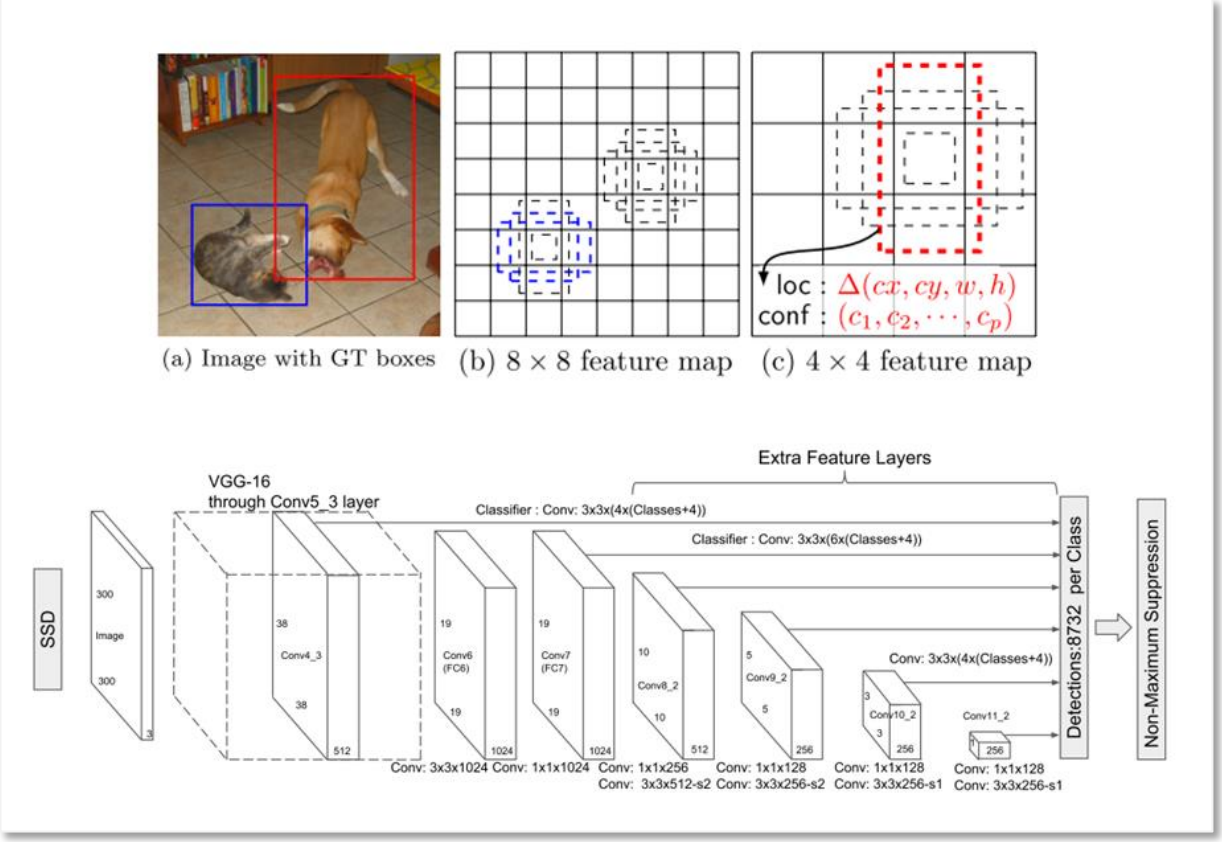


Figure 18: SSD intuition and architecture
Source: [24]

2.4.4 EfficientDet (2019)

EfficientDet [25] 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* [26], 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 boundinx boxes and their labels. The whole EfficientDet architecture is shown in FIGURE.

The purpose of the convolutional model EfficientNet [26] 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* [27], 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 [26], to obtain a series of compound-scaled convolutional architectures B0-B7.

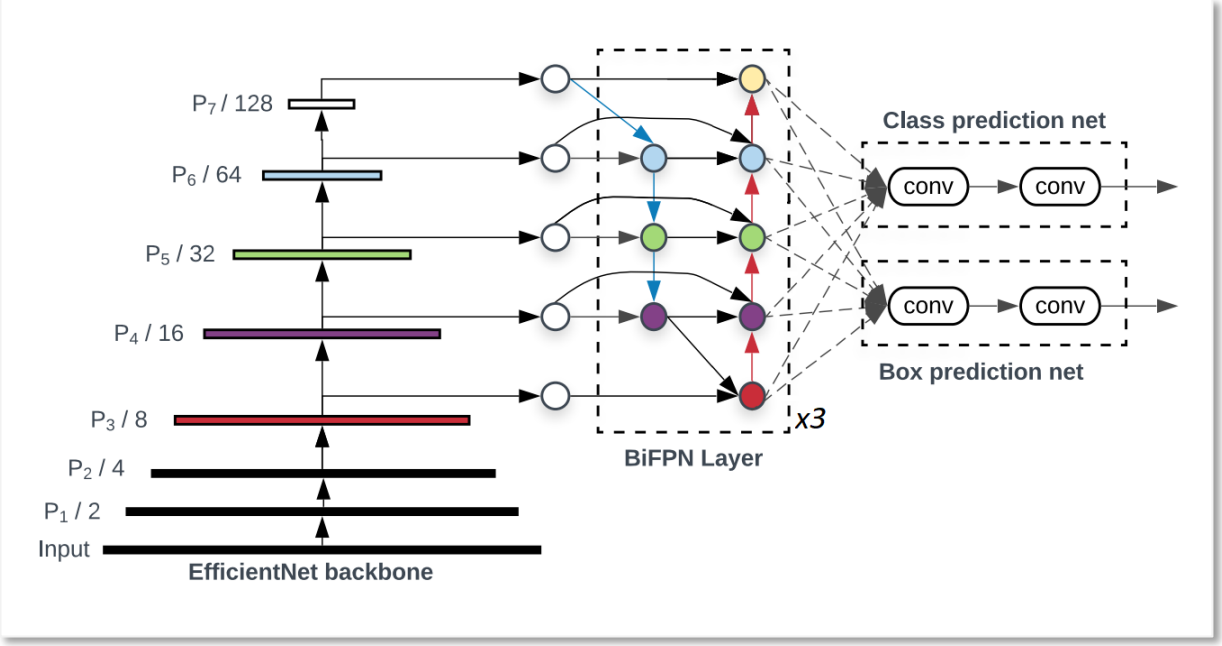


Figure 19: EfficientDet architecture
Source: [25]

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 [25], 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* [28]: Which adds a top-down pyramid architecture with shortcuts onto a size-decreasing convolutional backbone.
- *Path aggregation network* or *PANet* [29]: 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* [27]: Which proposes a complex model for feature aggregation layer discovered via an heuristic data-driven method.

From which, by applying and intuitions exposed at [25], the *BiFPN* layer model is derived as shown in FIGURE. 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 [26], the rest of the detector follows a series of scaling rules stated in [25]. Towards this work, it is more convenient to show the features of each particular model scale, and thus they are displayed in FIGURE.

	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 20: Scaling configurations for EfficientDet D0-D7
Source: [25]

2.4.5 YOLOv4 (2020)

YOLOv4 [30] 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 [21], YOLO9000 [21] and YOLOv3 [22]. The official implementation of YOLOv4, as well as the upgraded backwards-compatible version of Darknet, is available at [31]. The research paper in which YOLOv4 is introduced [30] 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 [30], 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:

- *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 [30]:

- *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.

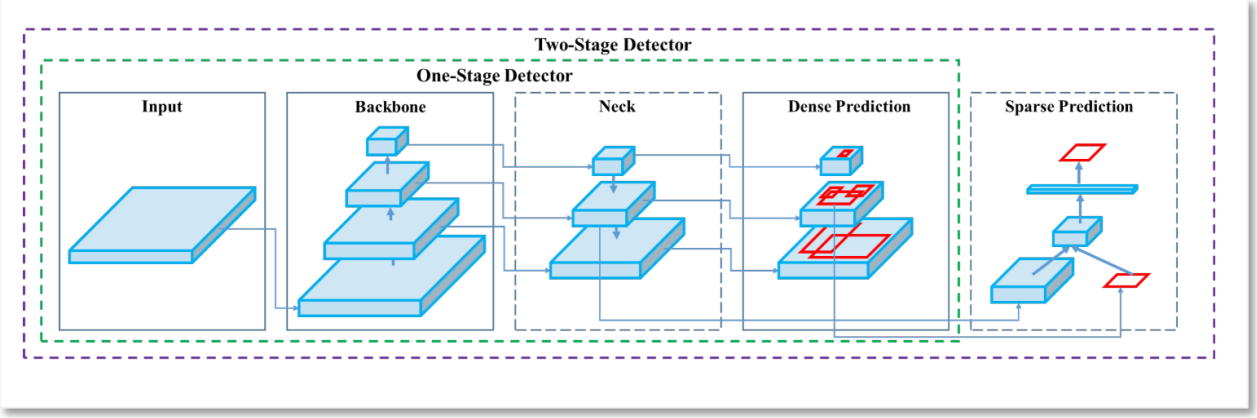


Figure 21: Generic object detection structure defined for YOLOv4
Source: [30]

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.

The YOLOv4 architecture is composed of [30]:

- 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 [32] 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 [30] 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.

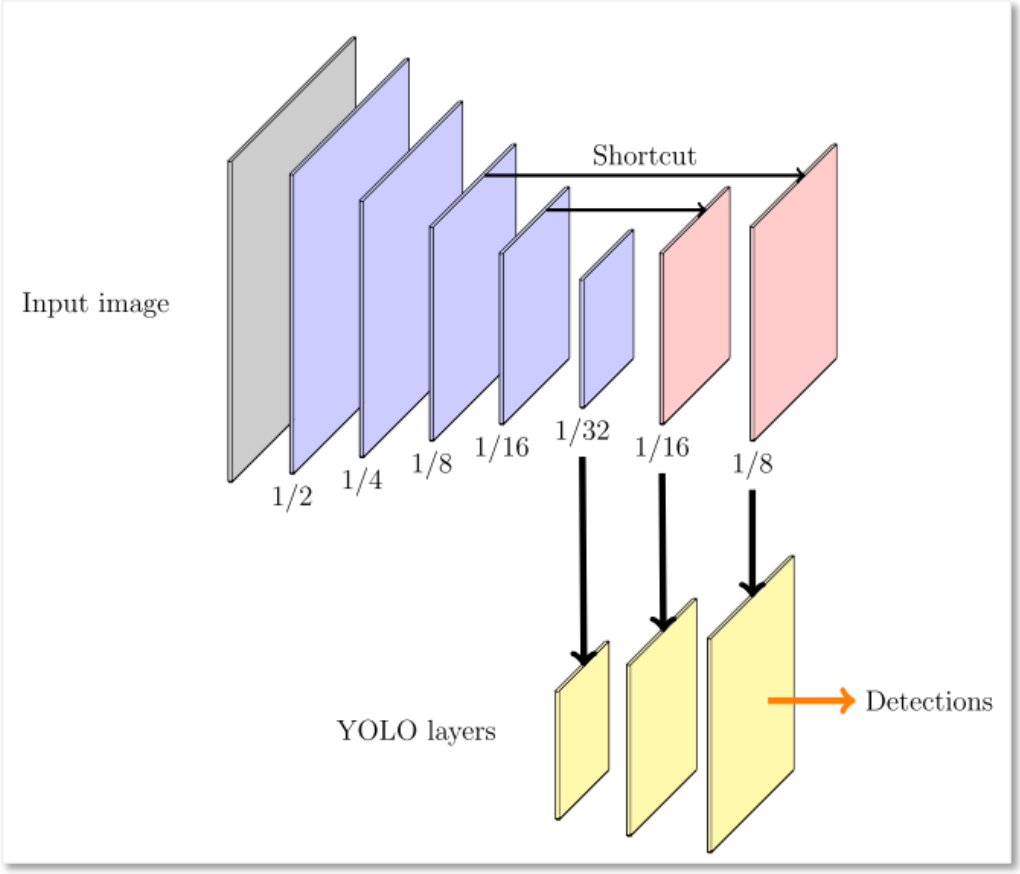


Figure 22: YOLOv3 overview
Source: [2]

FIGURE shows the architecture of the standalone YOLOv3 network [2]. 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 [33] show a scalability approach for the YOLOv4 architecture introduced in [30]. 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 [33]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 [33].
- Propose a series of upscaled/downscaled YOLOv4 networks.

The concept proposed in *CSPNet* [34] 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 [33]. 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 ponit, tiny and large YOLOv4 models are designed by following the scaling rules exposed at [33]. Thus, the models proposed at [33] 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

Benchmarking on MS COCO for soat detectors is shown in FIGURES XX and XX.

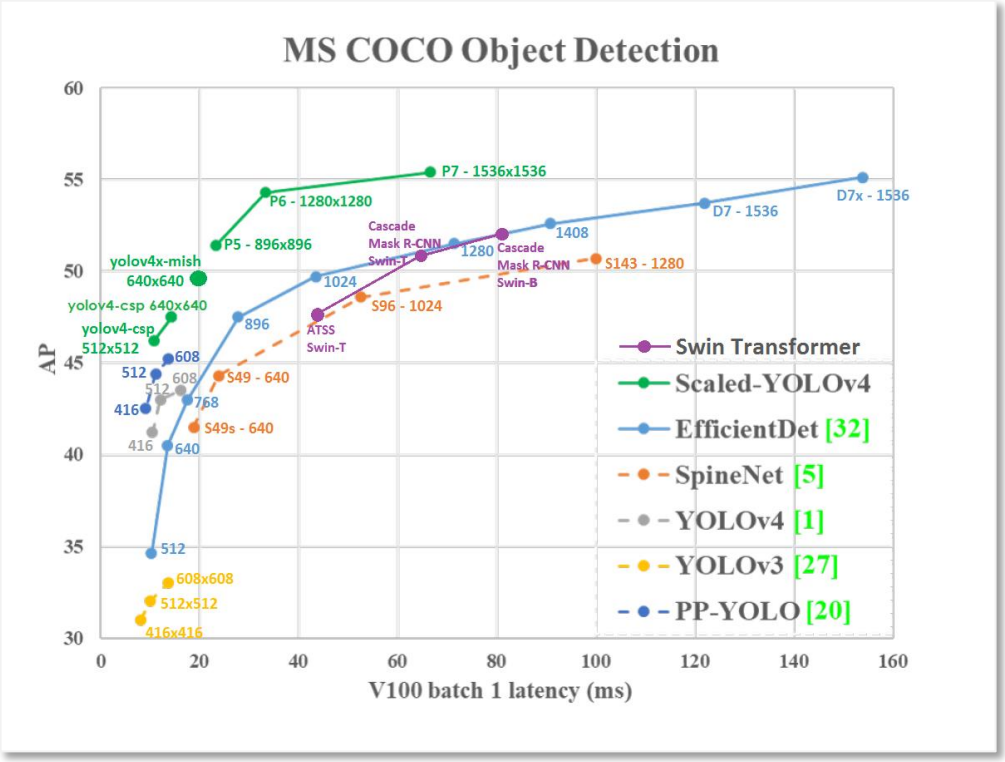


Figure 23: State of the art of different object detectors in MS COCO dataset I
Source: [31]

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 24: State of the art of different object detectors in MS COCO dataset II
Source: [33]

2.6 Enabler tools

2.6.1 Deep Learning libraries

2.6.2 Python 3

2.6.3 Jupyter notebooks and Google Colab

2.6.4 Bash

2.6.5 Unitys

2.6.6 Cv2

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, 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.

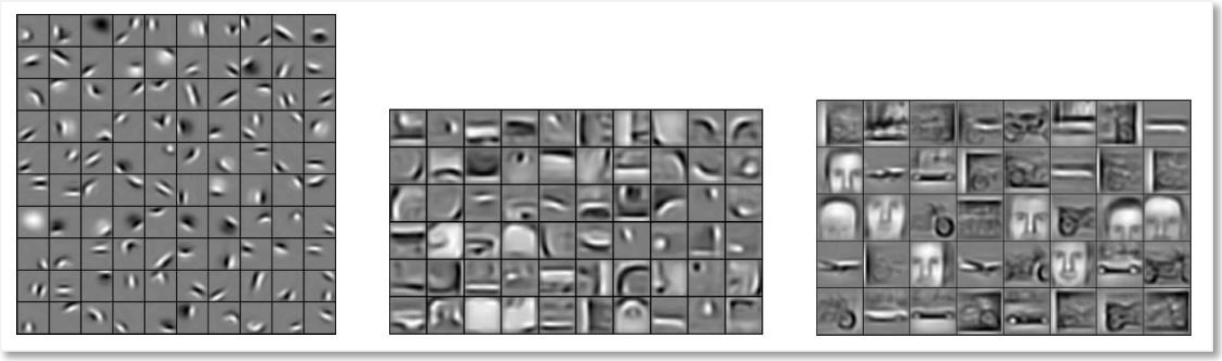


Figure 25: Progressively richer feature maps (lowest level information on the left)
Source: [35]

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

FIGUREX 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 [1]. 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 generally we 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. The upcoming sections will discuss each of these approaches in more detail.

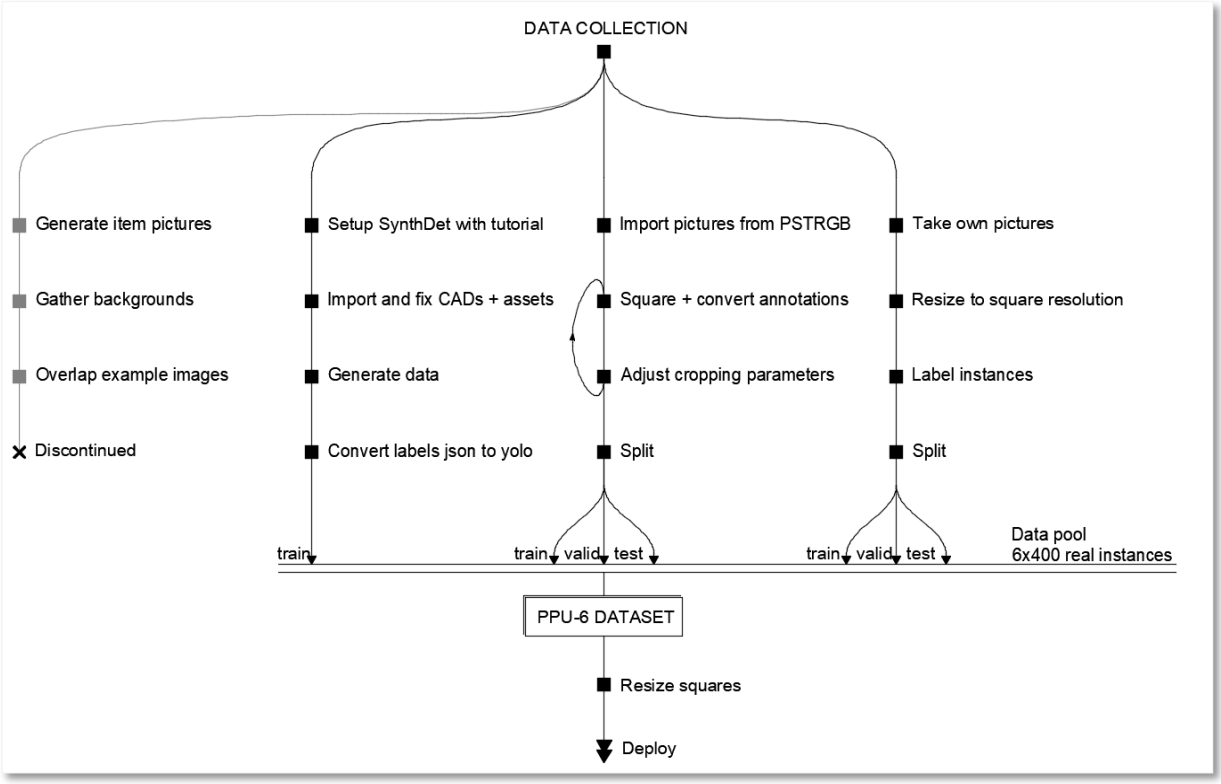


Figure 27: Data collection pipeline followed in this work
Source: Self-made

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 this is a less common extension.
- 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 [36], with the same name excluding extension. A full example of a YOLO annotation is shown in FIGURE. 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).

