Department of Information Engineering and Computer Science

Master's Degree in
Artificial Intelligence Systems

FINAL DISSERTATION

Object Detection at the Edge

Jointly optimizing a differentiable and non-differentiable system to track pedestrians on a microcontroller

Supervisor
Giovanni Iacca

Co-Supervisors
Elisabetta Farella
Elisa Ricci
Francesco Paissan

Student
Thomas Reolon

Academic year 21/22

Ringraziamenti

Ai miei genitori: prima o poi tutte le ansie e le preoccupazioni che vi ho gentilmente donato dovevano essere ripagate! Soldi non ne ho quindi spero che la mia tesi possa bastarvi. Ringrazio anche mia nonna che non vedeva l'ora che finissi, la mia sorellina, i miei amici che mi sopportano da anni, FBK per l'occasione di lavorare su questi argomenti, il prof. Iacca che ha avuto un mare di pazienza, l'università di Trento per la bella esperienza, i correlatori e ultimo ma non per ordine per importanza me stesso per essere rimasto forte e non aver mollato.

Contents

Abstract

In the early days of the computer era, these powerful machines were only available to large corporations and were called mainframes. Fast forward to today, we are carrying smartphones in our pockets, websites that used to be hosted in a central building have been distributed and replicated thanks to Content Delivery Networks (CDNs) and smart cities are starting to become a reality. Technology is rapidly evolving, however, when it comes to AI-based applications, the most common pattern is still to collect data at the edge and process it in central servers. This happens in a wide variety of contexts, from cameras in smart homes that are responsible to turn on the lights when a person is detected to smartphone applications that send information to a central server to process them and return the refined results to the final user. The main reason behind this behavior is that AI applications are often requiring a large amount of computational power, in fact, a discussion in the AI community affirms that deep learning has only lately become a popular machine learning approach because, twenty years ago, there was no hardware capable of running these models.

However we are starting to see the process of decentralization for AI applications as well, in fact, it has appeared for multiple years in the top 10 of technology trends by Gartner [69] proving that edge computing is going to become a central paradigm in our lives. This paradigm involves deploying computational resources closer to where data is generated, such as mobile devices, Internet of Things (IoT) devices, or local servers. This shift is being driven by the increasing demand for real-time processing, low latency, and efficient use of bandwidth. As a result, many AI applications, such as image recognition, natural language processing, and predictive analytics, are going to be run on the edge and the disruption that this paradigm could bring in both the way we interact with technology and the possibility to create new applications and services is enormous. This Thesis addresses the problem of how to run AI models at the edge with low computational power and energy requirements, in particular, our aim is to perform object detection of moving objects using only a motion map.

This Thesis was carried on in a joint effort with Fondazione Bruno Kessler (FBK) and its main objective is to assert whether it is possible to perform object detection from the motion map produced by a sensor developed "in-house". Object detection is a popular task in computer vision and can have many applications like traffic monitoring, which can be used to re-route cars when congestion is detected in a metropolitan area, pedestrian detection which could be used to identify the flow of the crowd for IoT applications or research purposes, security applications like surveillance. This technology could also be integrated into smart houses as an intelligent device that turns the lights on when someone enters a room (since the low requirements of the system make it runnable on cheap devices) or to collect statistics in remote areas where the energy required to run the AI system could be collected just by a single solar panel [67].

Object detection is a challenging task because it first requires identifying multiple objects in an image and then classifying them. Another important constraint is the resolution of the input image, in fact, detection on higher resolution images achieves better performances, but requires a higher computational cost; for our system, we are constrained to process 160x120 images because that is the size of the motion maps returned by our sensor [109]. Moreover, most of the existing literature is based on RGB image processing, while our input is just a binary map. Using binary motion maps over other sources of information like greyscale images is justified by the fact that the motion maps are sparse and contain mostly zeros, which allows a computational speedup each time we compute a multiplication with a zero entry, moreover, special models like XNOR-Nets [72] can be specifically applied to the processing of binary images.

Contributions:

- We introduce a novel dataset (Streets23) for multi-object detection and triggering.
- We create a framework using PyTorch [70] and python to simulate motion maps and train neural

networks. The simulator's code has been modified from Gottardi's work [109].

- We find a procedure to optimize a neural network's hyperparameters and architectural design with heuristics such as the Neural Tangent Kernel (NTK).
- We try different methods to optimize the simulator's parameters to optimize the detection performances, measured with the Mean Average Precision (MaP).
- We quantize the models with special libraries to see if they could be run on devices at the edge and how much the quantization degrades performances.

As important external contributions, other than the smart vision sensor [109] that generates the motion maps, we cite the lightweight backbone PhiNet [67], the YOLOv5 [45] codebase. Some parts of the code have been borrowed and modified from other papers, such as the work from Chen et al. [19], which implements the NTK score; minor citations such as this one will be cited later in the Thesis.

The Thesis is composed of three main chapters: literature review, methods and results.

The **literature review** introduces topics strictly related to this thesis such as Neural Networks (NN), motion maps, Object Detection (OD) and micro-controllers.

Methods firstly introduces which datasets are used to evaluate the performances for object detection (MOT17 [73], SynthMOT [28], Streets23); follows a description of how we customized YOLOv5 [45] integrating PhiNet [67] and the baseline results achieved from the model. Afterward, we describe the procedure we use to find efficient architectures and finally we explain how we can optimize the motion map simulator's parameters (including the failed trials).

Results is a chapter that aims at summarizing the results of the previously proposed methods. It discusses and visually compares possible network inputs (greyscale vs motion maps vs motion history images); motion maps (static vs policies) and models.