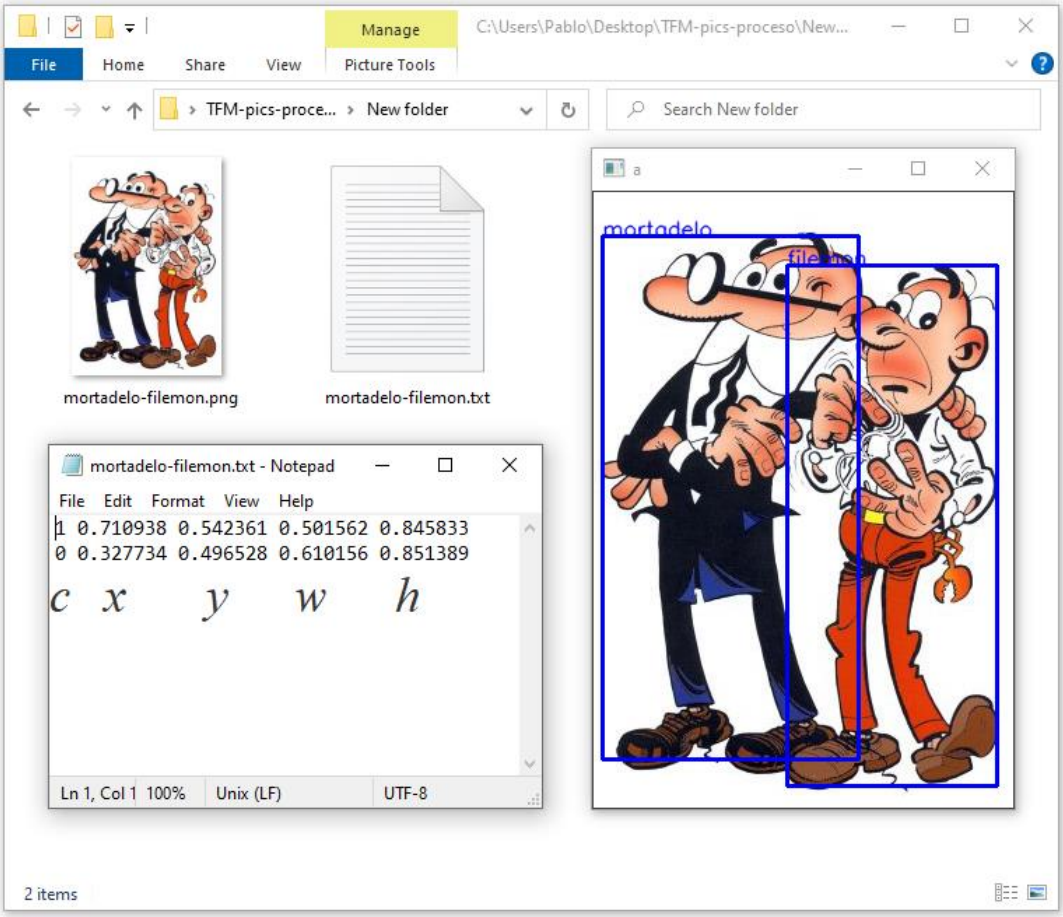


Figure 28: Full example of a YOLO annotated picture
Source: Self-made

On the other hand, MS COCO object detection annotations for many pictures are stored in common *.json* files [37]. 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 [37]. FIGURE shows the MS COCO annotations for the very same picture in FIGURE, both if it was a plain annotation or an inference result. Note that in the MS COCO annotation results some information required at [37] 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 [16], 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 29: MS COCO annotations for the picture in FIGURE
Source: Self-made

3.2 *Cut, Paste and Learn* from virtual models

Cut, Paste and Learn is a concept proposed in the research paper [38] 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.

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 [38], 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.

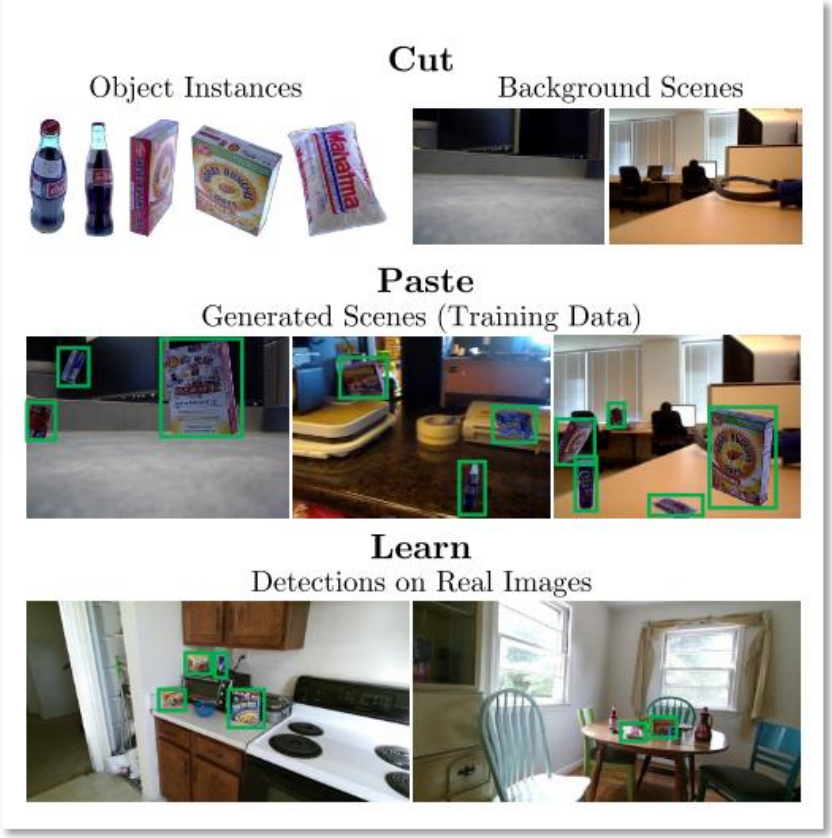


Figure 30: *Cut, Paste and Learn* overview
Source: [38]

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 [39]. 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 [39] 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 shows an outline of this approach.

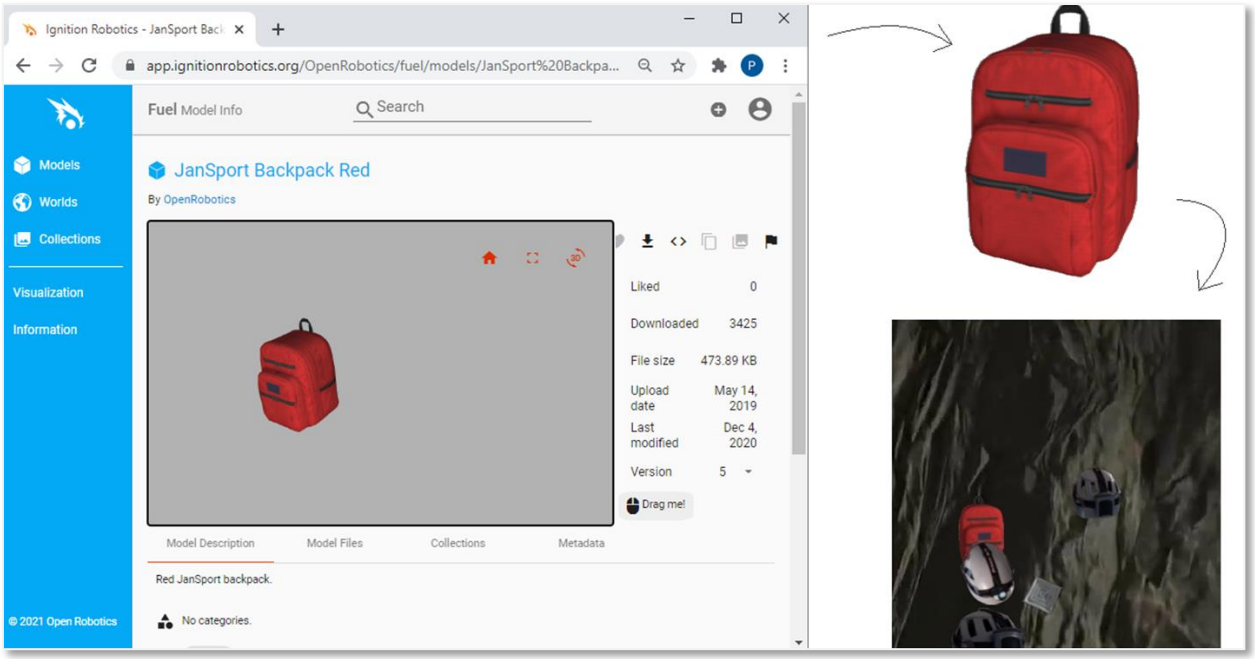


Figure 31: *Cut, Paste and Learn* overview
Source: Self-made

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 [40] 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 scripting.

It is shown in the research paper [41] 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.

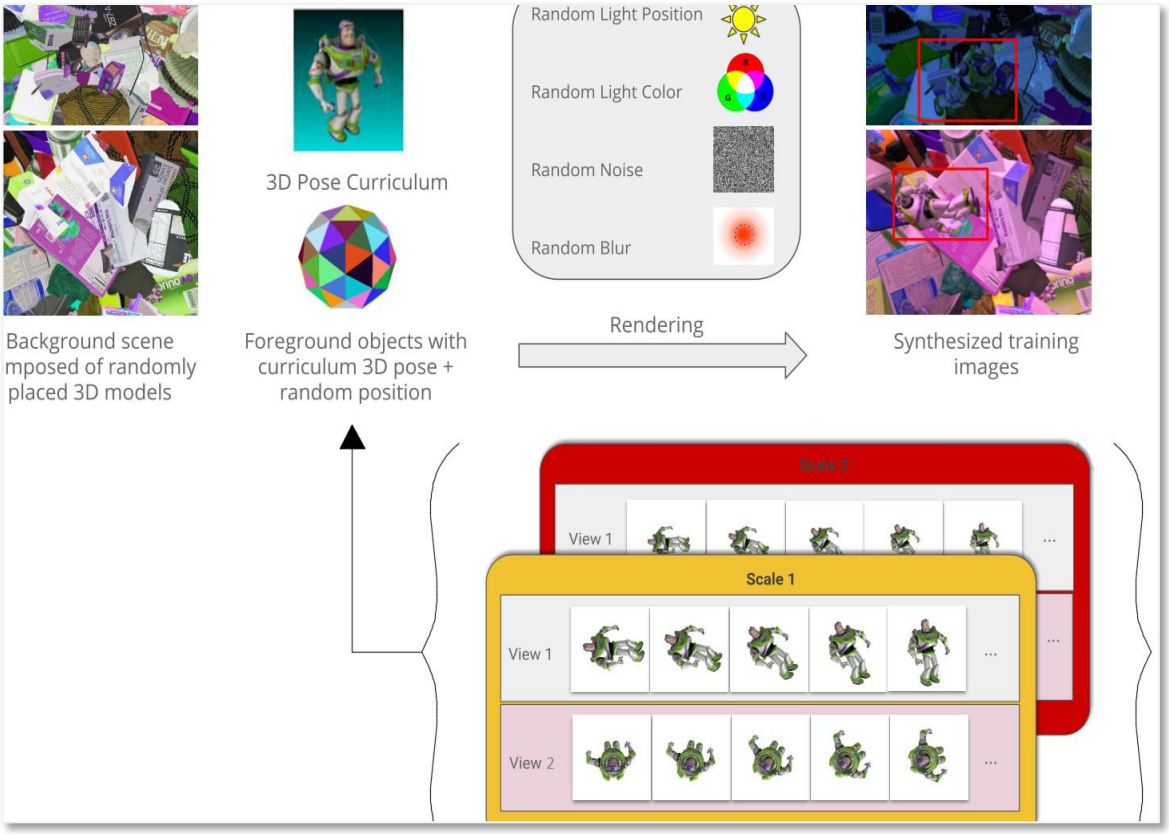


Figure 32: Synthetic data generation pipeline used in [41]
Source: [41]

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 shows an overview on how these items are placed in the environment during one arbitrary iteration, while FIGURE 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.

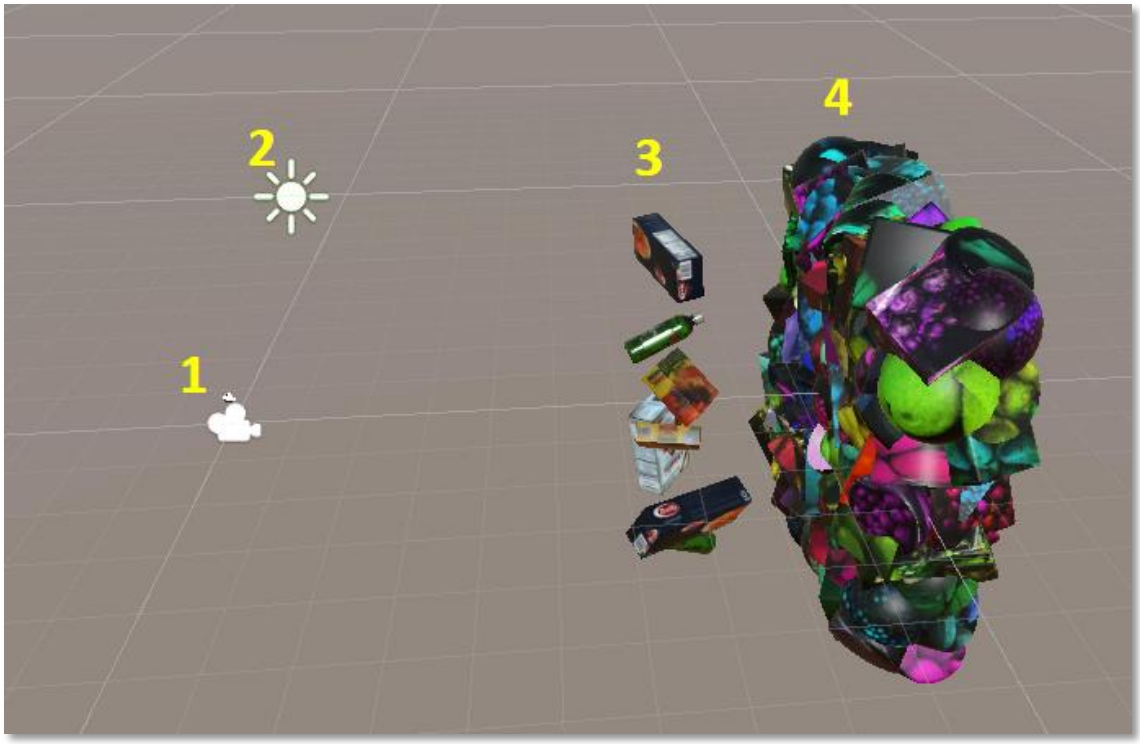


Figure 33: SynthDet environment. 1: Point of view; 2: light; 3: foreground layer 4: background clutter. Source: Self-made.

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

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

- Follow the tutorial at [42] 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 [39].
  - 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 [39] were meant to be used in a realistic simulation environment, in the sense that there would be floor and walls, the rope or the 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 neccesity. 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.

Among the assets that were used to generate synthetic data, some have been stored in this project's repository [#ref[]] 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 [#ref[]] for more details).

Figure 35: Outline of the Ignition Robotics' model issues after being imported into Unity
Source: Self-made

3.4 Importing data from PST-RGB

The PST-RGB dataset [43] 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 [44]. 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.
- A 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, 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.

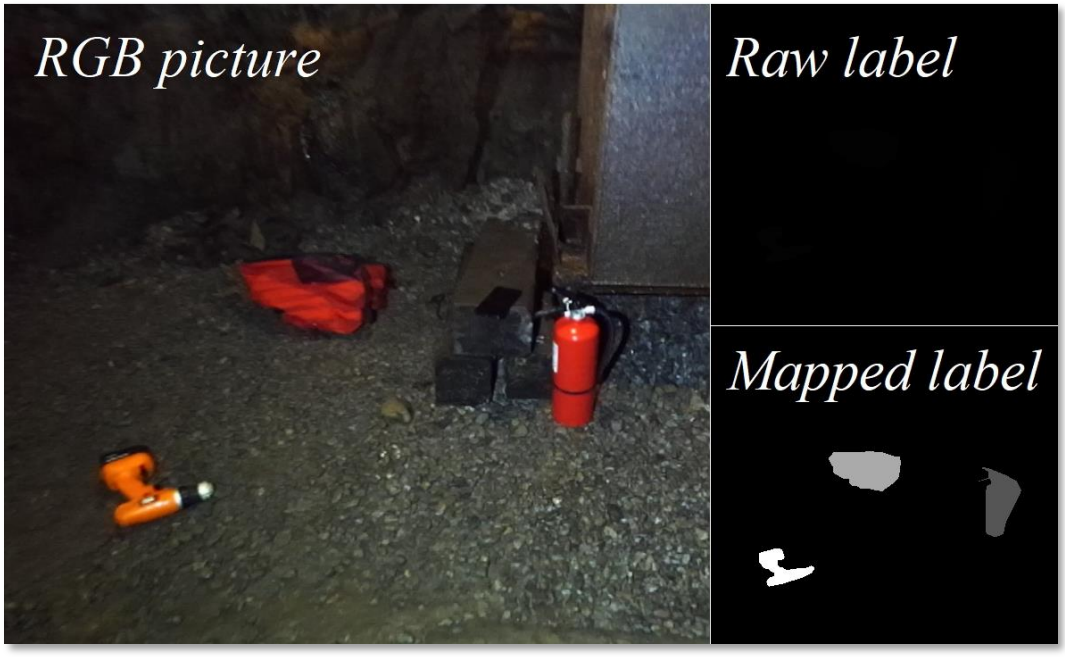


Figure 36: PST-RGB dataset sample
Source: Self-made

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 neccesary, 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 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 3th 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 3th visual inspection, the parameters used to convert the images were deemed satisfactory.

The scripts available at this project's repository at [#ref[]] 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 [2] 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 not carefully 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 [45], whose preview is shown in FIGURE.

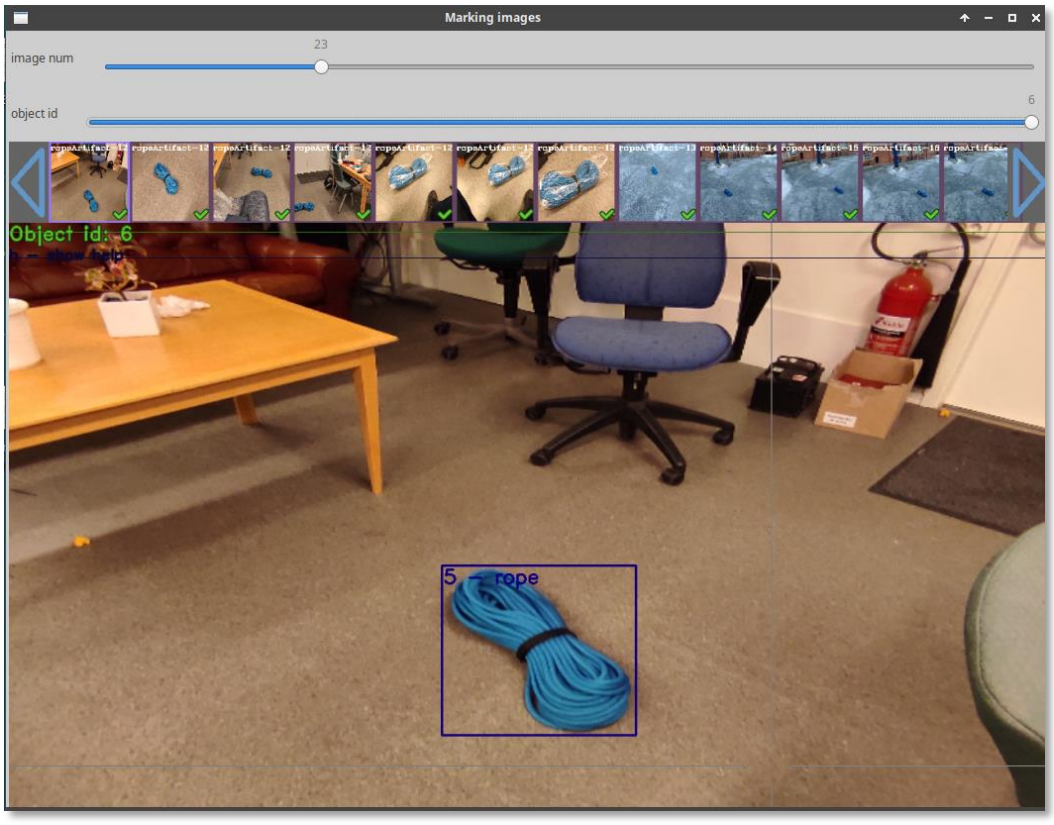


Figure 37: Detail of the annotation tool Yolo Mark
Source: Self-made

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 (Windows 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.

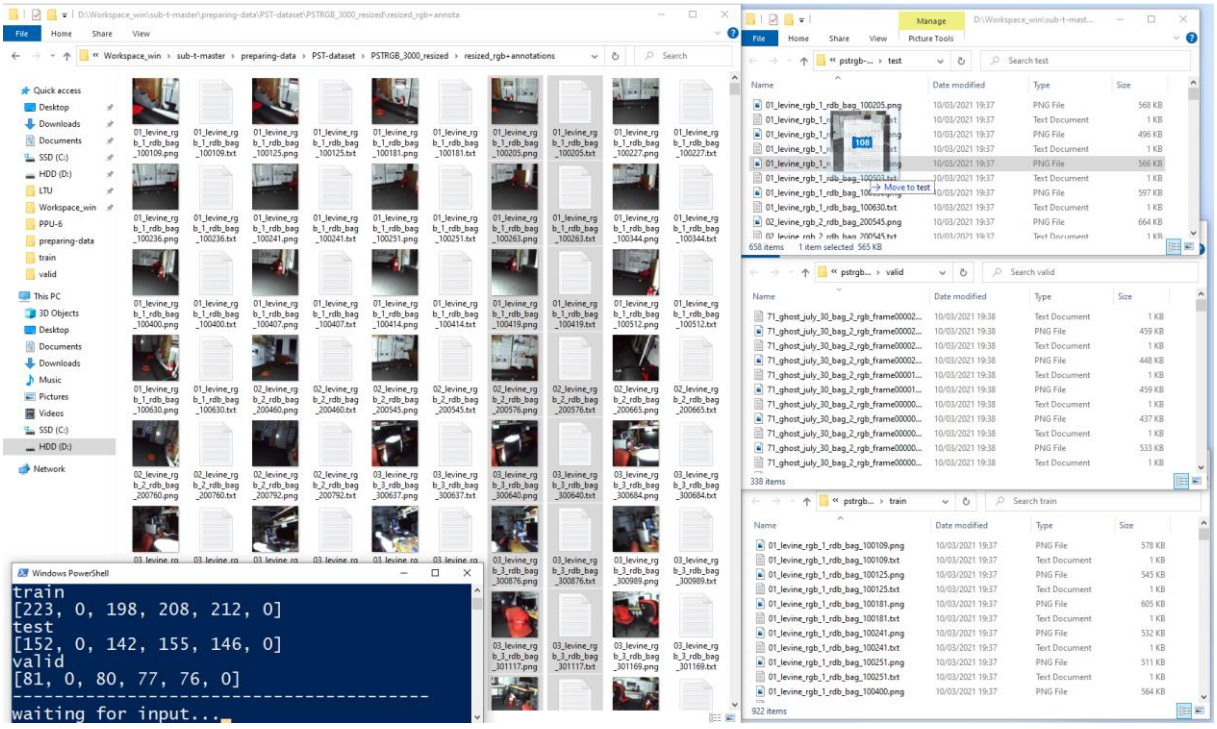


Figure 38: Overview of the split monitor tool usage.
Source: Self-made

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. 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, 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 relevant 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 39: 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.

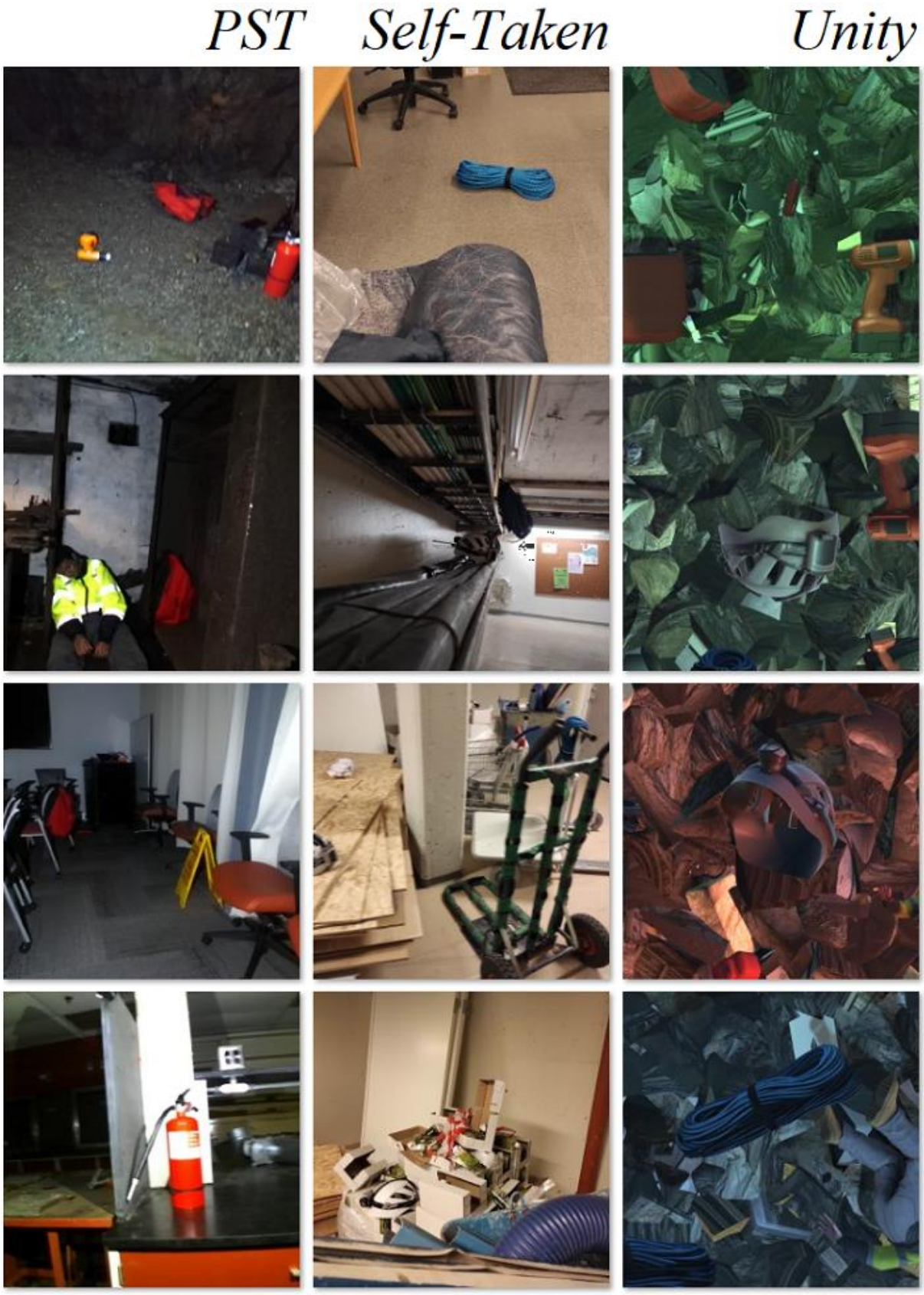


Figure 40: Showcase of the pictures in the PPU-6 dataset. By column: PST, self-taken and SynthDet samples Source: Self-made

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. 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, 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.
- **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.
- **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 [16], 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. 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, in which their speed and accuracy on the MS COCO dataset is compared. In FIGURE, 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 [25], [30], [33]. 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.

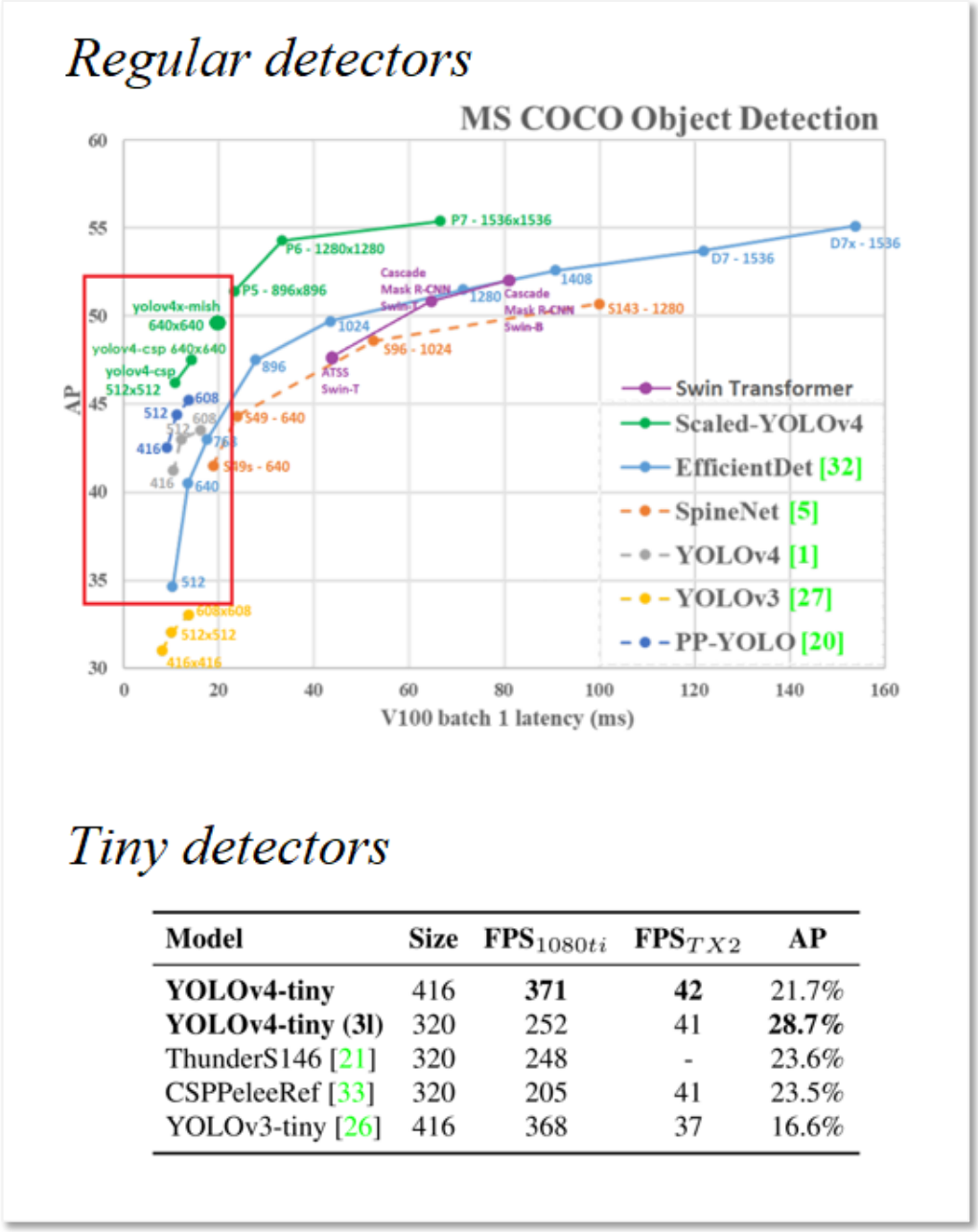
These models are all regular sized, meaning that, while one-stage, they still pursue high-accuracy. For faster applications, the trend has been to make scalable models that get smaller – and thus faster – when processing small-size pictures, at the cost of some accuracy. Aimed specifically for extremely 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.

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, in which each model resolution and implementation platform is also stated.

Resolutions are chosen to be close 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 41: 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: [31], [33]

Regarding implementation platforms, there are two different tools involved:

- Darknet: Which is the official publishing platform for YOLO models. Originally developed by Joseph Redmon [23], this work will use Alexey Bochkovskiy's fork, available at [31]. Darknet is an open-source library written in C and CUDA. Source is available at `https://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 `https://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 [23]. Darknet is easily imported and built from its github source [31] and holds the official implementation of YOLOv4, as well as some other models both from the YOLO family and not. The tutorial at [46] was used to get acquainted with YOLOv4 implementation in Darknet. While running the tutorial on a colab 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.
- *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 [46]. 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.

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 [31], 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 [47], 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 [48], structure then recycled for running the networks on custom data. This implementation is messier to work with than darknet.

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 terminal window. 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 [13]. 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 on the cloud using google colaboratory.
- Stage 1 YOLO models were trained on a remote laboratory computer.
- Remaining stages being performed on the author's *local machine*.

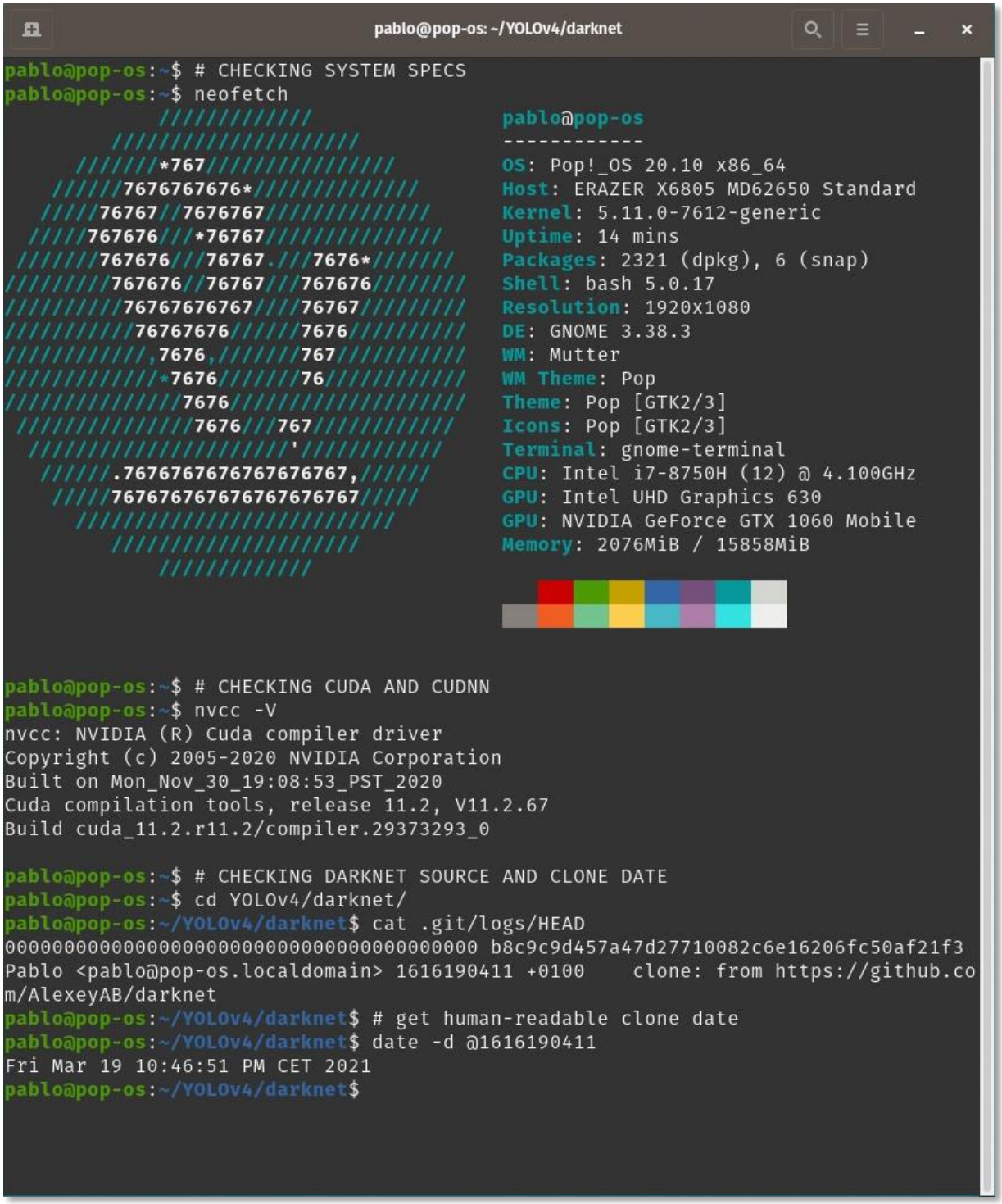


Figure 43: Local machine specifications
Source: Self-made

The specifications of the local machine used for training are shown in FIGURE, 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 [31].
- 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 [16].

The number of training epochs and other hyperparameters were set according to the pointers provided at [46].

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 [48], 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 [49].
- 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 [48].

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 [48].

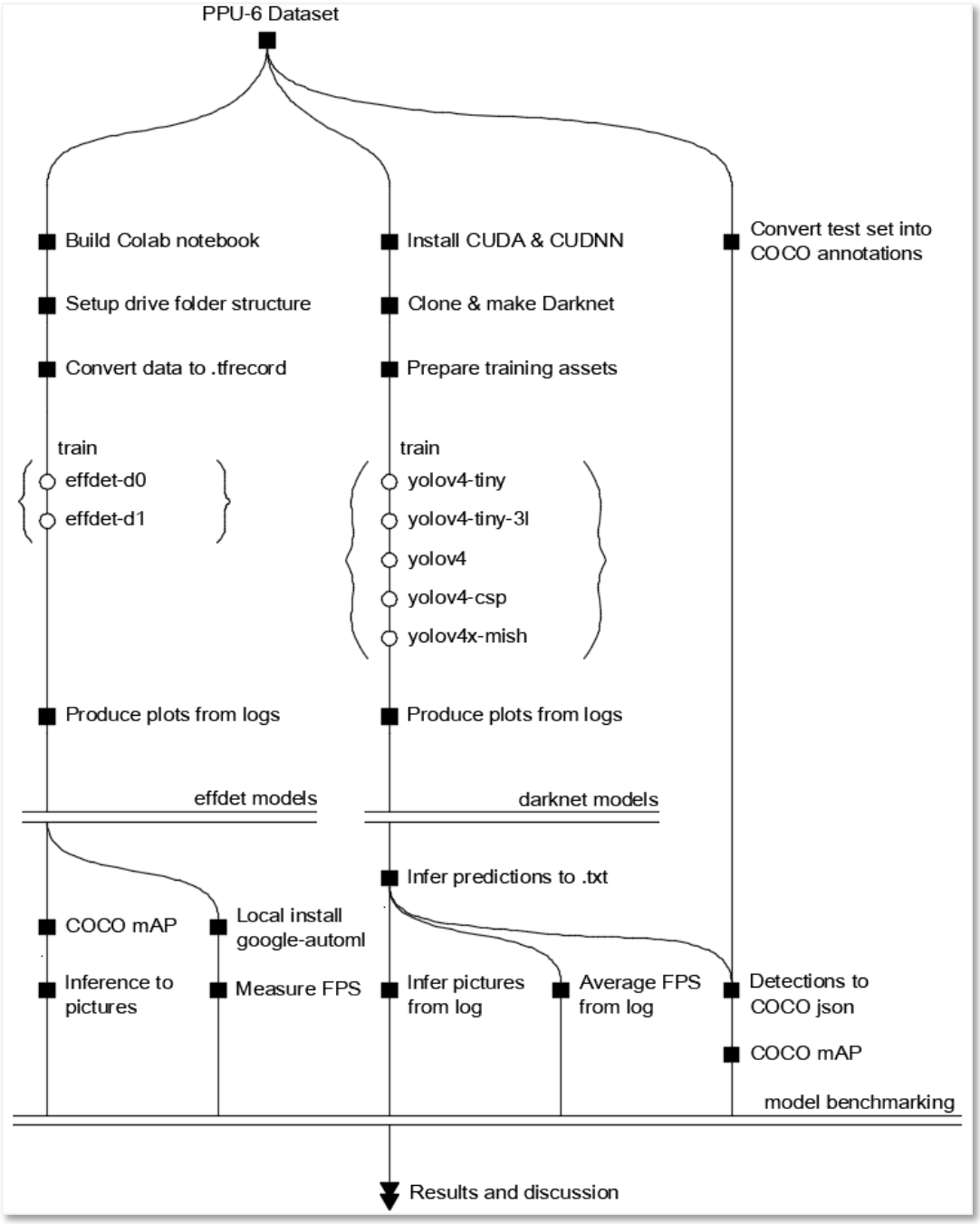


Figure 44: Training and evaluation roadmap
Source: Self-made

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

The benchmarking problem is approached by performing the steps previously described, repreating 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 results can be found in SECTION, a brief summary for them can be found at SECTION.

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, while SECTIONS 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 were trained under default configuration, as advised in [46] and [48], for the whole PPU-6 dataset, including synthetic samples. FIGURES 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.