A brief summary of the results follows. The MaP averaged across the dataset is equal to 0.31 when **phiyolo**, which is the name of the convolutional neural network we create, is trained on greyscale images and 0.25 when trained on the motion maps (our baseline). After optimizing phiyolo's architecture we find two efficient configurations, which respectively have 1.3 times fewer parameters (**77K**), but achieve the same performances; and 15 times fewer parameters (**7K**), achieving a MaP of 0.20, two times higher wrt. handcrafted architectures of the same size. The best quantization seems to be 8-bits, which brings an average loss of MaP of 0.02. 6-bits and lower produce drastically reduced performances, while other approaches like binary convolutions (eg. XNor-Nets [72]) seem not to work with every type of architecture.

The final MaP obtained by the quantized 77K architecture using the best policy is equal to 0.33, which is comparable with the baseline model trained on the greyscale, thus proving that motion maps are an efficient method to perform object detection. Moreover, this architecture requires 20 times less RAM wrt. the baseline model (400KB against 8MB) and requires 2 times fewer computations (30M Multiply-ACCumulate (MACC) operations versus 58M MACC of the baseline), making it a good choice for running AI applications on edge devices. Unfortunately, 30M MACC is still quite computationally expensive, for this reason, it is advised to adopt the 7K architecture on less powerful devices, which achieves a MaP of 0.22 with only 2.8M MACC.

Since we don't have a real case scenario and predefined hardware, it is hard to work on computational constraints, thus, specific fine-tuning is required when applying these methods in a business case. The self-imposed constraints we adopt are taken from previous work at FBK [67].
Source code available at: https://github.com/thomasreolon/svs_detection

1 Literature Review

This first chapter introduces some of the most important concepts that establish the foundation for the later work. The system we describe in this Thesis is an effort towards edge computing object detection with neural networks thus before explaining our system works we need to summarize some knowledge about the most important bricks that build our system like motion maps, object detection approaches and neural networks, with this latter topic being introduced in the following section.

1.1 An Introduction to Neural Networks

One of the most important parts of our system is the neural network that performs object detection, counting and triggering, for this reason, this section will introduce neural networks and how they can learn to recognize patterns.

Stochastic gradient descent. One of the main objectives of machine learning is to optimize an objective function, these objective functions can have multiple meanings, but they are usually bounded to a task and by optimizing this function we usually create a model that is capable of accomplishing that task. There are many optimization algorithms, for example, random search tries random solutions until it finds something that works, and local search algorithms try to slightly modify the current solution to search for a new set of parameters that improves over the old set (there are many local search algorithms: hill climbing only accepts solutions that improve the objective function, simulated annealing [21] can accept worsening solutions with a probability that decreases with the iterations, genetic algorithms are inspired from biology and modify the solutions using crossovers and mutations). There are many optimization algorithms and, since there is no perfect optimization algorithm [95], we have to wisely choose which one to use, based on our problem. When the objective function is differentiable we can use a special class of optimization algorithms: gradient descent optimization algorithms. This type of optimization is based on the fact that our objective function is a function of inputs and parameters and, since it is differentiable, we can see how changing each parameter will affect the output of the network.

Stochastic gradient descent is an iterative optimization algorithm that, at each step, slightly modifies the model's parameters to minimize a loss function as shown in Figure 1.1. We usually update the parameters only by a small quantity called learning rate, because for models with many parameters changing all the parameters at the same time changes too much the output function, thus requiring us to limit the change to a fraction of the gradient.

A known problem of the steepest descent is that it only finds the local best solution, in fact, as we can see from Figure 1.1, the optimization stops at a local optimum, which is not the best possible solution. Luckily for neural networks with thousands of parameters, it is really difficult for an algorithm to get stuck since if at least one of the parameters is not in a local minimum, we will always have a step that could allow freeing other parameters from their local minimum. To summarize: we can optimize neural networks with SGD because they are differentiable parameterized functions, during the optimization process we learn some parameters that help minimize a loss function through the formula:

$$w := w - \eta \nabla L(w)$$

where η is the learning rate and $\nabla Q(w)$ is the gradient of the loss function with respect to the weight.

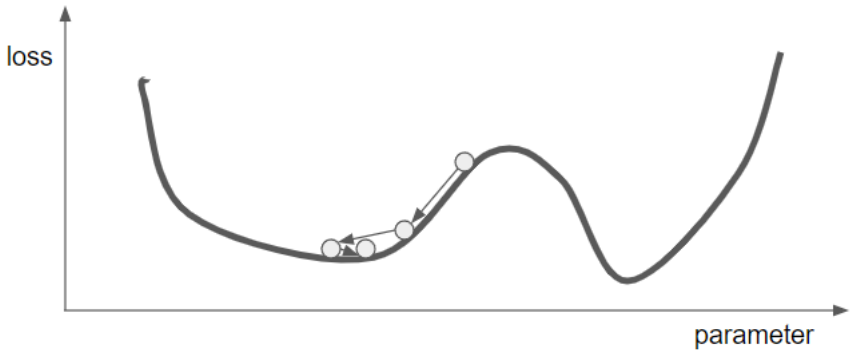

Figure 1.1: Steepest descent algorithm to minimize the loss function.

Chain Rule. The computation of the gradient for each parameter would be computationally expensive, fortunately, we can decompose formulas to compute the gradient in partial results and store those results to speed up the whole computation. This is possible because the gradient can be seen as:

$$\frac{dloss}{dw} = \frac{dloss}{dy} \cdot \frac{dy}{dw},$$

in this way, we can build a graph whose root is the derivative of the loss function and build this graph going backward in the model's computation's operation. If the model is sequential, the graph is just a tree, while, if we have skip connections or re-use parameters multiple times the graph acquires a more complicated shape (although it always remains a directed acyclic graph).

Optimizers. Stochastic gradient descent is one of the simplest optimization algorithms it simply updates the gradient of the parameters with respect to the loss computed on a single sample. To have more stability, batch gradient descent was introduced, the main difference is that the step, instead of being computed after every sample, is computed only after having calculated the average of the gradient over all the samples in the data set, this has the advantage of producing gradients that point toward the minimum of the function instead of jumping around, but, it is way slower and it is more probable to get stuck in a local minimum. for this reason mini-batch was invented; it is a trade-off between stochastic (1 sample) and batch (all the samples), that computes the gradient over a small subset of the data allowing us to have more regular gradients and don't worsen the speed of the algorithm too much since the samples in the mini-batch are usually parallel processed in the GPU.

New strategies have been invented for improving convergence rate, for example, momentum keeps in memory the old gradient (usually updated using exponential average) and updates the weights using both the new gradient computed from the loss and the old one (thus building a sort of momentum). Another common trick used in optimizers is weight decay which adds an invisible L2 regularization loss that aims not to have parameters with huge values, moreover, it helps with generalization and avoiding over-fitting. An evolution of the momentum is the NAG algorithm [66], which firstly looks ahead by stepping using the old gradient and then computes the updated gradient with the updated parameters.

Schedulers are used to adjust the learning rate through time, usually decreasing it at each step. Some common implementations of the scheduler are: milestone based, which decreases the learning rate at the epochs decided by the user; exponential, where $\nabla_{t+1} = c * \nabla_t$, with c¡1; cosine, which decreases gradually (we use it in the implementation); cosine annealing with warm restart [53].

Even though we can optimize the learning rate through time it would be appreciable to adjust the learning rate for every individual parameter. A popular optimization algorithm is Adam [46], which improves over RMSProp by keeping the exponential average of the past gradients (thus making it similar to momentum), but with the automatic re-scaling of the learning rate thanks to dividing it by the exponential running average of the standard deviation of the gradient. In our implementation,

AdamW is used [54], it slightly changes how weight decay is implemented.

Other strategies that influence training a neural network are to be taken into account too: shuffle the training data to avoid biases [79] or sort the training samples by difficulty (curriculum learning); batch normalization which helps the parameters not to diverge allowing us to use bigger learning rates; early stopping to avoid over-fitting by stopping the training when the validation loss stops decreasing; adding noise to the gradient to promote exploration in deep networks [65] (noise amount decreases exponentially with time).

Frameworks. Since neural network implementation isn't a trivial task, many collectives of people have developed frameworks to facilitate the training of deep learning-based models. Tensorflow, created by google, [4] is one of the most well-known libraries for machine learning, but its usage saw a huge decrease during the last years, especially in the research field. It still remains a popular solution at production time since its capability of supporting multiple types of platforms and environment (GPUs, TPUs, embedded, javascript).

In recent years JAX, another framework by google, [14] became a really popular choice for deep learning applications especially due to its speed and performance.

Although the two cited frameworks (and many others like café, MatLab, [...]) are good choices, we implemented our neural networks using Pytorch [70], which is probably the most used deep learning library as of 2023. The choice has been primarily influenced by the fact that it's the simplest to use for the writer.

Architectures. The most famous architecture for neural networks is called feed-forward neural network (FFNN) or multi-layer perception (MLP), which is powerful enough to approximate any function given enough parameters and depth (of 2) [38]. the FFNNs are built by alternating matrix multiplications with an activation function, the weights of the matrix allow the network to apply transformations to the data, while the activations are used to introduce non-linearities which are necessary to learn complex patterns. The typical activation function used to be the sigmoid function $1/(1+e^{-x})$ but nowadays rectified linear unit has become the goto solution. Variations to the ReLU like SiLU, LeakyReLU, Swish, [...] are similar to the original ReLU, but they often have a gradient different from zero for x ¡ 0, this helps optimization convergence.

Relative to image processing, convolutional neural networks (CNNs) have achieved state-of-the-art performances in almost every computer vision task since 2012. Their power lies in the fact that FFNNs need to learn different weights for each output location while in CNNs the same weights are applied to the whole image by shifting the kernel. Krizhevsky et al. [47] showed that the first layers of the CNNs can learn kernels capable of identifying corners and edges, while the later layers evolve these features so that they can identify more abstract concepts like faces. A big breakthrough was achieved in 2015 with the invention of skip-connections [37], which allow creating deeper architectures (gradient backpropagated better, each block that uses a residual connection can learn a different type of feature (this is key for transformers' success)).

1.2 An Introduction to Micro-controllers

Microcontrollers are small computers that are built to run and execute all their functions on a single integrated circuit. They are widely adopted in many areas like industrial automation, security automotive systems and many more. Microcontrollers' instructions are often written with low-level languages like assembly or C to perform single tasks, but with the advancement of edge computing this mini-computer are being put in charge of solving always more tasks to reduce the latency that would be unavoidable by processing the information in a central server. Microcontrollers have become fundamental in the Internet of Things since they allow other devices to communicate with each other, moreover thanks to their flexibility and lower costs they have seen a wider adoption also between students (for example with Arduino).

This brief paragraph will talk about micro-controllers optimization. Yue et al. [98] improve on a standard Intel 8051 micro-controller by (1) improving the ALU that computes multiplications and division making these operations 6 times faster; (2) overall improving the ALU by using only

combinatorial operations that can be computed in a single clock; (3) they use multiple clocks and a hardwired control unit. The paper finishes showing the improvements in speed and the benchmarks results. Aerabi et al. [5] make a comparative study about different micro-controllers for IoT systems and cryptography algorithms to secure the privacy of the transmission to benchmark which is the best trade-off between security and power needed to compute cipher the communications securely. The topic of saving energy in micro-controllers is becoming more and more important, in fact, many fields require MCU to be as low energy consumption as possible, this is simple to understand for medical fields [11] but it is important for many other sectors as well (eg. wearable devices [81]).