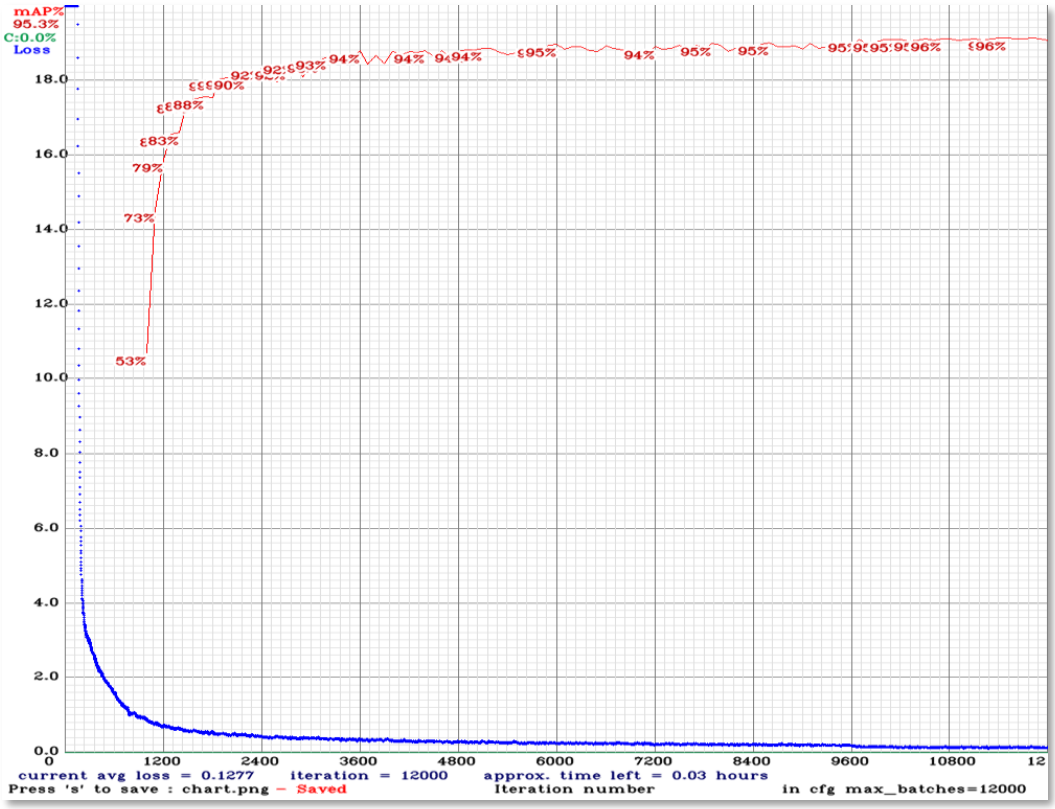


Figure 45: YOLOv4-tiny. Loss and mAP during training
Source: Self-made

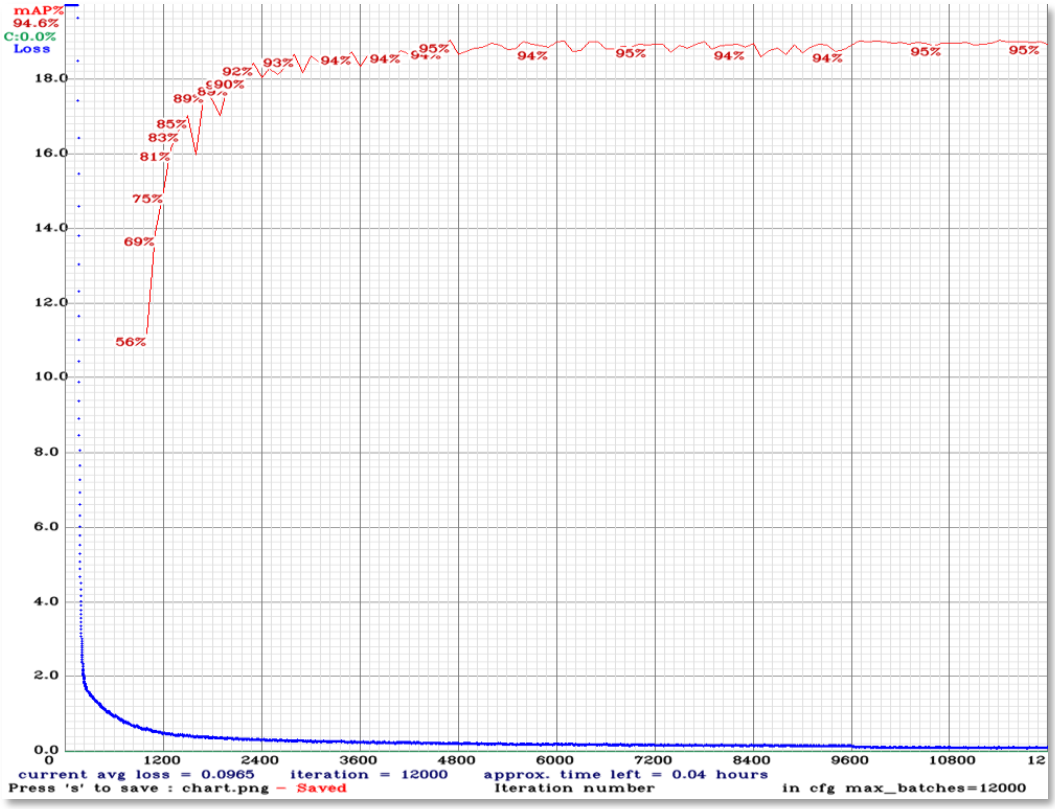


Figure 46: YOLOv4-tiny-3l. Loss and mAP during training
Source: Self-made

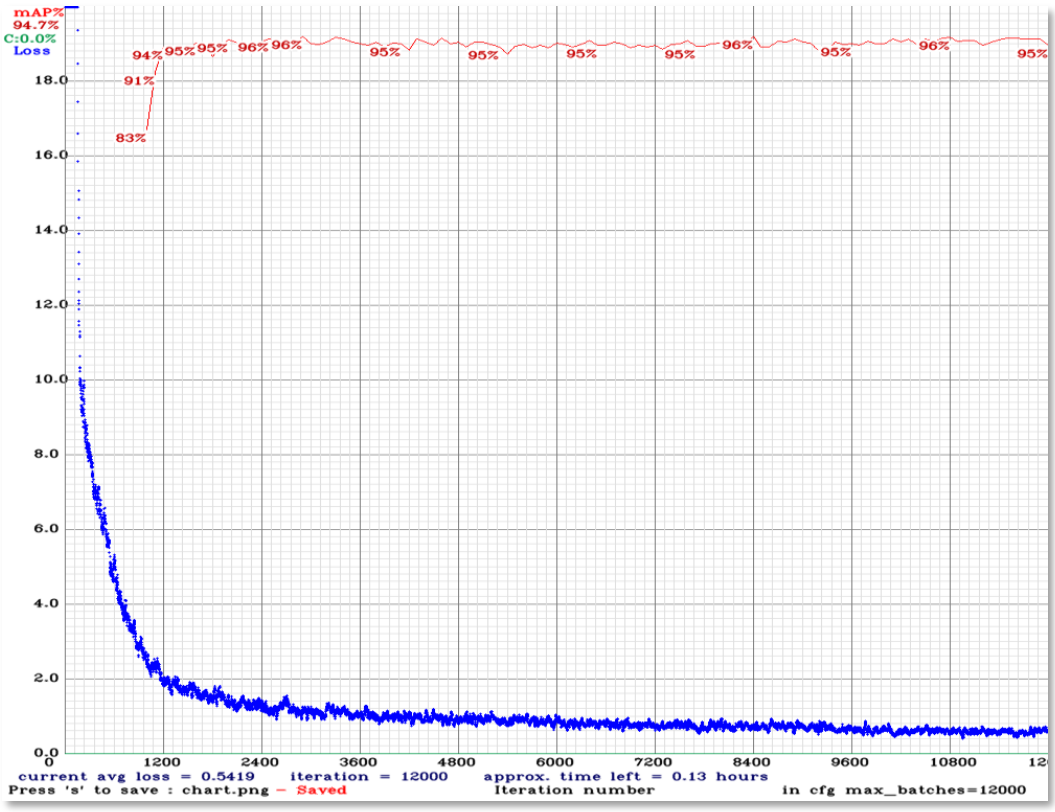


Figure 47: YOLOv4. Loss and mAP during training
Source: Self-made

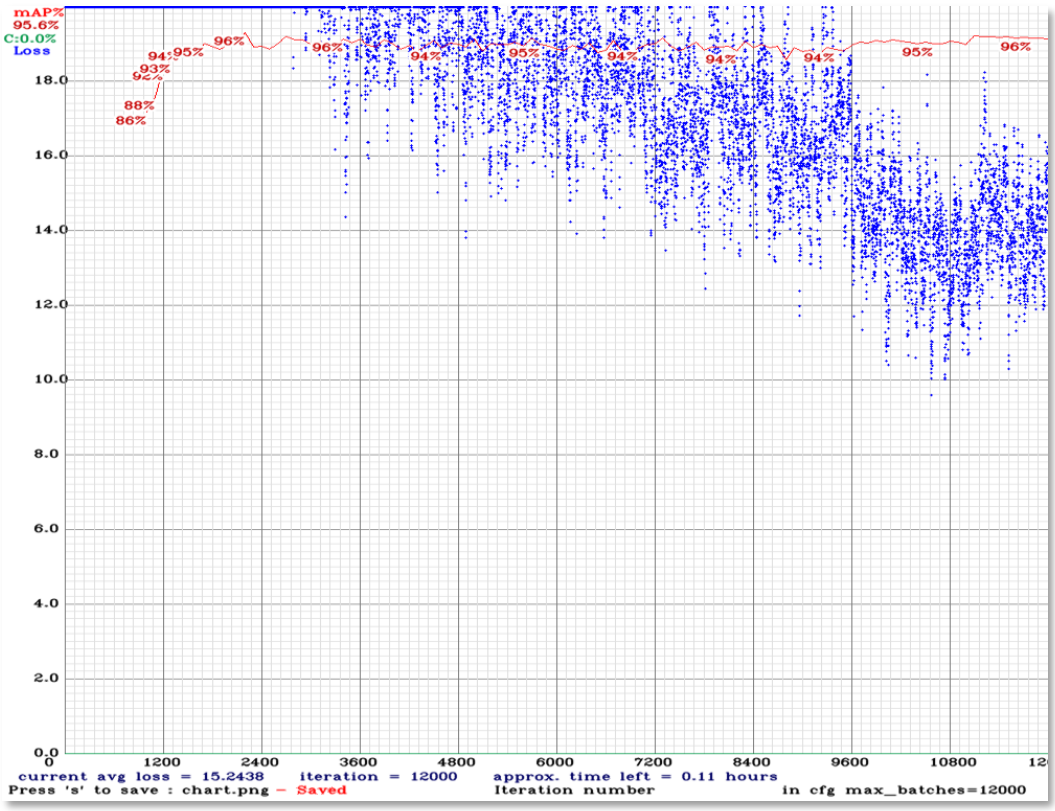


Figure 48: YOLOv4-CSP. Loss and mAP during training
Source: Self-made

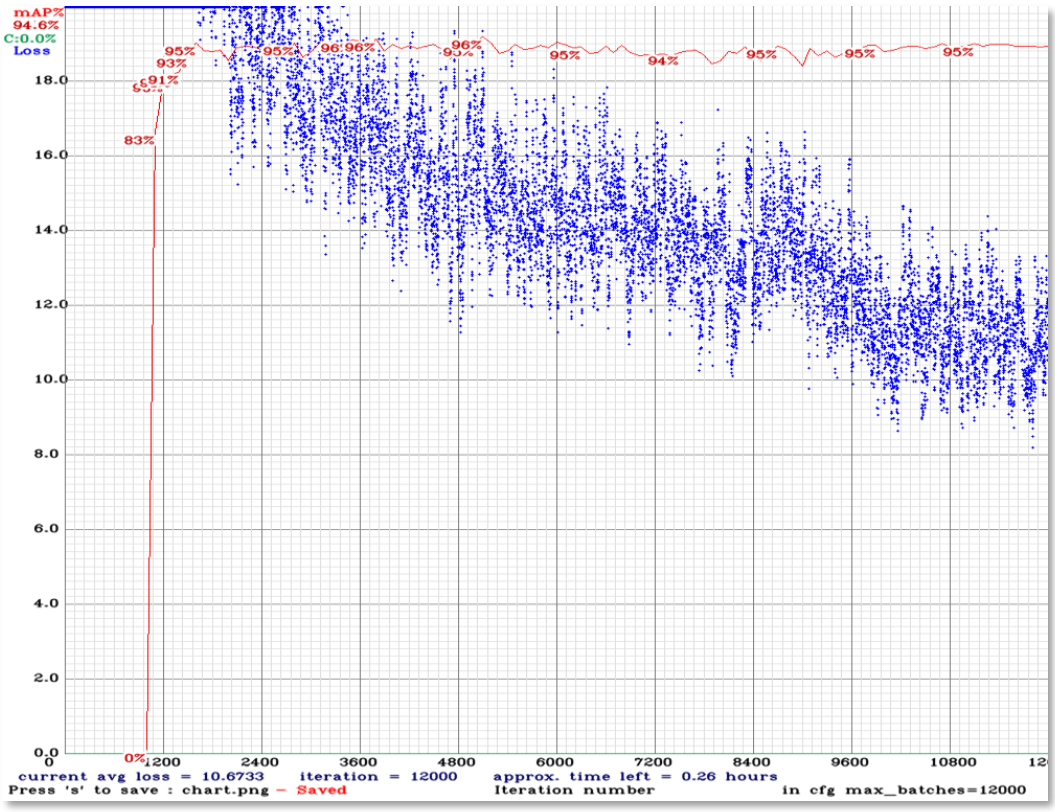


Figure 49: YOLOv4x-mish. Loss and mAP during training
Source: Self-made

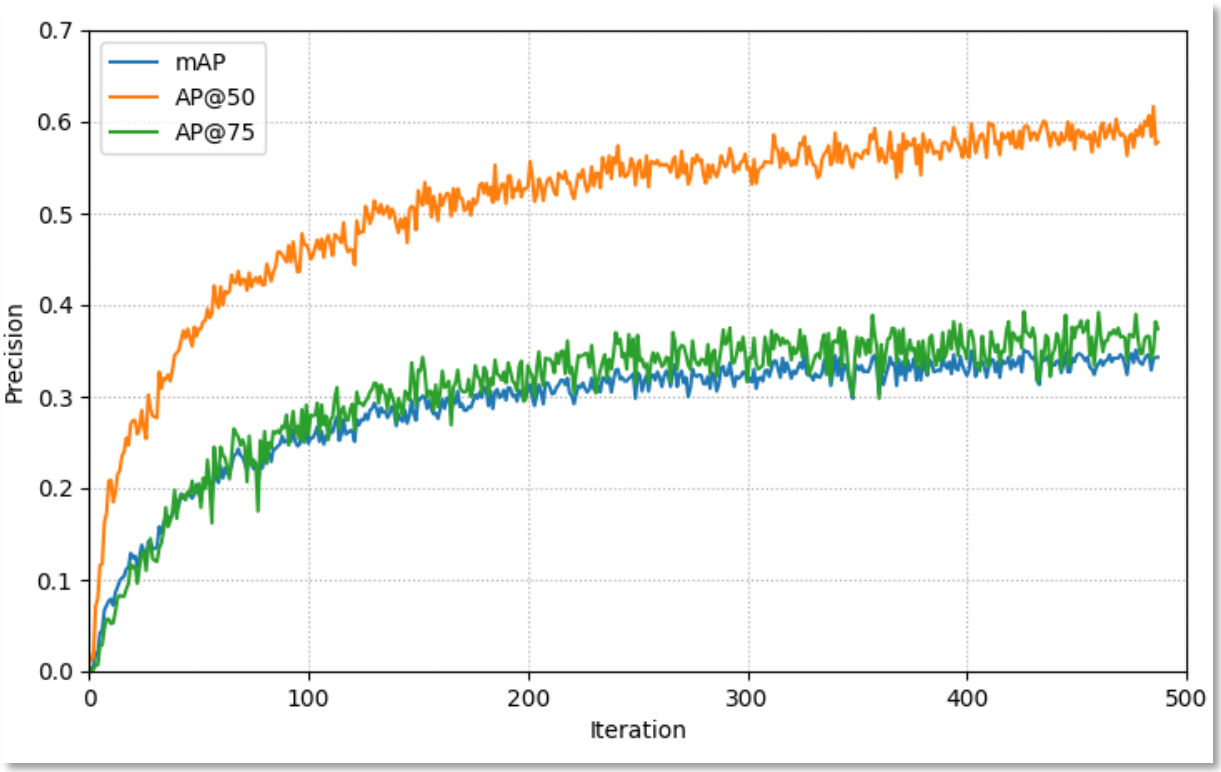


Figure 50: EfficientDet-D0. AP metrics during training
Source: Self-made

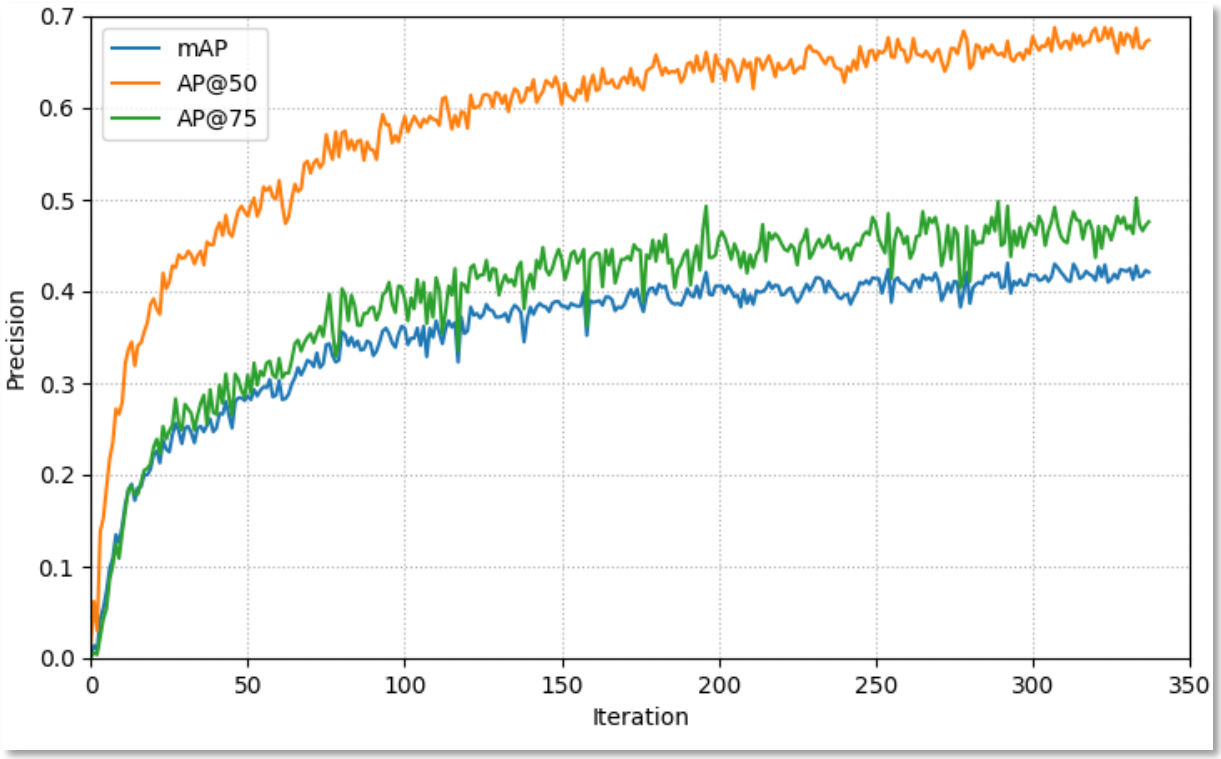


Figure 51: EfficientDet-D1. AP metrics during training
Source: Self-made

While training losses cannot be compared for different moels (since generally each of these models uses a different one), a common intuitive magnitude is the validation mAP. From FIGURES, 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 [50]). AP metrics are shown graphically in FIGURE. 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), the YOLO networks have greatly improved their performance. This is not so much true for the EfficientDet models, which have stalled into mAPs around ~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. 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 Darknet YOLOv4 (28 FPS) and YOLOv4-CSP (26 FPS) are close to each other, while YOLOv4x-mish (9 FPS) lags reasonably 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

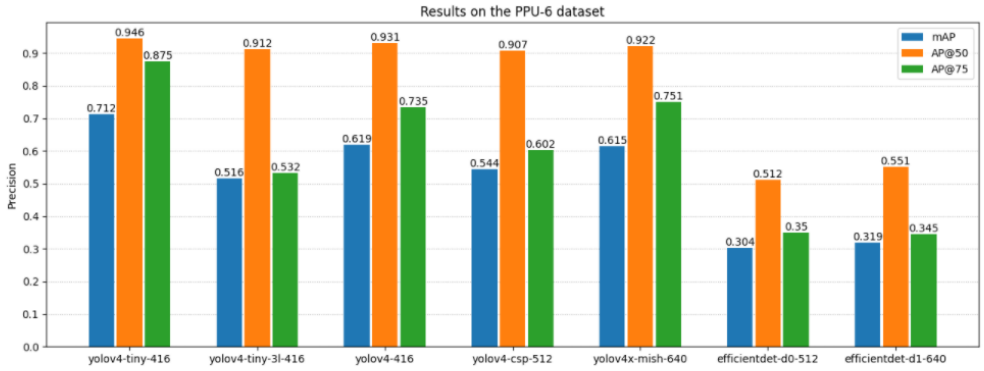


Figure 52: Benchmarking results on the PPU-6 dataset
Source: Self-made

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. FIGURES show the training metrics for each of the models involved in this stage.