Micro-controllers. This paragraph will list the specifics we should know when using a microcontroller [86].

- input voltage: expressed in Volt, it is the necessary pressure needed to run the MCU (Arduino 7-12)
- word lengths: the number of bits (usually 32)
- clock speed: speed in cycles per second (Arduino is 16Mhz, ARM-M3 100Mhz)
- SRAM: static ram in KBs (2 arduino, 64 ARM-M3)
- flash memory: permanent memory (512KB ARM-M3)

Smart vision sensors. For our specific scenario, our microcontroller is considered to be attached to a smart vision sensor. Smart vision sensors are small cameras that can perform computer vision tasks using low computational power in real time. They are usually used in quality control, automation and surveillance with the latter being also one of the main tasks of the thesis. There could be many applications for Smart Vision Sensors (SVSs), for example, Fernandez et al. [30] develop a sensor capable of enhancing privacy by automatically blurring faces in the captured images.

For our thesis, the smart vision sensor designed by Gottardi et al. [109] produces QVGA grey-scale images and 120x160 binary motion bitmaps. Its voltage is 3.3 V and it's power consumption is 1.6mW when running at 15 fps. For the aim of our thesis, we will focus our attention on the task of object detection using the motion map, thus the next section will present some works relative to motion map processing.

1.3 An Introduction to Motion Maps

Motion detection algorithms are often used to analyze smart video sensors' captured frames. In this section we firstly introduce some well-known motion detection algorithms, we then proceed to explain Gottardi's smart background suppression algorithm and we finally review other works that use motion maps in smart sensors for real-time applications.

Motion maps algorithms. Motion maps have been widely used in the past to classify actions or detect moving objects, for this reason, there is a wide literature about techniques to compute them. The most popular technique is frame difference, it is widely adopted because it's effective, very sensitive to changes and very efficient to compute.

The drawback of this method is being too susceptible to changes and thus containing a lot of noise. A more robust approach is background subtraction, the most basic implementation of the algorithm uses static background images which has the problem of not being adapted to light changes and thus, making this method weak in open environments.

A simple solution is making the background adaptive to the changes in the environment, for example with a running average where $B_t = \alpha * B_{t-1} + (1 - \alpha) * I_t$, where B is the background and I is the new frame (or the slightly more advanced technique that uses a different α for pixels in the background and in the foreground, which are computed in respect of a fixed threshold).

Other variants of background subtraction use the color similarity of pixels to remove shadows from the motion map [56], explore how to better initialize the background value [57], try to identify

objects in harder environments (eg. a flow of water) by analyzing how many pixels are classified as foreground in different frames (which they call "estimating the spatial variance along the temporal frame") [49], use a mixture of background subtraction and frame differencing [41] and improve the updating strategy for the background [87].

Other methods used to compute motion maps are optical flow [93, 82], where the movement of objects is built from the apparent motion of the pixels in consecutive frames building a vector field, and approaches using neural networks [99, 7] that often use a convolutional neural network to extract high-quality backgrounds.

Finally, one of the most famous motion detection algorithms is the gaussian mixture models, in which, one or many probability distributions are built for each pixel; when we detect a pixel value that's very unlikely for the current distribution, we consider it motion and slightly change the probability distribution to make that value of pixel more probable. The advantage of this approach over other approaches is that it can handle pixels that have more than one value for the background (eg. the leaves of a tree moved by the wind). This advantage is also obtained with a background subtraction algorithm (as shown in Figure 1.2) from Gottardi et al. [109], which is used in our system and is better explained in the next paragraph.

Vision sensor with dynamic background rejection (from FBK). The QVGA vision sensor introduced by Gottardi et al. [109] produces 320x240 greyscale images and 160x120 binary motion maps. The chip consumes 1.6mW when operating at 15fps with a power voltage of 3.3V/1.2V. The algorithm producing the motion maps works by maintaining two background thresholds, when a pixel is detected outside of the thresholds (plus a tolerance hyper-parameter called delta_hot) we consider it motion. After this first detection, erosion (with a programmable kernel) is used to remove the noise.

We report here the simulator's algorithm (with some modifications from the original one):

```
def next_frame(self, frame):
    Threshold_H, Threshold_L = self.Threshold_H, self.Threshold_L
    open, close, dhot = self.open, self.close, self.dhot

    # activations "pixel became a lot brighter"
    tmpH = (frame - Threshold_H) > 0  # pixel open
    tmpH_hot = (frame - Threshold_H) > dhot  # pixel hot
    Threshold_H[tmpH] = Threshold_H[tmpH] + open  # update open
    Threshold_H[~tmpH] = Threshold_H[~tmpH] - close  # update close

    # activations "pixel became a lot darker"
    tmpL = (Threshold_L - frame) > 0  # pixel open
    tmpL_hot = (Threshold_L - frame) > dhot  # pixel hot
    Threshold_L[tmpL] = Threshold_L[tmpL] - open  # update open
    Threshold_L[~tmpL] = Threshold_L[~tmpL] + close  # update close

    # triggered pixels
    tmp_hot = tmpH_hot | tmpL_hot
    heat_map = np.zeros_like(frame)
    heat_map[tmp_hot] = 255
    heat_map = cv2.erode(heat_map, self.erosion_kernel, iterations=1)
    return heat_map.astype(np.uint8)
```