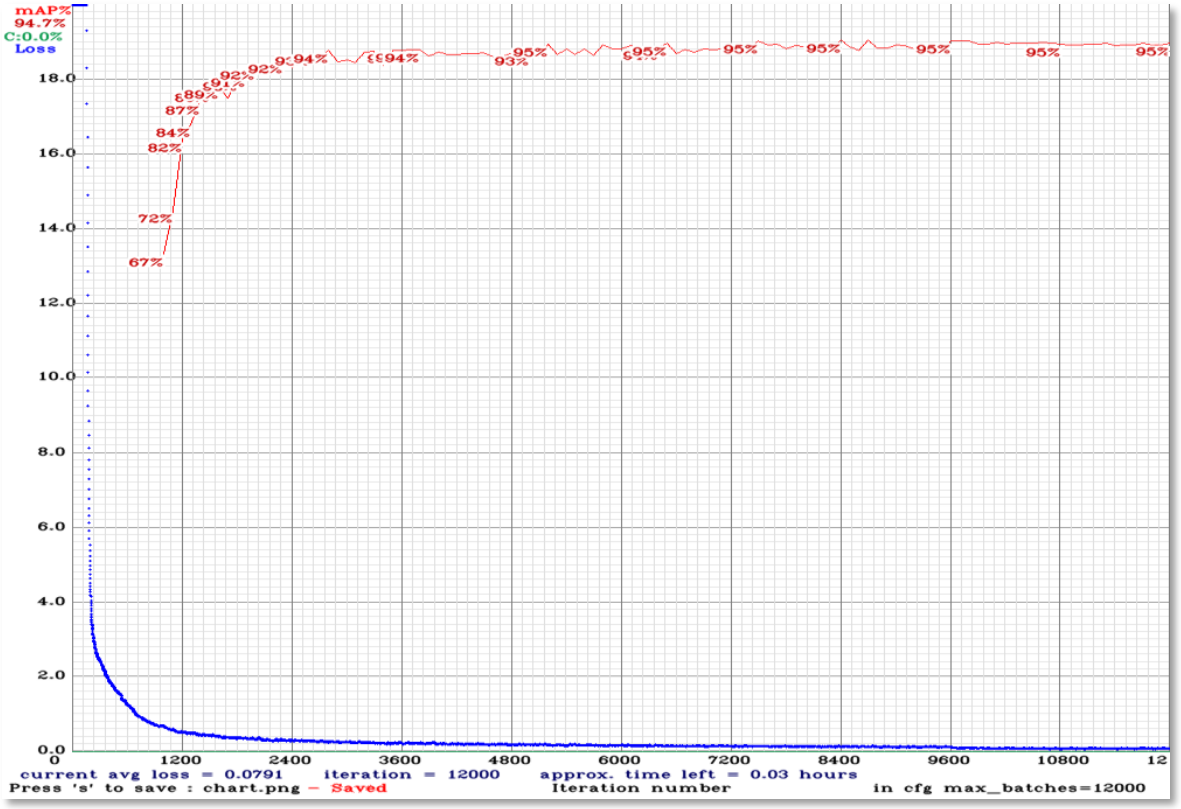


Figure 53: YOLOv4-tiny
Source: Self-made

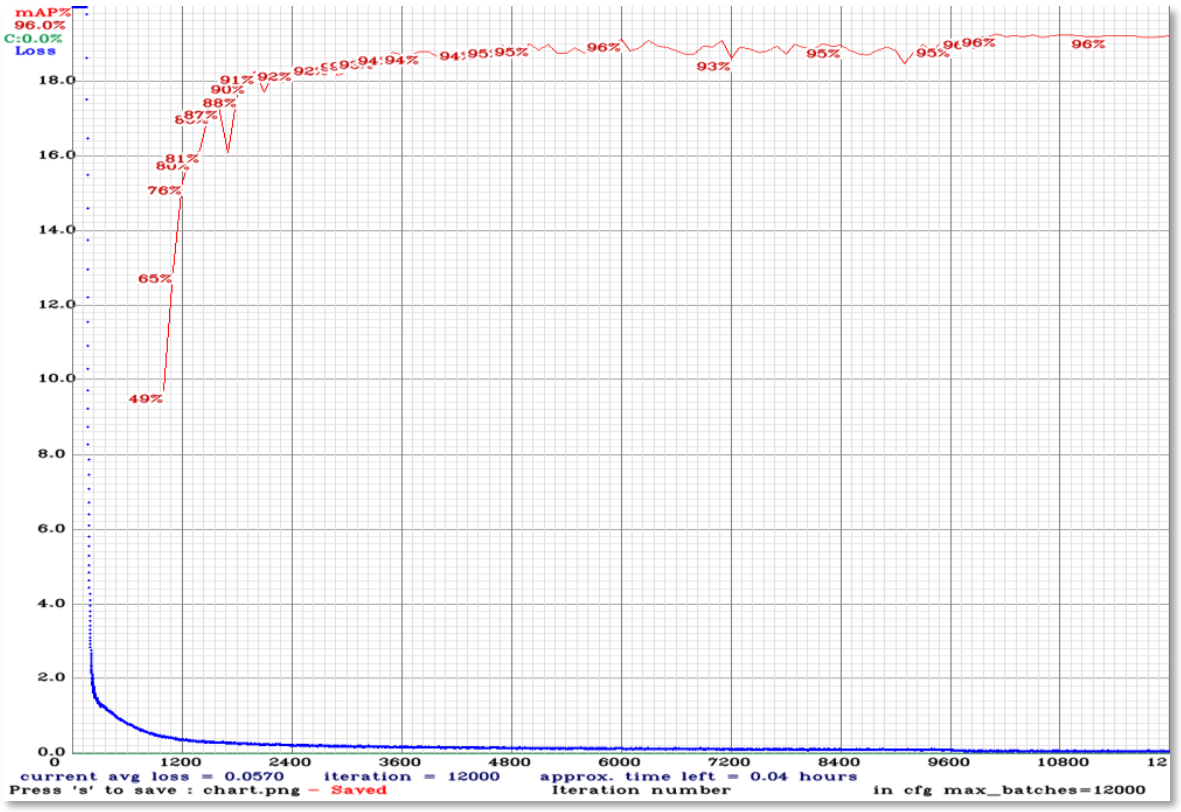


Figure 54: YOLOv4-tiny-3l
Source: Self-made

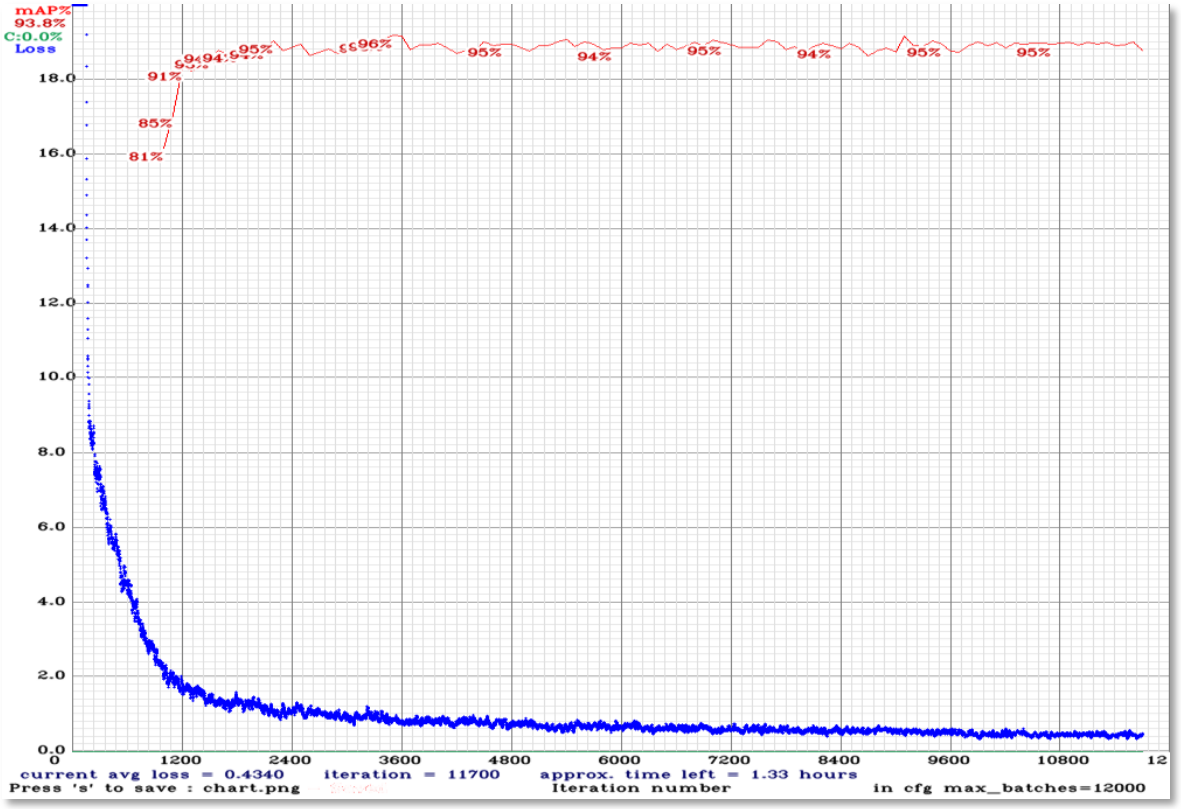


Figure 55: YOLOv4
Source: Self-made

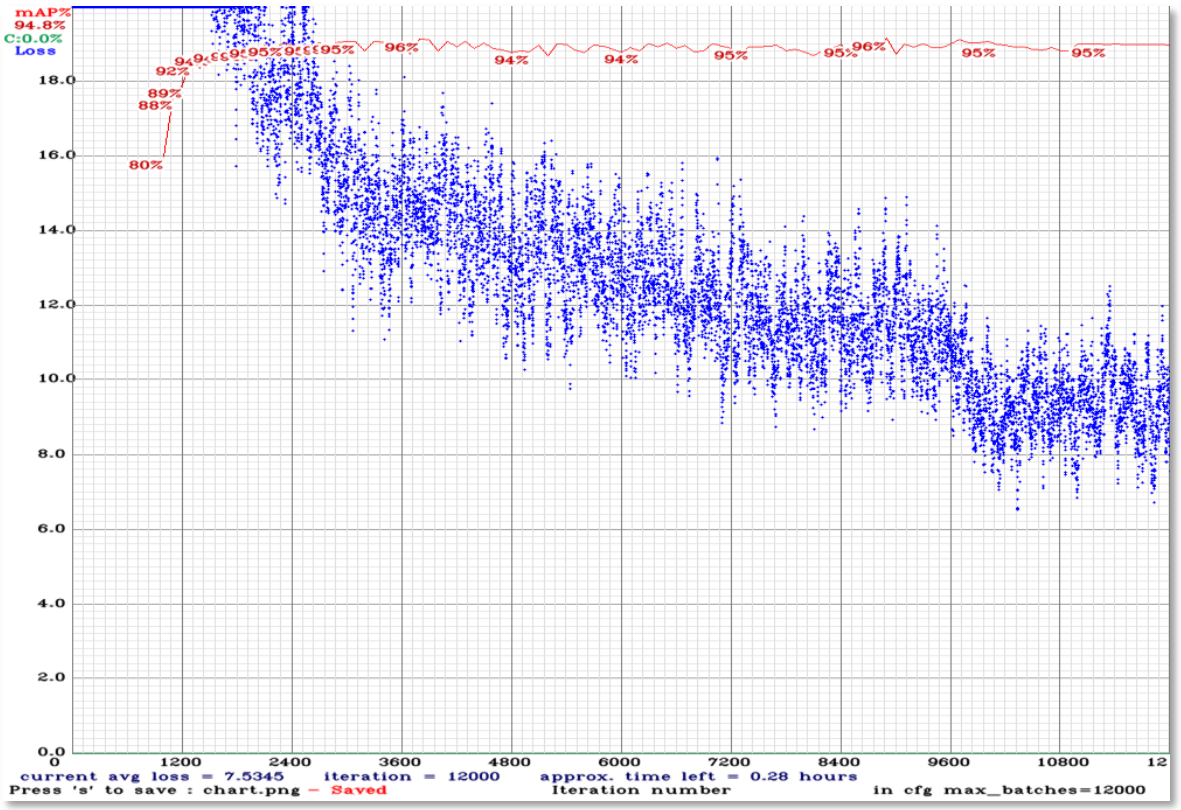


Figure 56: YOLOv4-CSP
Source: Self-made

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, this stage took place in a local machine so the training times are significative (because the hardware is known). TABLE shows the time it took for each network to produce the best validation accuracy and to complete its training.

Table 57: 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). 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, 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

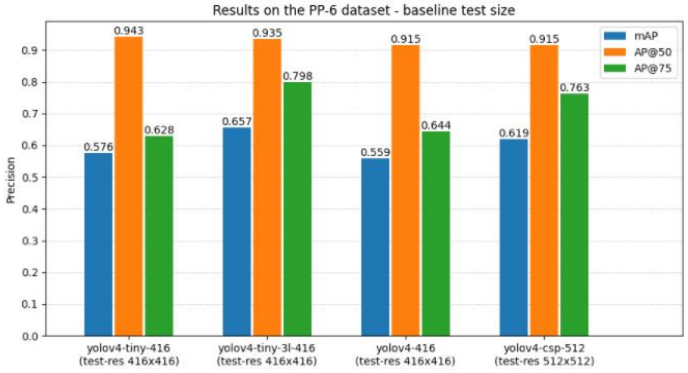


Figure 58: Benchmarking results on the PP-6 dataset. Stage 2.1
Source: Self-made

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 [31]. 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.

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. In this experiment, 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

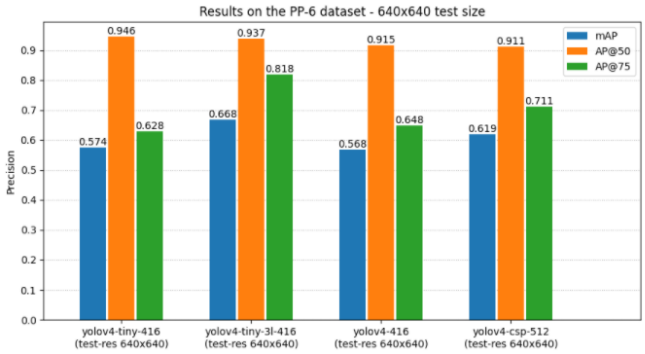


Figure 59: Benchmarking results on the PPU-6 dataset. Stage 2.2
Source: Self-made

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

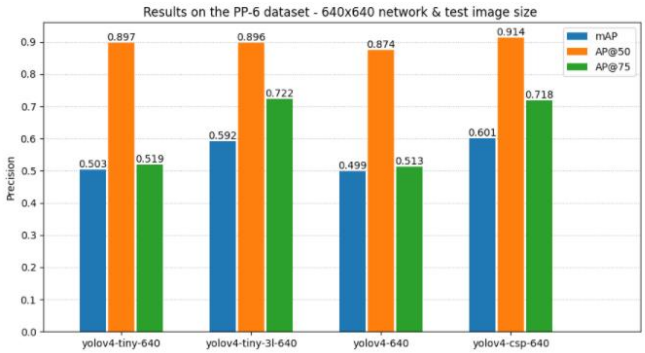


Figure 60: Benchmarking results on the PPU-6 dataset. Stage 2.3
Source: Self-made

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 [#ref[]], 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 (metrics of which are plotted in FIGURES and S) are shown in FIGURE. 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

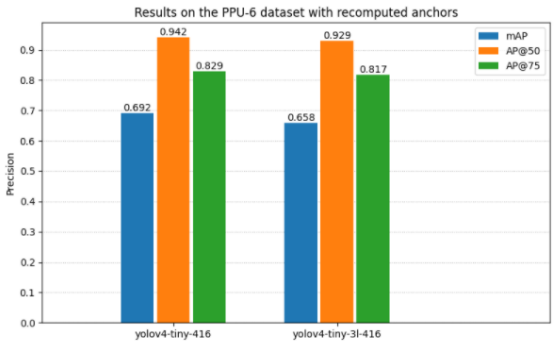


Figure 61: Benchmarking results on the PPU-6 dataset. Stage 3
Source: Self-made

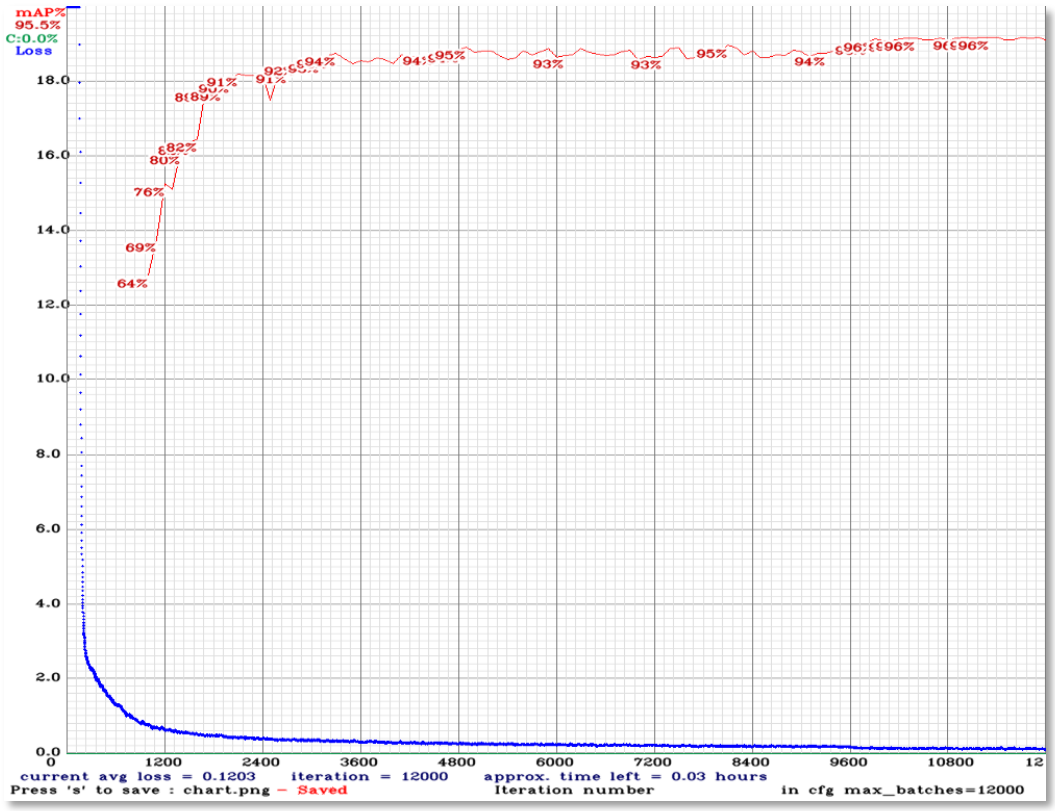


Figure 62: yolov4-tiny
Source: Self-made

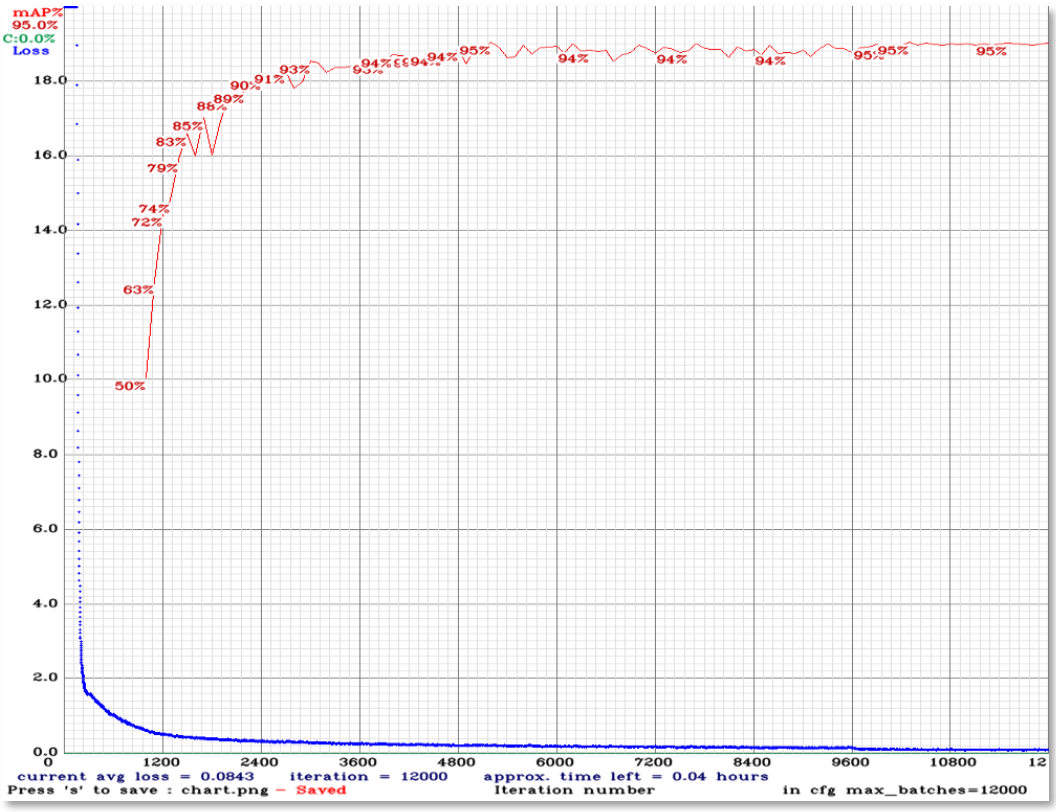


Figure 63: yolov4-tiny-3l
Source: Self-made

4.4 Additional experiment on synthetic data

4.4.1 The unity-1000 dataset

4.4.2 Unity-1000 dataset + fakes

4.5 Results

4.5.1 Benchmarking results

TABLE shows a full summary of all of the models discussed in SECTIONS. The initials in it stand for the discussed extra features:

- *SD* : Synthetic data.
- *ITS* : Increased test size.
- *INS* : Increased network size.
- *RA* : Recomputed anchors.

Table 64: Object detector benchmarking on the PPU-6 dataset

Model	Size	SD	ITS	INS	RA	AP50 %	AP75 %	mAP %	FPS
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

4.5.2 Best detector proposal

These experiments identify as overall best the model *YOLOv4-tiny with synthetic samples* for object detection on the PPU-6 dataset. Deeper discussion on this topic can be found at SECTION.

5 TOWARDS ITEM LOCALIZATION

5.1 The localization problem

5.1.1 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 [51]. 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 [51].

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 $\hat{\boldsymbol{x}}$ [51]. It is also common to approach the position of the robot with a probability density function or PDF. The intuition behind this concept is shown in FIGURE. 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, intuitively the same but 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) [51]. 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 [51].

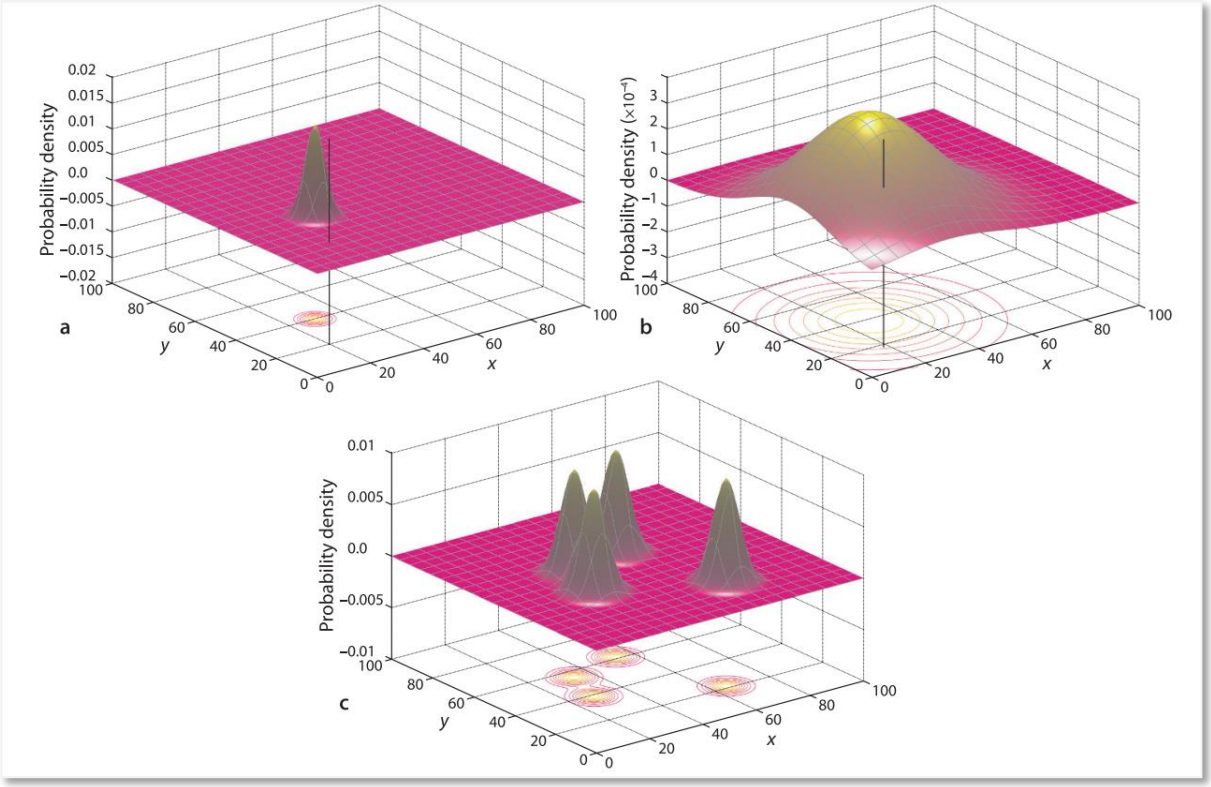


Figure 65: Likelihood of a vehicle's true position expressed as a PDF.
Source: [51]

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 a 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 [52]. 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.

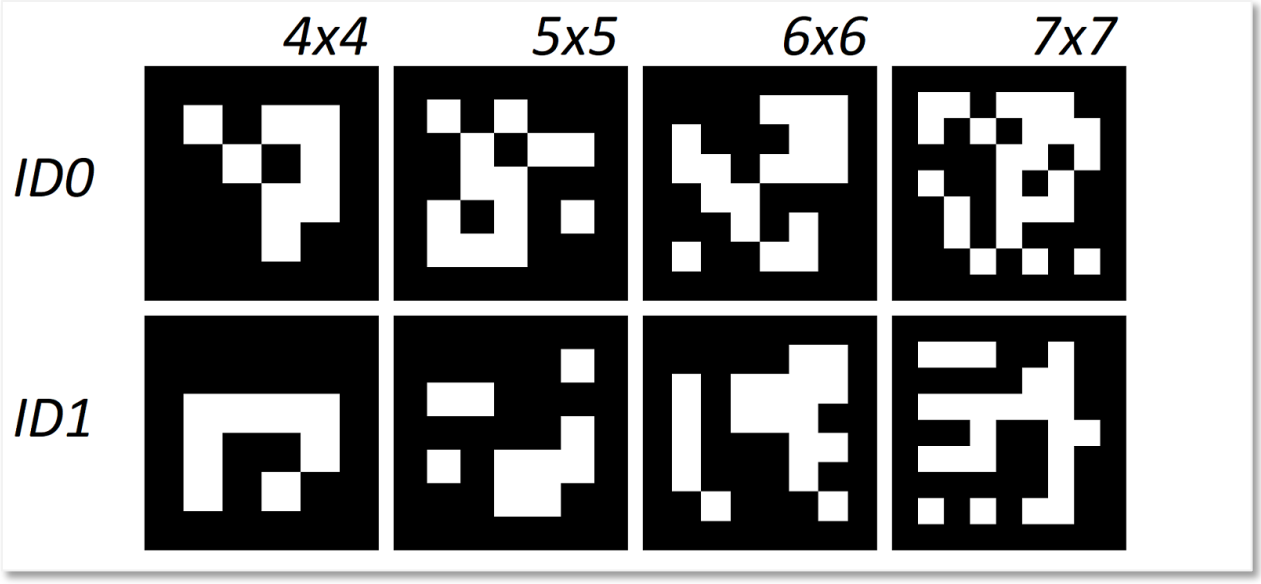


Figure 66: Some examples of ArUco markers 0 & 1 from different libraries.
Source: Self-made

The fiducial system of choice for camera pose estimation within this work are the ArUco markers, introduced at [52]. 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 [52] 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 contains some examples of ArUco markers from different libraries generated with the tool at [53].

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.

5.1.2 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 [51]. 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 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, 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 [51], while this work will now focus on an example-led appraoch to show the numerical methods used to compose different poses.

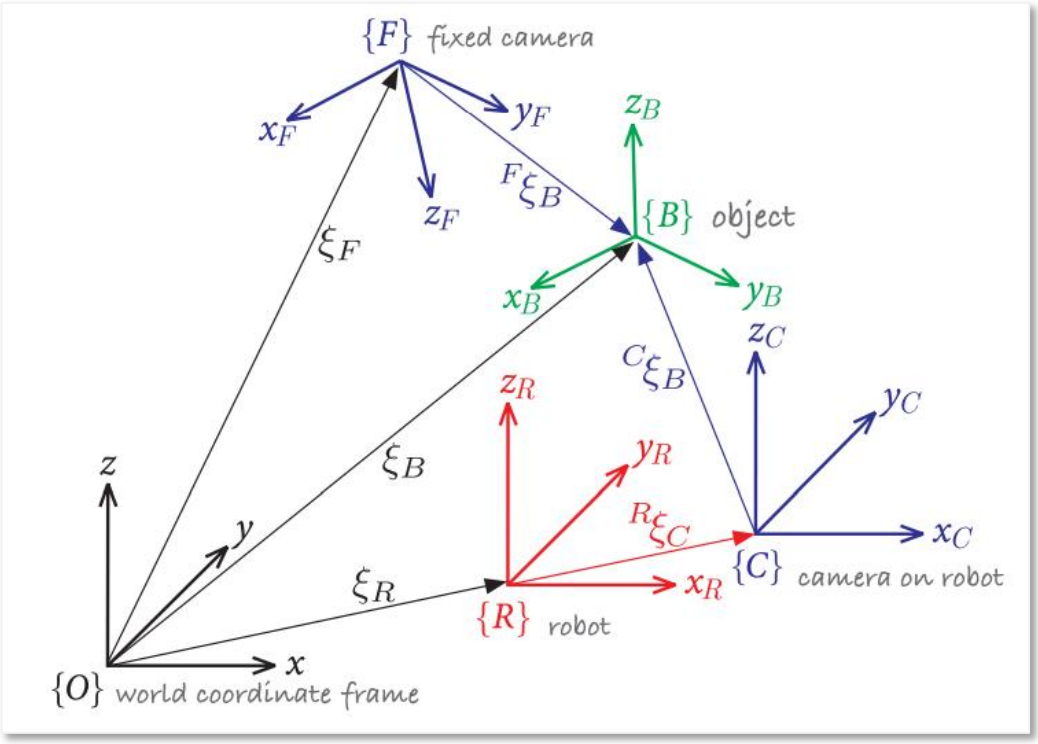


Figure 67: Multiple 3-dimensional coordinate frames and relative poses
Source: [51]

Take the 2D, cartesian system shown in FIGURE.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: ${}^{A}\xi_{B} \sim (x, y, \theta)$.

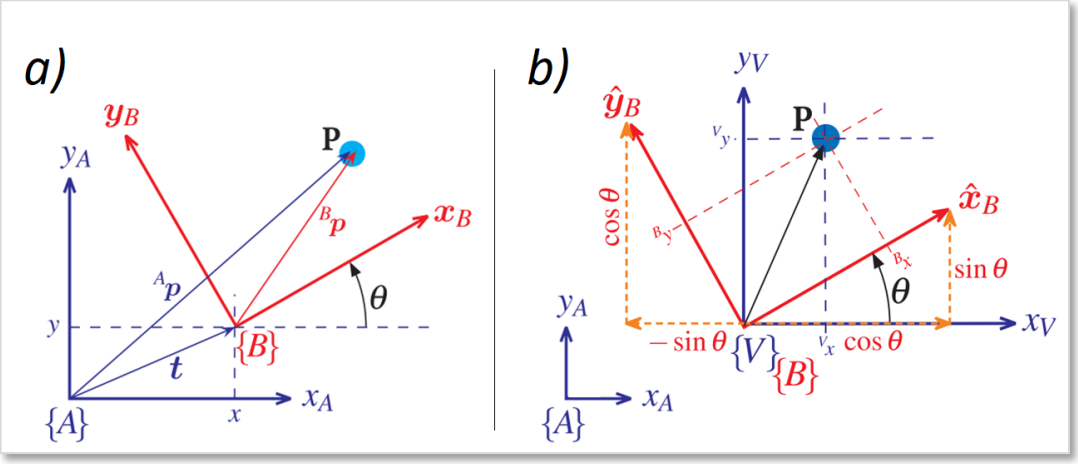


Figure 68: Example coordinate change in the cartesian plane
Source: [51]

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.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, ${}^{A}\boldsymbol{R}_{B} := {}^{V}\boldsymbol{R}_{B}$:

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

Where the Homogeneous Transformation ${}^{A}\boldsymbol{T}_{B}$ is defined. Note that, from the first part of the equation, 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 below:

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

For more information on this topic, and a detailed explanation on the contents here reviewed please refer to [51].

5.1.3 State space models and Kalman filters

A *state space representation* [54] 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. The 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 model of a generic discrete physical system is shown in #EQUATION:

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

In the propagation equation, the future state of the state vector X_{k+1} is computed from its current state X_k and input 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 w_k.

In the measurement equation, the current output of the system Y_k is modelled from the current state vector X_k and the current input u_k. The vector Y_k is composed by those magnitudes that can be measured by real sensors. A measurement noise random variable v_k is added to account for the sensor noise that makes the real measurements mismatch the theoretical model.

The Kalman Filter [55] 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 [55]:

Propagation cycle (time update)

$$\hat{\boldsymbol{x}}(k+1 \mid k) = \boldsymbol{A}(k) * \hat{\boldsymbol{x}}(k \mid k) + \boldsymbol{B}(k) * \boldsymbol{u}(k)$$
$$\boldsymbol{P}(k+1 \mid k) = \boldsymbol{A}(k) * \boldsymbol{P}(k+1 \mid k) * \boldsymbol{A}^{\boldsymbol{T}}(k) + \boldsymbol{\Gamma}(\mathrm{k}) * \mathbf{Q}(\mathrm{k}) * \boldsymbol{\Gamma}^{\boldsymbol{T}}(\mathrm{k})$$

Upgrade cycle (measurement update)

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

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 [56] is highlighted.

5.1.4 The pinhole camera model

The pinhole camera model [57] provides a simplified mathematical representation of a camera device, setting the foundings that allow relating real world positions and picture coordinates. The pinhole camera model is displayed in #FIGURE. 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.

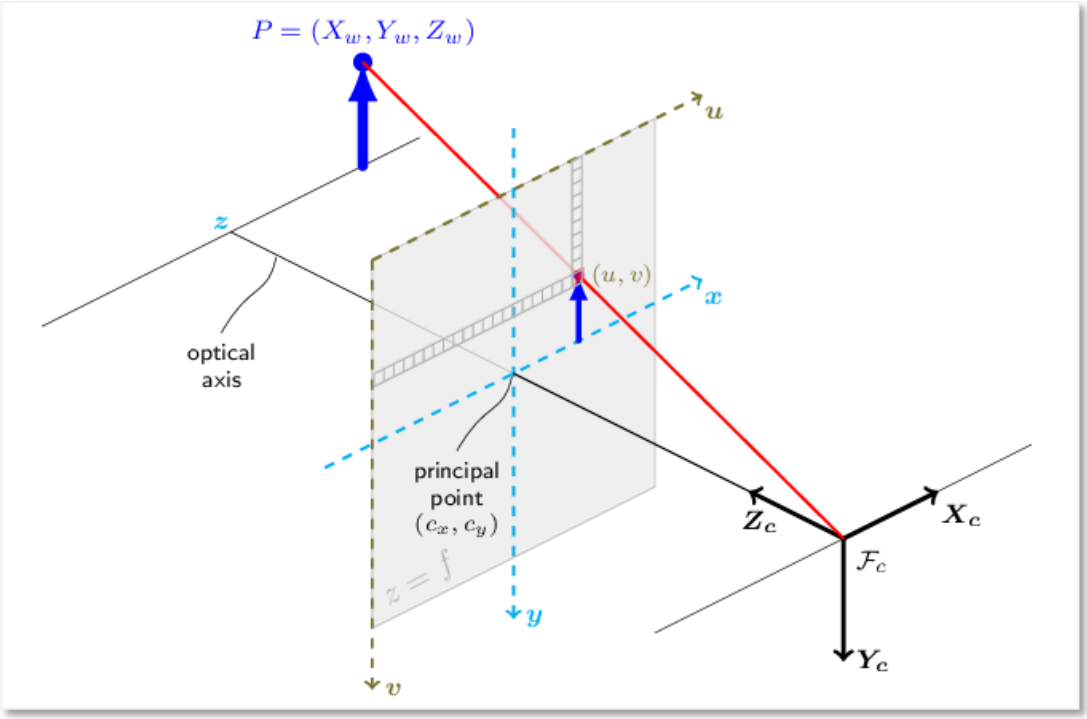


Figure 69: The pinhole camera model
Source: [57]

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 #EQUATON. In it, we can find the following terms:

- $\boldsymbol{P_w}$: 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.

$$s * \boldsymbol{p} = \boldsymbol{A} * [\, \boldsymbol{R} \mid \boldsymbol{t} \,] * \boldsymbol{P_w}$$

Which simplifies to the following expresion 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 the following equivalence:

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

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 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 coordinate, 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 [57]. FIGURE 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 [57]:

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

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

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

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

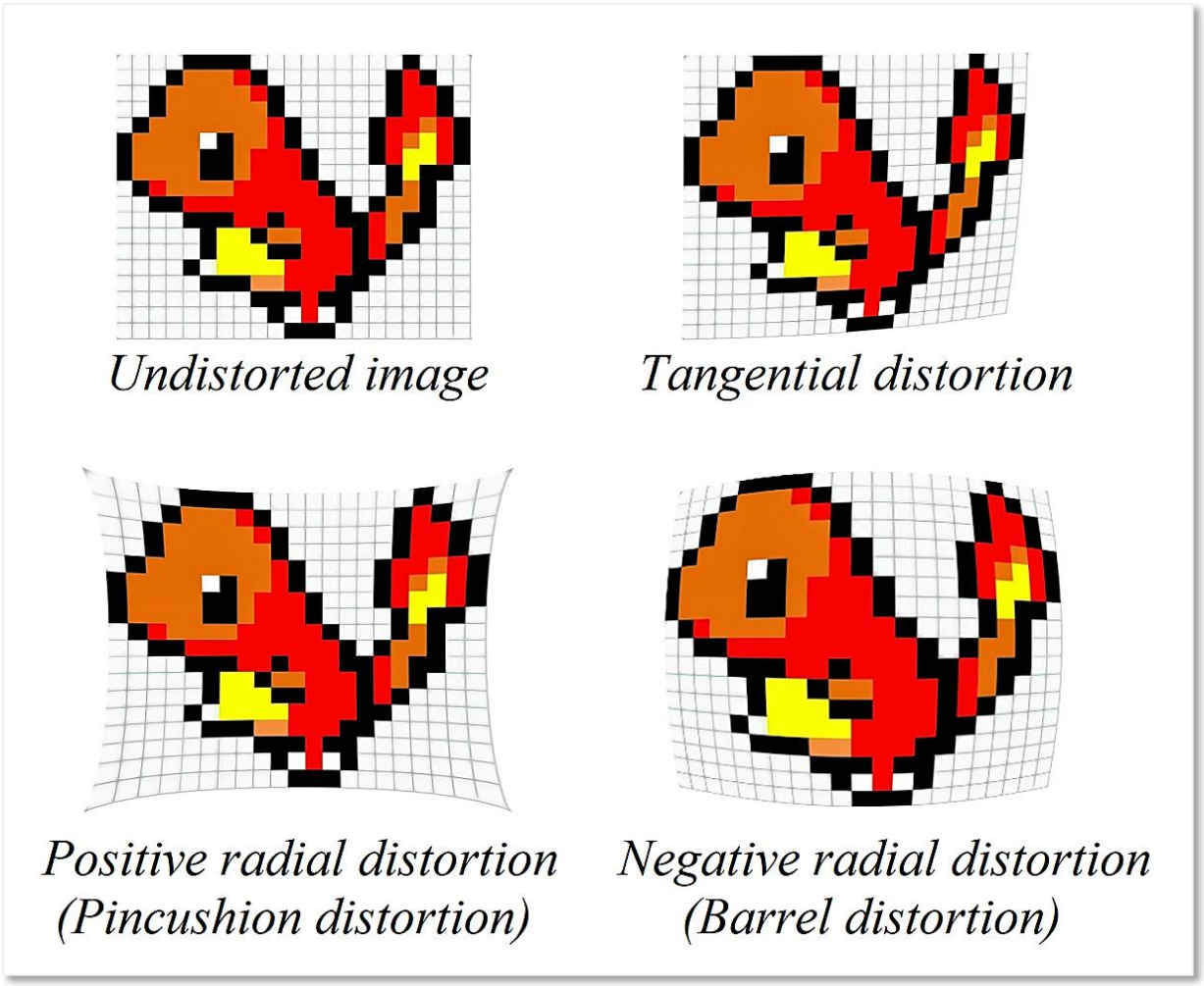


Figure 70: Tangential and radial distortions in images
Source: Self-made

5.2 Experimental setup

As it has already been stated, solving the localization problem for a camera usually generalizes to solving it for the robot or autonomous device. The work shown in this section attempts to solve the horizontal localization problem for a camera device by using only consumer-degree technology. The main assets used on the development of this solution were:

- 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.

Live feed is to be taken from an Android device and sent over 2.4GHz or 5GHz signals (WiFi) to a local server hosted on the smartphone. The IP Webcam app [58] 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 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. Not only video feed, but readings from a variety of built-in Android sensors are also published to this server. 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.

5.3 Camera pose estimation with ArUco markers

5.3.1 Camera calibration

The camera calibration is performed by using a ChArUco board. 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, since there are more points for reference and occlusions and partial views are allowed [59].