There are other background subtraction algorithms that aim at improving computational speed, but they come with drawbacks, for example, Yuriy Kurylyak et al. [50] try to improve the computation speed of the motion map using another adaptive background subtraction technique, this proves that efficiency is important in the community of computer vision, moreover, since their algorithm is implemented in Visual C++, while the one used in our project runs in-hardware. Ferreira et al. [90],

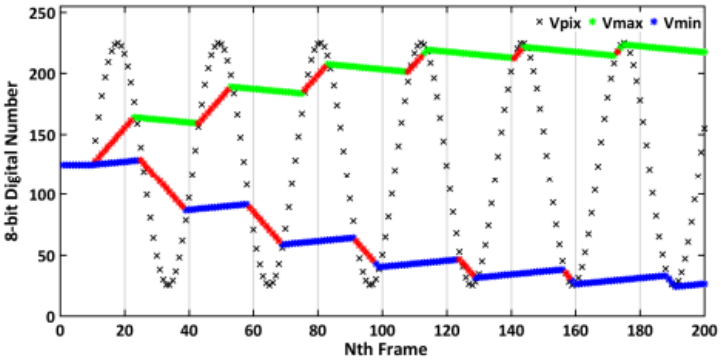

Figure 1.2: The use of two thresholds allows incorporating into the background pixels with a varying periodic value [109].

although, implement an in-hardware background subtraction algorithm but their background image is static, thus making it weak in outdoor scenarios.

After this brief review of motion detection algorithms, we see how they have been used in the literature.

Use of motion maps in the literature. Before the deep learning era motion maps were widely used in computer vision, for example, Marc Micatka [60] explains how to train a human action classifier, we notice that this classifier uses a motion history image (MHI), an image that includes the information of multiple timestamps and is iteratively computed as follows:

$$MHI = \begin{cases} t & if B_t(x,y) = 1 \\ max(MHI(x,y,t-1) - 1, 0) & if B_t(x,y) = 0 \end{cases}$$

where B_t is the binary motion map at the t^{th} frame. The result is a grey-scale image with whiter pixels where the movement is recent.

Once we have our feature image that contains the motion of the action we need to make the features scale, shift and rotation invariant. We can use the image's central moments to understand where the center is (solving shift), how many pixels are white (to re-scale) and how much to rotate the image [32]. Finally, we can try to learn the pattern of the MHI images with models like nearest neighbour, SMV and random forest (random forest seems to be the best classifier for [60]).

Zhu et al. [106, 107] justify the use of motion maps in smart vision sensors for two reasons: it's hard to process high-resolution images in real-time and RGB images are a complicated problem to solve because we need to learn how to exclude the background from our computations. They propose a two-stage object detector that generates region proposals from a coarse motion map and then proceeds to use a convolutional neural network to generate bounding boxes with a fine-grained resolution from the image frames. Another reason for using motion maps is that they seem to be a robust method for detecting fast-moving objects [78].

Mondal et al. process event-based data using spectral clustering [64]. Their approach is based on neuromorphic vision sensors but can be extended to motion maps, it works by finding objects by clustering their movements and building a graph that spans multiple frames. The clusters are found using spectral clustering.

1.4 An Introduction to Object Detection

The previous sections introduced the tools we use to build our framework, now we pay some attention to the tasks we are trying to solve, starting from object detection (OD).

The task. While in image classification is sufficient to assign a single label to the whole image, in object detection we want to assign the label to a specific area of the image called the bounding box (an

even further evolution is image segmentation, where we assign a label to each pixel). The advantage of object detection over image classification is that it allows the identification of more than one object per image and can be applied in many useful fields like self-driving cars, face recognition and robotics.

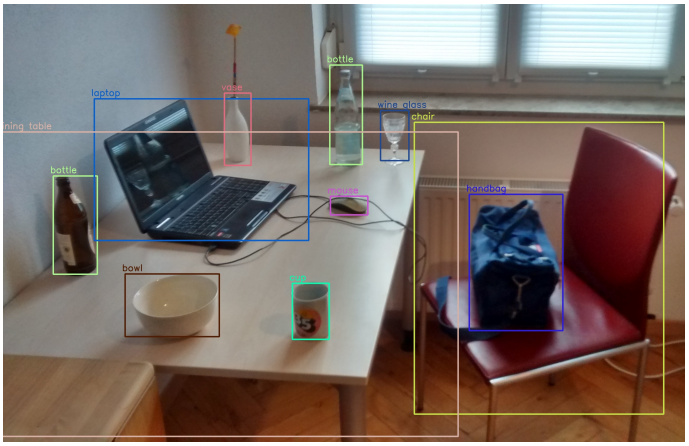

Figure 1.3: Example of object detection. We identify the class of objects (eg. laptop) and their location in the image, which is referred to as bounding box [2].

There are many datasets to train neural network models for this task, for example, COCO [52] and Pascal VOC [26] are two of the most used datasets for this task in the literature. COCO contains 80 different classes including people and cars making it a good starting point to train the network for detecting a wide range of objects.

Other approaches. Previous to the deep learning era features used for object detection used to be handcrafted; one of the most famous approaches was the Viola-Jones algorithm which used some pre-learned 2d kernel to extract features and could compute the convolution very quickly thanks to the use of the integral image, which allowed the computation of the sum of a rectangular area with just 4 operations, as shown in Figure 1.4.