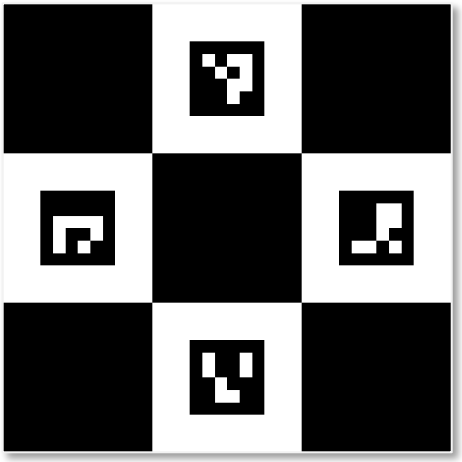


Figure 71: Example ChArUco pattern
Source: Self-made

Camera calibration is performed by taking a set of pictures 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 $(k_1, k_2, k_3, p_1, p_2)$. For simplification, the rest of distortion coefficients shown in the distortion-correction equations are assumed to be zero [60].

Because the video feed sent by IP Webcam is of unknown characteristics (camera source, resolution, in-line processing... etc.), images straight up taken with the phone camera in a conventional manner should not 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.

5.3.2 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 [61], which returns the position and rotation vectors of the ArUco marker with respect to the camera.

5.3.3 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 Rodrigues' Rotation formula [57] 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:

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

Finally, after obtaining the position, the camera yaw θ is computed by obtaining the Euler angles [51] from the projection matrix ${}^{A}\boldsymbol{T}_{C} \coloneqq {}^{C}\boldsymbol{T}_{A}{}^{-1}$.

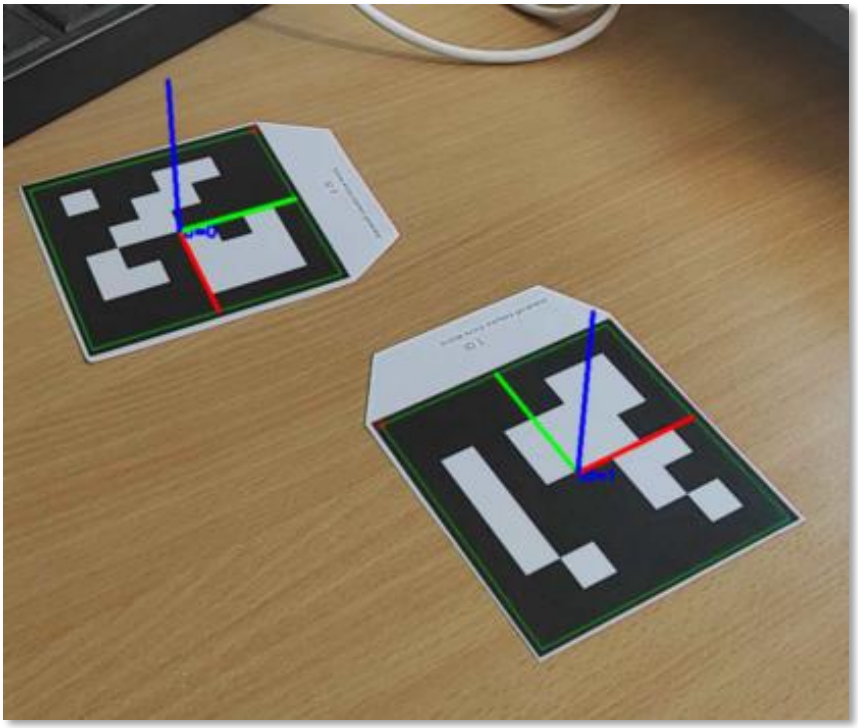


Figure 72: Detection of ArUco markers
Source: Self-made

#FIGURE shows two ArUco markers and their poses being identified by the described procedure. The coordinate frames painted on these markers follows the RGB convention. If we define one of these as the absolute origin, we can also stablish a criteria to be followed through this work, and thus the axes pair as follows:

- R : *X* axis, horizontal.
- G : *Y* axis, vertical aiming upwards.
- B : *Z* axis, horizontal aiming towards the picture viewer.

Because we are only interested in solving the horizontal localization problem, the pose of the camera will be defined by $({}^{A}x, {}^{A}z, {}^{A}\theta)$, where the superscript ^{A} denotes the reference ArUco marker. In this representation, the horizontal plane comes defined by this marker, and thus a different pair from x, y, z axes might need to be used to denote horizontal coordinates depending on the placement of the reference fiducial marker in a different setting.

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, origin ArUco representing the world's coordinate frame. However, while doing real-time tests it is observed that the position signal is slightly noisy.

It is important to note that this localization method does not pursue a high-accuracy localization. An approximate position of the camera with an error in the order of a few hundreds of milimeters is deemed a reasonable outcome to this implementation *a priori*.

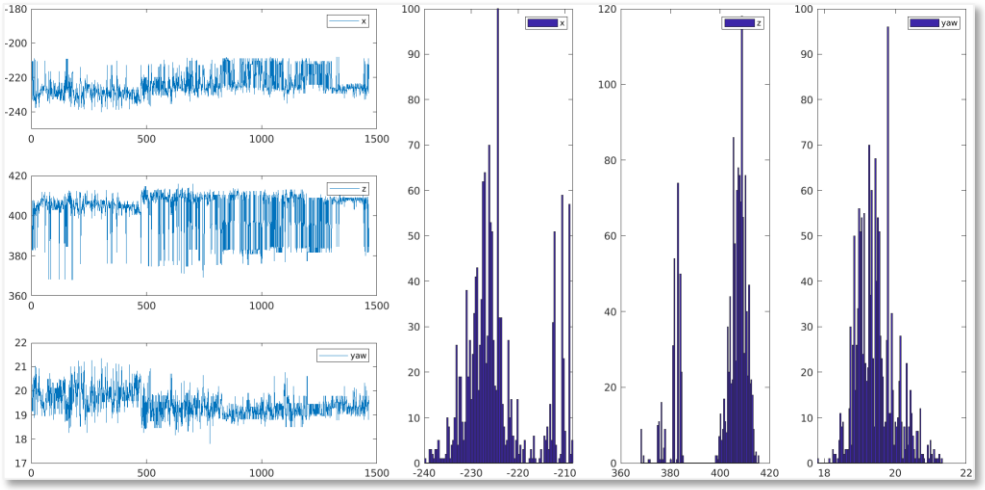


Figure 73: Static position error distribution on ArUco-based localization
Source: Self-made

In the first experiments, the measurements (numerically displayed on a terminal window) showed certain unstability with slight resemblance to white noise. An experimental test was then perfomed to get a profile of the noise and check the characteristics of the measured pose signals. In this test, the camera was placed static on a tripod while aiming at an ArUco marker, the position at each instant being recorded in disk. A time-series and histogram of the recorded data is shown in FIGURE. From these, it is observed that the noise does not follow a Gaussian distribution. Also, the current perturbation is actually very small compared to the expected location accuracy: the peak-to-peak variation of the x-positioning is smaller than 50mm for a camera placed ~500mm away from the ArUco marker. The positioning errors are thought to come from the following sources:

- Artifacts due to compression in the livestream footage.
- Artifacts due to lighting conditions changes: people passing by, ambient ligh... 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.

The most predominant of which being the two former ones, that slightly displace the ArUco marker a few pixels. This discrete displacement could explain the discontinuous noise profile that the histograms show.

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 contition fails with suboptimal outcome. Despite the sensor noise being non-gaussian and reasonably small, 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 use of sensor fusion to combine the visual ArUco positioning measurements 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 was 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 the system is shown in FIGURE. 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.

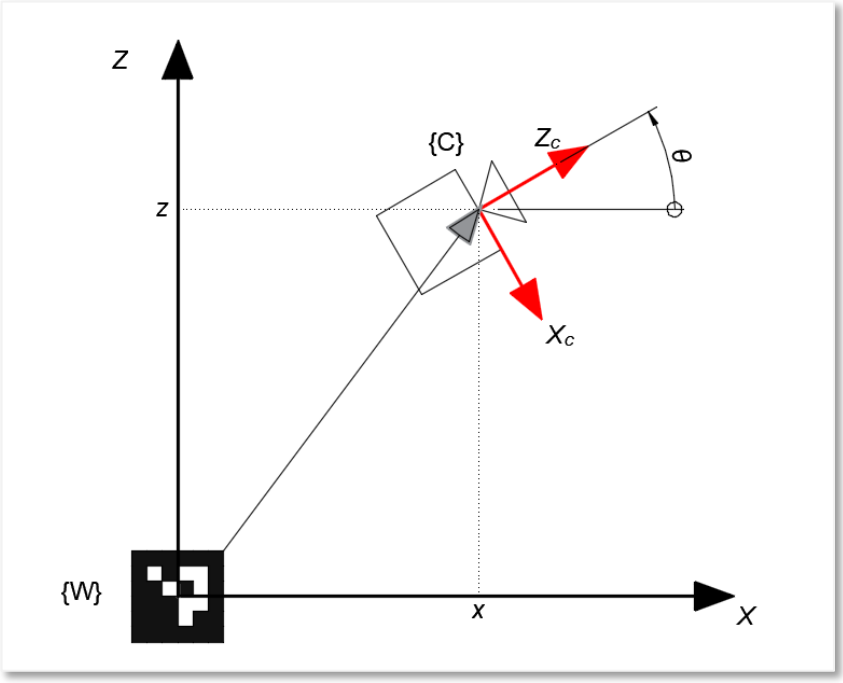


Figure 74: Example dummy schematic of the horizontal single-ArUco localization state space model layout
Source: Self-made

And thus, the mathematical representation of this state space model is laid out as:

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

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.

The implementation of the filter for denoising a pose signal computed from a single ArUco marker is succesfully completed under these assumptions. FIGURE shows a time-series of the yaw θ and the position x, both processed and unfilered. 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, though for our purposes this is not an issue since the position tolerance is high and biases of ~30mm are deemed admisible.

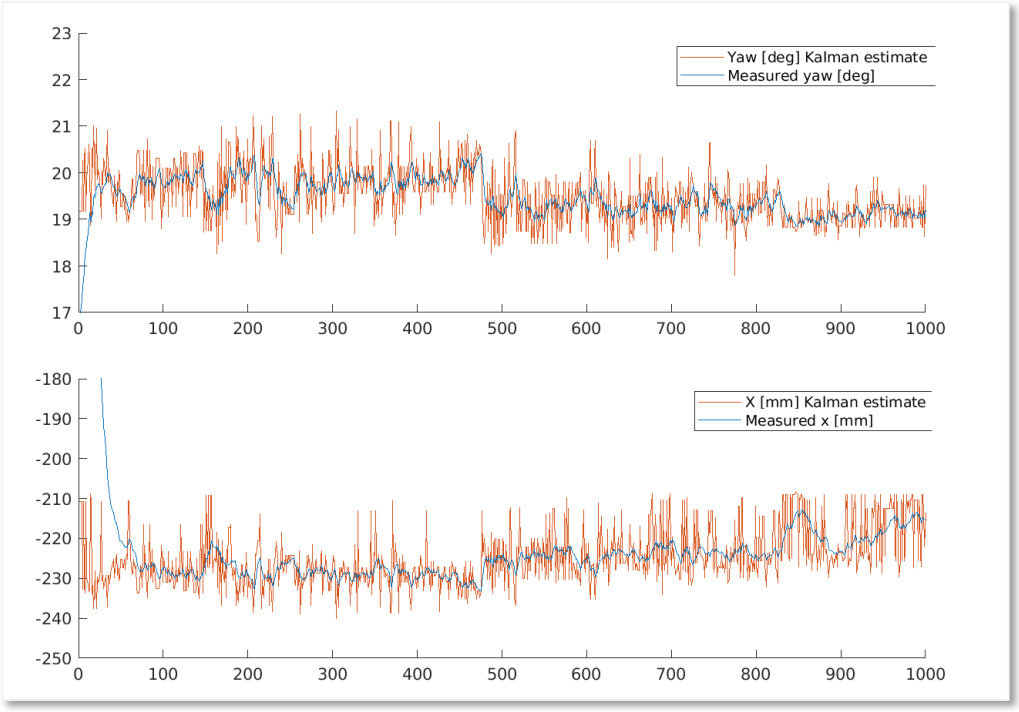


Figure 75: Example schematic of the horizontal single-ArUco localization state space model layout
Source: Self-made

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 [56] 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 neccesary 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. #EQUATIONS 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.

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

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

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 EQUATIONS.

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, 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 ~.

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

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

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

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 insignificant, 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, the estimated pose being accurate enough, with an average error of ±50mm in the range of ~1m of distance ArUco-camera. FIGURES show an schematic of the top view plus an overlook on the experimental setup used for validating the model.

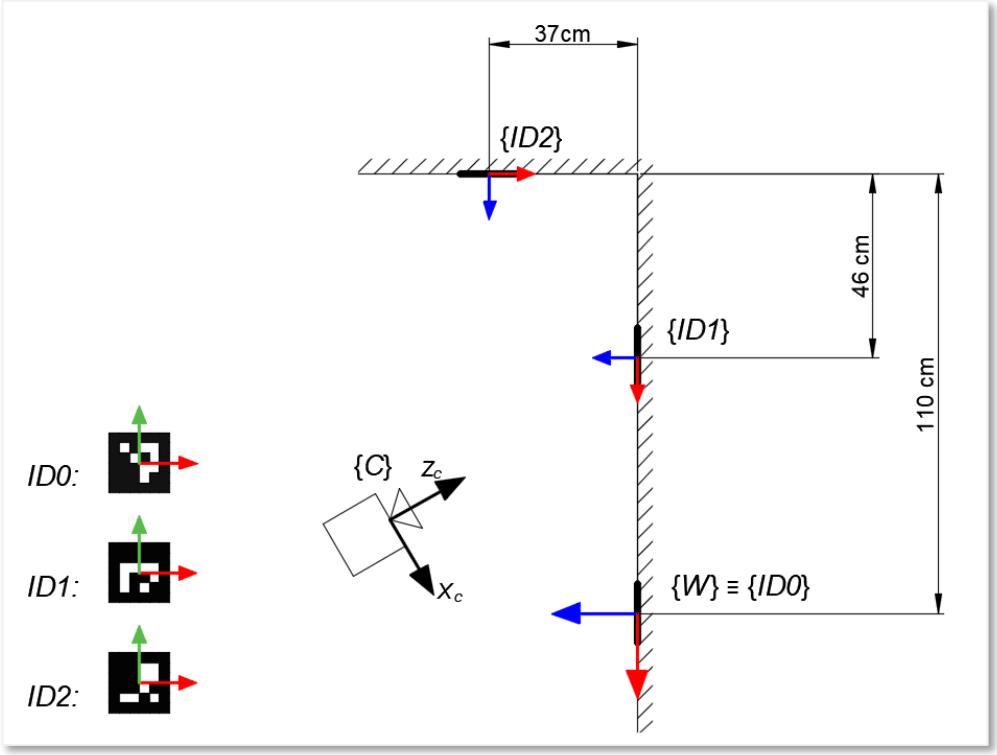


Figure 76: Schematic of the layout used to validate the multi-ArUco Kalman filter pose estimation
Source: Self-made

Figure 77: Experimental layout used to validate the multi-ArUco Kalman filter pose estimation
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:

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

5.4.4 Sensor fusion for blind pose estimation

The main pitfall of the previous model is that it can only work while at least one ArUco marker is within the range of the camera. 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 when no position information is obtained from the markers. This is the main idea of dead reckoning, which can be improved by the use of the ArUco markers as landmarks that will correct the filtered position estimate after some time working only with odometry, in which error will tend to accumulate over time.

There are multiple ways this can be implemented on the model described at SECTIION, depending on the sensor data that is available to the system. Android Smartphone devices come equipped with a variety of digital sensors, some examples of being:

- Accelerometer
- Linear accelerometer
- Gravity sensor
- Proximity sensor
- Gyroscope
- Magnetometer
- Rotation matrix

These can be streamed in real time while the camera feed is live, so that if the lag difference between the sensor and camera WiFi streamers is neglected, both video feed and sensory values can be obtained for each instant in time.

One straightforward manner of blindly tracking the camera is by using accelerometers and a gyroscope. Let's assume that the mining of these values happens outside of the state space model, much like we have been doing for the camera pose (x, z, θ), so that the system is conveniently linear and measures the accelerations $\ddot{x}$ and $\ddot{z}$ and horizontal gyro $\dot{\theta}$ along the axes of $\{W\}$. Then, we can define a state space model based around the following state and measurement vectors:

$$\boldsymbol{X} \sim \left(x, \dot{x}, \ddot{x},\ z, \dot{z}, \ddot{z},\ \theta, \dot{\theta}\right)$$
$$\boldsymbol{Y} \sim (x, \ddot{x},\ z, \ddot{z},\ \theta, \dot{\theta})_{measured}$$

From these, we define a preliminary 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 shown in the following EQUATION. In this model, the broadcasting steps that were performed towards the implementation of the Kalman filter can be keep, only applying minor modifications.

$$\begin{cases} \begin{pmatrix} x \\ \dot{x} \\ \ddot{x} \\ z \\ \dot{z} \\ \ddot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & Ts & \frac{1}{2}Ts^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & Ts & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & Ts & \frac{1}{2}Ts^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & Ts & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & Ts \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}_k * \begin{pmatrix} x \\ \dot{x} \\ \ddot{x} \\ z \\ \dot{z} \\ \ddot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{w}_k \\ \begin{pmatrix} \left\{ \begin{matrix} \ddot{x} \\ \ddot{z} \\ \dot{\theta} \end{matrix} \right\}_{IMU} \\ \left\{ \begin{matrix} x \\ z \\ \theta \end{matrix} \right\}_a \\ \vdots \\ \left\{ \begin{matrix} x \\ z \\ \theta \end{matrix} \right\}_b \end{pmatrix}_k = \begin{pmatrix} \left\{ \begin{matrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{matrix} \right\}_{IMU} \\ \left\{ \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{matrix} \right\}_a \\ \vdots \\ \left\{ \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{matrix} \right\}_b \end{pmatrix}_k * \begin{pmatrix} x \\ \dot{x} \\ \ddot{x} \\ z \\ \dot{z} \\ \ddot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{v}_k \end{cases}$$

A simpler way of laying this system would be to model the new measurements $(\ddot{x}, \ddot{z}, \dot{\theta})$ as inputs of the system in the input vector $\boldsymbol{U}$ instead of placing them as measurements in $\boldsymbol{Y}$. This way, we can preserve the vectors $\boldsymbol{X}, \boldsymbol{Y}$ as they were defined in the previous sections, which leads to smaller matrices $\boldsymbol{A}, \boldsymbol{C}, \boldsymbol{L}, \boldsymbol{Q}, \boldsymbol{R}$, so that the Kalman filter becomes more efficient computation-wise and the overall math is easier to follow, improving the readability of the implementation. In this newer model, we supress the propagation of $\dot{\theta}_{\in \boldsymbol{X}}$ to the next state and instead use the measured angular speed that we have modeled as input $\dot{\theta}_{\in \boldsymbol{U}}$ in the propagation equation.

Thus, we define the state-space model for the localization problem of a camera based on dead reckoning with ArUco markers as landmarks, for an instant in which a number of N ArUco markers are in sight with ids ranging from a to b as:

$$\begin{cases} \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & Ts & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & Ts & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & Ts \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}_k * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \begin{pmatrix} \frac{Ts^2}{2} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{Ts^2}{2} & 0 \\ 0 & Ts & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}_k * \begin{pmatrix} \ddot{x} \\ \ddot{z} \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{w}_k \\ \begin{pmatrix} \begin{Bmatrix} x \\ z \\ \theta \end{Bmatrix}_a \\ \vdots \\ \begin{Bmatrix} x \\ z \\ \theta \end{Bmatrix}_b \end{pmatrix}_k = \begin{pmatrix} \begin{Bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{Bmatrix}_a \\ \vdots \\ \begin{Bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{Bmatrix}_b \end{pmatrix}_k * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{v}_k \\ \boldsymbol{w}_k \sim N(0, \boldsymbol{Q}) \; ; \quad \boldsymbol{v}_k \sim N\left(0, \begin{pmatrix} \boldsymbol{R}_{3x3} & \vdots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \boldsymbol{R}_{3x3} \end{pmatrix}_{3Nx3N}\right) \end{cases}$$

The implementation and experimental validation of this model was left outside of the scope of this project, which is satisfied with theoretically laying down this model and progressing on the Item localization task with the model based only on visual localization based on ArUco markers.

5.4.5 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 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 this 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°.