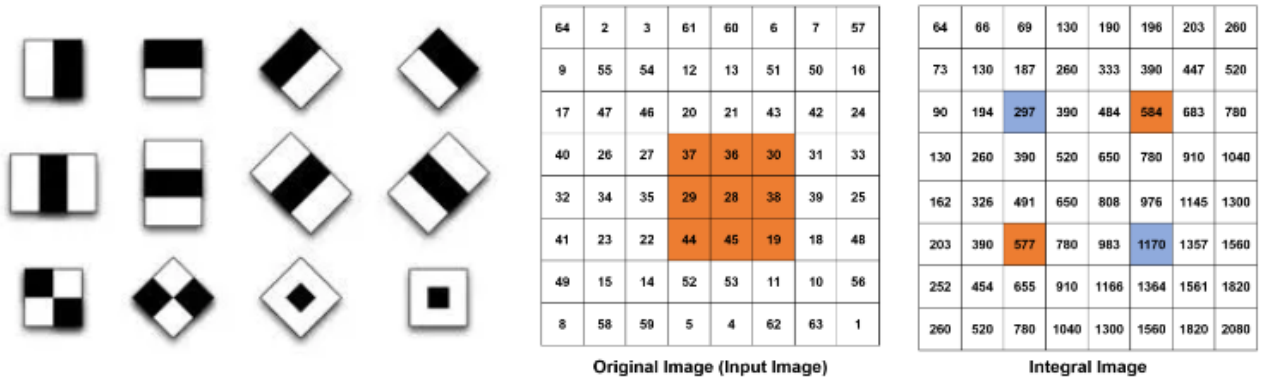

Figure 1.4: HAAR Filters (left). Example use of the integral image (right): instead of performing nine sums, we can perform two sums and two subtractions, thus having a constant computational cost instead of a linear one. [3]

After 2012, features started to be learned through neural networks, reducing always more the need to handcraft solutions. At first, CNNs were only used as feature extractors, for example, R-CNN [35], a 2-stage OD algorithm, firstly used a region proposal algorithm (eg. selective search) to find possible objects and then used a classifier based on CNN's features to classify what that region was. An improvement in speed was achieved by the same authors [34] when they used the image features instead of the whole image for the image proposal part of the algorithm. Finally, Ren et al. [76] achieved real-time OD by using another neural network (the so-called Region Proposal Network) for the region proposal part of the algorithm, instead of relying on the slower selective search.

Even though two-stage models achieve the highest performances, one-stage models are widely studied due to their higher speed. YOLO will be introduced in the next section, while we will spend

some lines introducing vision transformer-based approaches to OD. Transformers achieved state-of-the-art performance in many NLP tasks and were proved to be universal computers [33], thus lately they are being used in introduced in computer vision tasks too. An advantage of transformers over convolutions is that transformers can look at the whole image while convolution can only use the information in that area; other than this being the advantage of transformers, this is also a big drawback: since each pixel needs to compute its similarity with each other pixels ($A = \frac{W_q \cdot F \cdot W_k \cdot F}{\sqrt{d}}$ [91], where W are learned weights and I are the image features ($W_q \cdot F$ are the queries, which are a linear projection of the image features)). The similarity matrix, called attention matrix, is quadratic in the number of pixels thus making this approach unfeasible for high-resolution images. A first solution was proposed by Dosovitskiy et al. [24] where they reduced the size of the image by concatenating all the pixels in a patch along the channel dimension. The so-called architecture called vision transformer was then used for object detection [17] and novel approaches like [108] tried to avoid the use of patches and solved the quadratic space problem by limiting the number of pixels that the attention can look at.

Experimental approaches try to learn OD in an unsupervised manner with adversarial learning on videos: one network performs object segmentation while the other network tries to predict the optical flow from the previous frame and the current frame, masked outside of the predicted segmentation area. If the bounding-box/segmentation network extracts a bad prediction the flow network will have an easier job of understanding how the object moved since the background will be visible, while, if the segmentation network extracts a good segmented area it will be harder for the optical flow network to reconstruct it [97]. This approach can be applied for moving object detection but is not using motion maps, so it is out of our aim. The next paragraph will review some papers that specifically perform OD using motion maps.

Object detection with motion-maps. Object detection with motion maps can be summarized in the following steps:

- acquire a sequence of images from a static camera
- estimate the motion using one of the previously discussed techniques (frame difference, optical flow, ...)
- create the binary motion map (usually by thresholding the motion estimate)
- detecting objects (usually by clustering the moving pixels)

More advanced techniques have been introduced more recently, for example, Chen et al. [18] prove the utility of motion maps for understanding crowd behavior and performing object detection of the people with segmentation. Svitov et al. [89] learn the background and use it to enhance mean average precision (MaP) for moving object detection using a CNN (similar to YOLO, but using motion map too). Finally, Batra et al. [10] use an Arithmetic Distribution Neural Network [101] to perform background subtraction and later use an anomaly detection network for triggering.

YOLO. You Only Look Once (YOLO) is a one-stage object detector based on a convolutional neural network introduced in 2016 by Redmon et al. [73]. It predicted bounding boxes and classes with a single forward pass, making it significantly faster than other methods and allowing it to achieve real-time performance without the use of region proposals.

YOLOv2 [74] was proposed the following year introducing batch normalization, the use of anchor boxes and aspect ratio (which embed some prior about the object we are trying to detect, thus making its detection simpler); another change was that instead of predicting the upper-left corner of the bounding-boxes, the center pixel was now predicted.

YOLOv3 [75] was introduced in 2018 and it's the last paper by the original author since he thought his works were going to be used for bad purposes. It added a feature pyramid network to increase the detection accuracy for small objects through the use of multi-scale features.

YOLOv4 [12] changed backbone switching to CSPDarknet53 and improved the training by adopting the focal loss (introduced from [51], which is an OD model really similar to YOLO) to solve the problem of unbalanced classes and improved data augmentation techniques.

2020 saw the introduction of YOLOv5 [45] as an independent version of Ultralytics, it introduces mosaic data augmentation, which simply consists in concatenating 4 images to form a grid and improved the backbone.

YOLOX [31], introduced in 2021, changes the detection head to use a multi-scale prediction strategy to increase accuracy.

Finally, YOLOv7 [92] improves the training procedure by adding some guiding losses to improve the accuracy and convergence time of the parameters.

Other subsidiaries implementations of YOLO have been created in the years [44], in particular, some of them have been developed especially for being adopted in IoT devices [83, 40], but their main innovation is to use a more lightweight backbone. Roszyk et al. [77] adopt the YOLOv4 architecture to detect people for autonomous driving, using both RGB and thermal images.

Figure 1.5: Ground truths for object detection on the MOT17 dataset [22], MOTSynth [28] dataset and Streets23 dataset.

1.5 An Introduction to Triggering and Counting

Object detection is a challenging task, so we want to see how simpler tasks like triggering or counting (counting isn't simple, but we are) can be solved by a simpler network.

Triggering For this task, we are not interested in identifying the objects' position in the image, but we only want to know if something is happening inside of the image, for example knowing if there is at least one person inside of the camera view. An example is shown in Figure 1.6. This task could be used to acquire more high-quality data once the triggering event has occurred, allowing to run a low energy consumption algorithm for most of the time and use more energy-consuming algorithms only when an event is triggered.

A simple triggering system based on motion maps could work by comparing the number of pixels of the motion map that detect movement to a threshold, if the number of pixels surpasses this threshold we conclude that something is happening and execute a corresponding action. More advanced systems like the one used by [68] use blob detection to extract interest regions and further process each blob extracting additional information such as the number of movement pixels and the center of mass of the blob (image's moments) to later classify this blob using a support vector machine (SVM).

Counting Counting in computer vision involves algorithms and image processing techniques to automatically count objects in an image or a video. There are many possible techniques for this task and the final choice must be taken considering the complexity of the scene, the appearance of the objects to count and the performance requirements of the system. Usually counting is performed by first performing object detection and then counting the number of objects found, but this task could

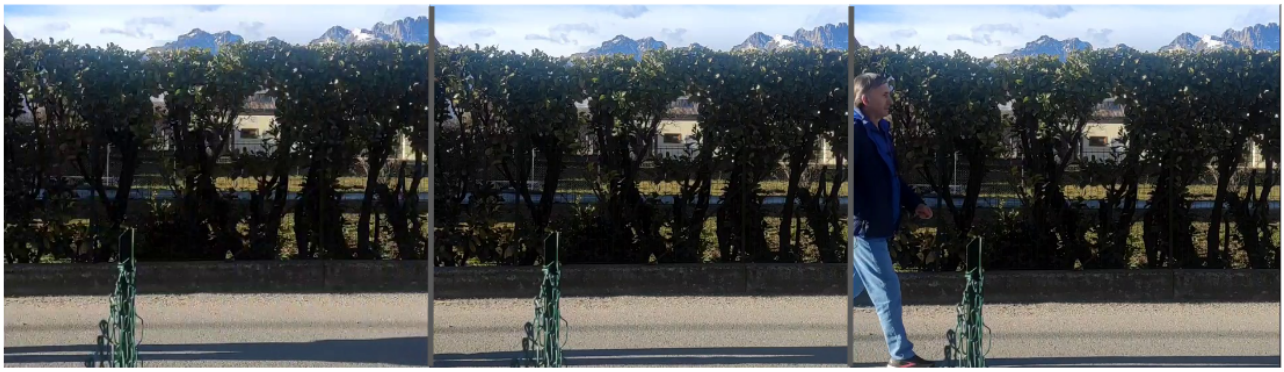

Figure 1.6: Given a stream of frames (eg. a video) we want to understand when a certain event occurs. In this case, the triggering happens when at least one person enters the scene.

also be performed in an end-to-end way, by training the regression head of a neural network to count the number of objects [85]. Low energy approaches to object detection can include blob detection (which is simple to perform from a motion map), HAAR cascades (like Viola-Jones that was discussed in the previous section and optical flow, which cannot be performed on binary images).

Figure 1.7: Illustration of the counting task. We want to predict how many objects are currently in the scene. Frames from SynthMOT [28].

Counting in computer vision can be adopted for a wide range of applications such as monitoring crowds, analyzing the flow of traffic and performing quality control. Having accurate estimations of the count of objects can give useful insights for decision-making choices (eg. how to re-route highly trafficked streets in metropolitan areas).

2 Method

This chapter describes the adopted method and the choices adopted to implement this framework. The chapter is divided into four main sections: choice of the datasets and data preprocessing, implementing the baseline framework that is built upon a static simulator plus YOLO [73], how we optimize YOLO's architecture and adapt it to our problem and how we can learn a policy to update the simulator's parameters. The overall architecture can be summarized by Figure 2.1.

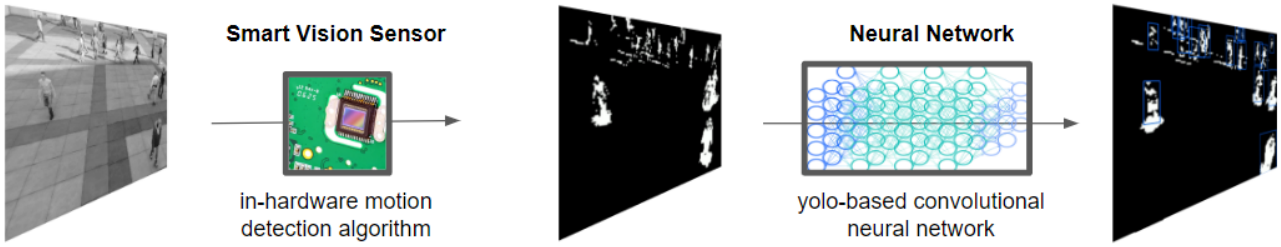

Figure 2.1: General Overview of the Framework. A Smart Vision Sensor (SVS) captures grey-scale images with a 160x120 resolution and applies an in-hardware algorithm to produce binary motion maps. The motion maps are then processed by a neural network to perform object detection, triggering and counting.