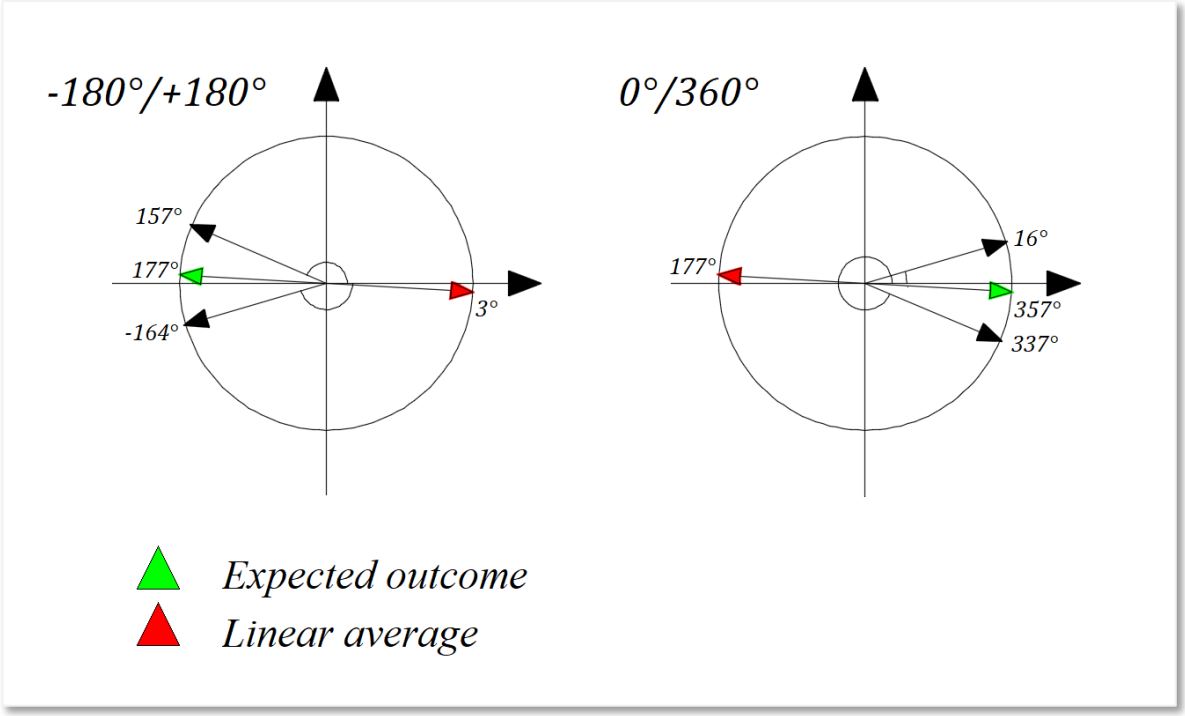


Figure 78: Issues with Kalman weighted average for different angle representations
Source: Self-made

In order to overcome this issue, the Kalman filter proposed at SECTION can be modified so that the angle is expressed as both its sinus and cosinus. While these are bounded magnitudes too, they can be safely operated with within a linear Kalman filter and avoid the described issue. The state-space representation of choice for this localization problem, of a camera based on dead reckoning with ArUco markers as landmarks, for an instant in which a number of N ArUco markers are in sight with ids ranging from a to b, is:

$$\begin{pmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}_{k} * \begin{pmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}_{k} + \boldsymbol{w}_k$$

$$\begin{pmatrix} \begin{Bmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{Bmatrix}_{a} \\ \vdots \\ \begin{Bmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{Bmatrix}_{b} \end{pmatrix}_{k} = \begin{pmatrix} \begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix}_{a} \\ \vdots \\ \begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix}_{b} \end{pmatrix}_{k} * \begin{pmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}_{k} + \boldsymbol{v}_k$$

$$\boldsymbol{w}_k \sim N(0, \boldsymbol{Q}_{4x4}) \; ; \quad \boldsymbol{v}_k \sim N\left(0, \begin{pmatrix} \boldsymbol{R}_{4x4} & \cdots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \boldsymbol{R}_{4x4} \end{pmatrix}_{4Nx4N}\right)$$

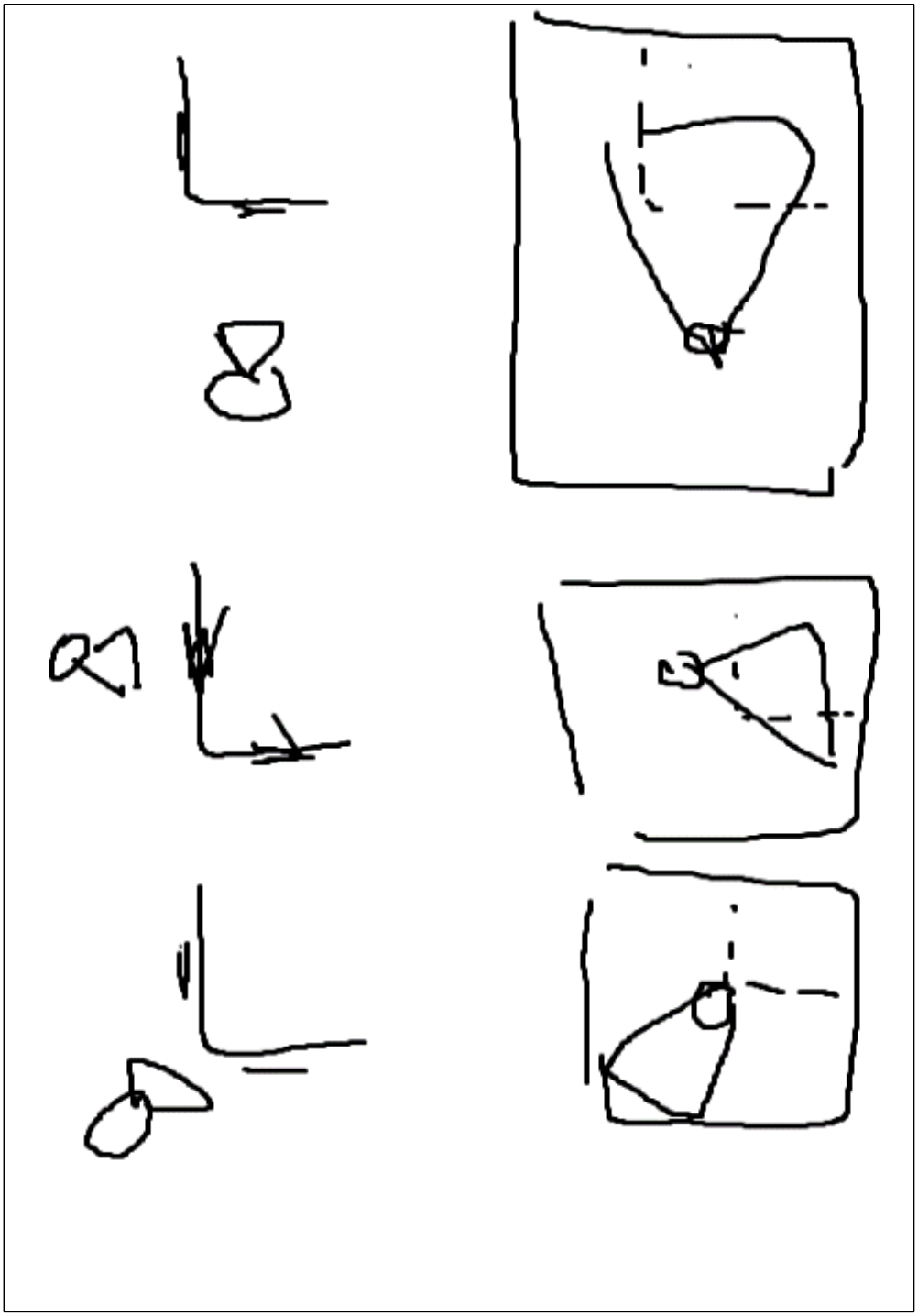


Figure 79: Issues with Kalman weighted average for different angle representations
Source: Self-made

An additional simplification has been adopted in this model, consisting on the removal of 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.

5.5 Item localization against camera coordinate frame

5.5.1 Estimating item position

By following the pinhole camera model described in SECTION, one can relate real-world points to image ones, provided an image is undistorted. These real-world points obtained are, indeed, referred to the camera. 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.

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 shows a depiction of this situation.

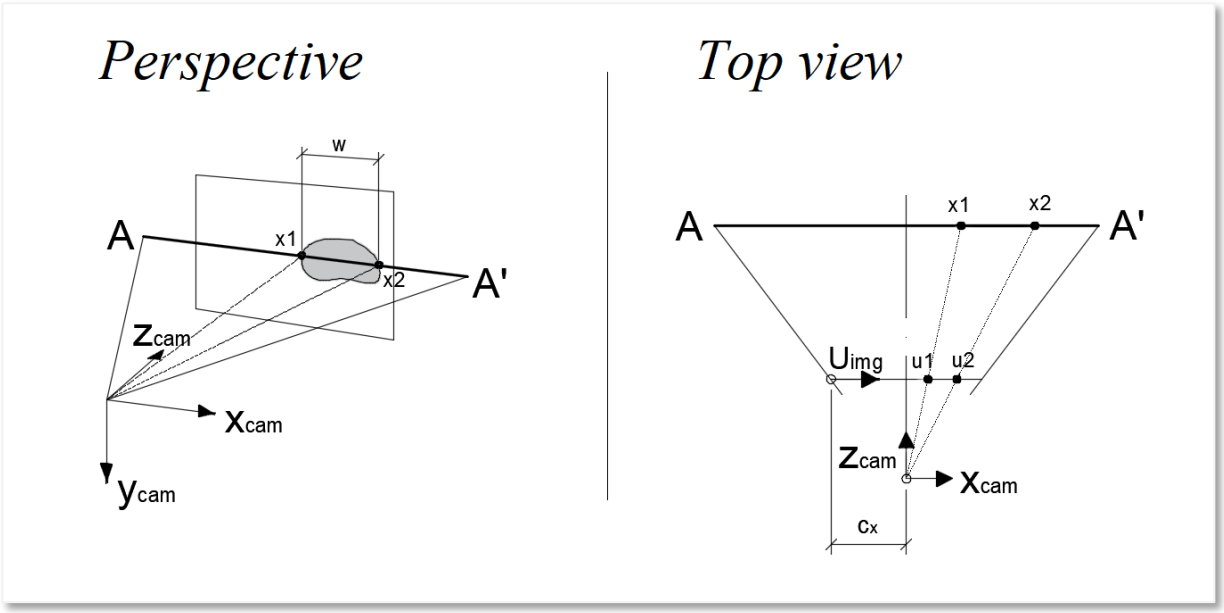


Figure 80: Laying out the problem of finding an item's position with camera known its width
Source: Self-made

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 $w = x_2 - x_1$:

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

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

From the top view in FIGURE, 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:

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

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

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:

$$z = \frac{s * u_1 - x_1 * f_x)}{c_x}$$

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

Arriving to the position of the item with respect to the camera. Optimally, this problem could 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.

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 very noisy measurement by default, plus some ArUcos can induce missplacements when seen from ambiguous perspectives.
- Darknet output inconsistency: 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 inconsistency: 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. Traditionally, probability density functions PDFs 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.

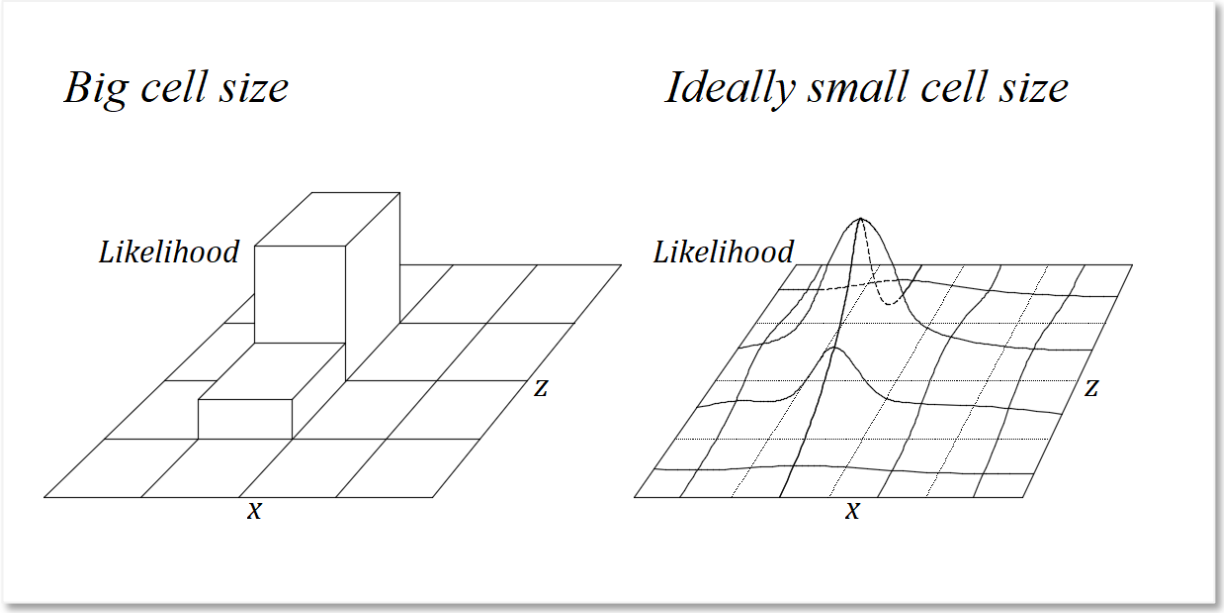


Figure 81: Intuitions behind the likelihood map. Discretization and cell size
Source: Self-made

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*:

- 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:

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

- $sat\,{}_{0}^{1}$: 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 sme 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.

IMAGE shows more graphical intuitions on the concept of likelihood maps.

- Pros
  - Dampens false positives/negatives
  - Dampens positioning accuracy in a smooth manner
  - Deals with not being able to track instances
  - Deals with moving objects
- Cons
  - It cannot perform instance counting*
  - The likelihood function does not follow a strict statistical definition*

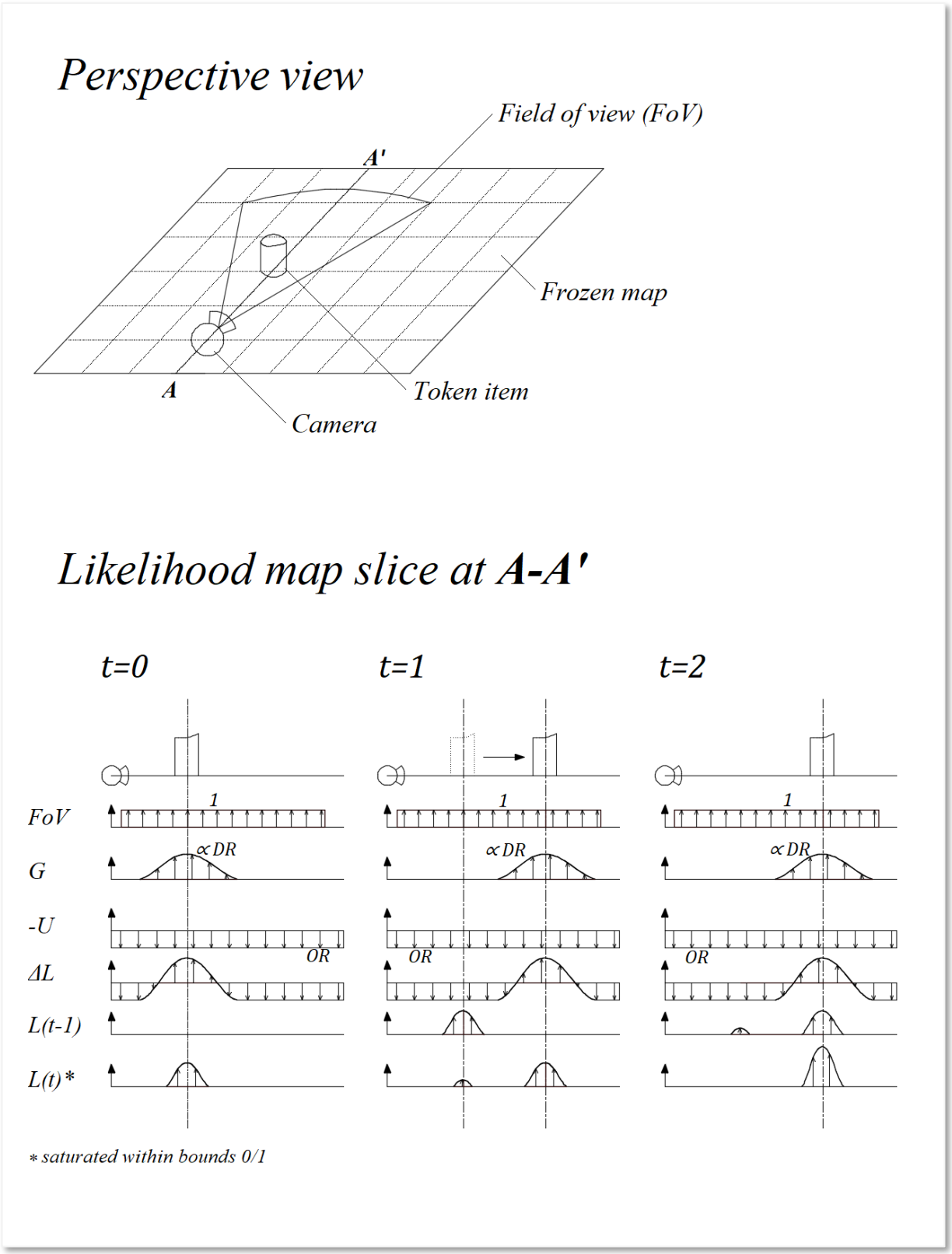


Figure 82: Intuitions behind the likelihood map. Appearance and disappearance
Source: Self-made

6 VISION-BASED ITEM LOCALIZATION PIPELINE

6.1 Introduction

6.1.1 Requirements and approach

What do we mean by item localization pipeline

Python

Image data

Darknet with yolov4-tiny

MS COCO

Custom Kalman filter

Image processing to generate maps

6.1.2 Computational issues

Memory leaks

Darknet python bindings

Ctypes

Achieving real time

Parallel computing

Python bindings

6.2 Pipeline overview

6.2.1 Concurrent approach

By joining the concepts in SECTION, we get

6.2.2 Parallelized approach

The model at XXX was clearly a CPU-driven process with blahblah

Low framerate + significant delays (3s)

6.2.3 Overall implementation aspects

6.3 Module-specific implementation aspects

6.3.1 Image capture

Undistortion

Happens on Mains

6.3.2 Camera pose estimation

Aruco dictionaries

Finding Aruco markers

Fake undistortion coefficients

Exceptions: no arucos shown, unexpected arUco, no arucos shown on start

What happens if localization can’t be made

6.3.3 Camera pose filtering

Importing a hardcoded Kalman filter model

Instancing the Kalman filter defined in SECTION

Exceptions: no arucos shown, angles non-linearity

6.3.4 Object detection

Darknet bindings

Starting the network

Looping detections

Issues when importing darknet without OS-level links: cd, sys.add path etc.

6.3.5 Item localization against camera

Using width and bounding boxes

Incomplete objects

6.3.6 Addressing coordinate complance

Show the multiple coordinate frames used through this work and how to match them

6.3.7 Mapper

Field of view

Instantaneous detection

World coordinate frame and map coordinate frame

Printing the pictures here

6.4 Demonstration

6.4.1 Set-up maps

6.4.2 Live tests on living room

6.4.3 Testing outcomes

Kalman filtering tuning

Angle issue discovered

ArUco inception when recording the screen

- Not implemented but can do
  - Walls
  - Occluded items

7 DISCUSSION

7.1 Object detector benchmarking

- Discuss overall approach
- AP and mAP. Recall. False positives vs false negatives which is better
  - Yolov3
  - What is more important
  - Data labelling
  - Mention YOLO models tend to work poorer on mAP
- Why weren't all yolo models subjected to anchor recomputation
  - Hard to justify additional 70h of training
  - Mention total training time
  - YOLOv4 tiny high Ap and 200fps hard to beat
- Why did tiny models perform better
- Why did efficientdets perform worse
- How does this top accuracy compare to other works
  - Carls. Our data is more difficult
- What would be the approach to take if a problem like this was to be solved quickly

7.2 Towards item localization

- Overall approach
- If item localization uses remote computations why use light object detectors
  - Not clogging RAM
  - Easy to train
  - Light to use (20MB)
- Why dont we pursue high precision
  - Proof of concept only
- Why not implementing full Kalman
  - Time, open end planning
- Top positioning accuracy (REVIEW MENTIONED REQUISITES)
- Mention integrated approach MONET

7.3 Conclusion

7.4 Goal response

7.5 Further work

Pseudolabelling

Readdressing labels manually
more data with more variability

Kalman filter tuning

Blind pose estimation implemented

8 REFERENCES

MY REPOSITORY MISSING MAYBE CHANGE USERNAME flagged [#ref[]]

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

[2] 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.

[3] 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.

[4] 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.

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] "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).

[15] "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).

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

[17] 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.

[18] 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.

[19] 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/.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] "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).

[32] 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.

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

[34] 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.

[35] 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.

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

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

[38] 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.

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

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

[41] 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.

[42] "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).

[43] 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.

[44] "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).

[45] "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).

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

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

[48] "EfficientDet - tutorial.ipynb - Colaboratory."

https://colab.research.google.com/github/google/automl/blob/master/efficientdet/tutorial.ipynb (accessed May 27, 2021).

[49] "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).

[50] "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).

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

[52] 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.

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

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

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

[56] 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.

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

[58] "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).

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

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

[61] 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.