2.1 Dataset

Since the micro-controller has been internally implemented by Fondazione Bruno Kessler and there is no literature trying to do the same as us, we don't have a baseline to compare our results to, and thus, we don't have a benchmark for evaluating our models. For this reason, we have to decide on a bench-marking dataset on our own. Even though detection is our main task, we cannot use popular image detection datasets such as COCO [52] or VOC [27], because we need videos to produce motion maps, for this reason, we look at tracking datasets where detection bounding-boxes are provided. One of the most common and famous tracking datasets is MOT17 [22], which is used to train and test models for tracking pedestrians in crowded public spaces like streets and squares.

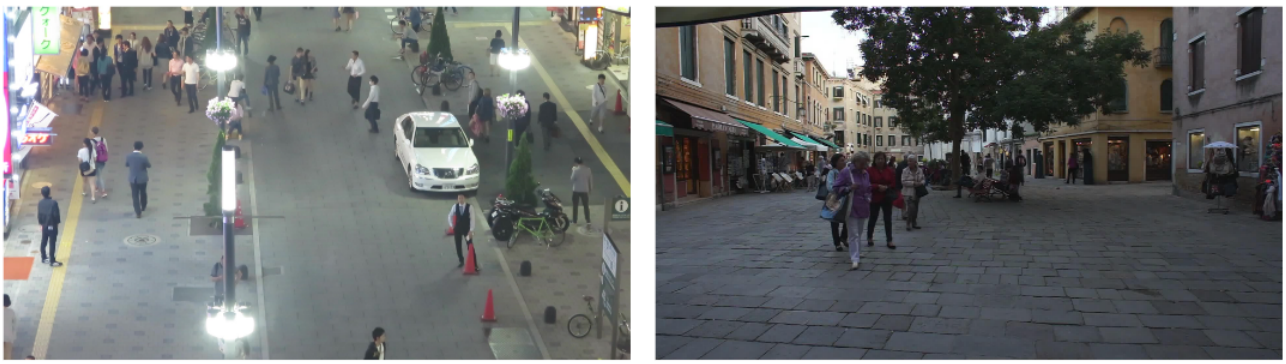

Figure 2.2: Sample Sequences from MOT17 [22]

There is a previous version of this dataset, namely MOT15 [61] [22], containing 21 sequences, but given the popularity of this task, newer and improved versions have been released over the years. In 2020, MOT20 [23] was made available to the public for the MOT20Challenge [22] and contained

very challenging video sequences with hundreds of people appearing in the same frame. Input data is fundamentally important in machine learning, thus dataset choice must not be taken lightly, we decide to exclude MOT20 because it becomes too challenging to detect people in very crowded places from the binary motion map with a resolution of 160x120. After having excluded MOT20, we check MOT17, which is a very diverse dataset containing both videos with pedestrians passing near the camera and sequences recorded further away. The dataset is split into two parts: training and testing. For the training split there are two types of label available: ground truth annotations, which contain high-quality bounding boxes of the walking pedestrians annotated by hand, and public detections, which contain low quality bounding boxes, found with detection algorithms, these latter annotations should be used to compare tracking algorithms. In fact, the tracking tasks used to be divided into two parts: first detecting the objects (where the people are in the image) and second tracking their movement between frames, assigning a single ID to the same person along multiple frames.

Since public detections are common for all researchers, they can compare tracking algorithms without taking into account the accuracy of the detections they have. Since these public detections contain many errors, nowadays benchmarks use the so-called private detections.

Unfortunately, in MOT17, the hand-written annotation is not available for the test set and this implies that we can use only the training split which contains only 7 sequences.

We also notice that most of the sequences of the training set have been recorded with a moving camera, thus we cannot apply our background suppression algorithm since it requires a static camera. After having filtered out all the sequences with a moving camera we remain with only 3 sequences. Since the original dataset contains more than 9 times the number of videos we decide to split these three videos in train/test in a particular way: the first 60% of the video's frames are used for training, and the next 20 frames are not used and the remaining frames are used in the test set.

We decide to integrate our data with new datasets by adding sequences from SynthMOT [28], another tracking dataset released in 2021 containing hundreds of videos recorded from both moving and static cameras. These videos have not been recorded in real scenarios, but they have been generated with a simulation. In the original paper, it was shown that training detection algorithms on data augmented with these simulated sequences improve the final scores on MOT17.

Figure 2.3: Sample Sequences from SynthMOT [28]

Even though SynthMOT is meant to be used only as a training dataset, we use it for both training and testing, justifying this decision with the fact that we don't have enough videos from MOT17. This choice is further justified by the fact that after the videos have been processed by the micro-controller simulator, it is impossible to distinguish the synthetic data from a real video since the motion detection algorithm is robust to color augmentations. Moreover training on one dataset and testing on another one would create a domain shift that would make it hard to trace back problems if they were to happen.

Moreover, we notice that in both MOT17 and SynthMOT, the number of people present in the frame at every time step is always bigger than one. In fact, those sequences contain on average 10 to 30 pedestrians. Thus, if we want to train our network for the triggering task we need to have video sequences that contain segments that do not contain any person inside the camera view. We

need to have both positive and negative samples to train our network, otherwise, if we trained our network to predict if there is at least one person in the frame (triggering) on the MOT17 dataset, the network could just learn to predict always true and that would be a perfect prediction. For this reason, we create a novel dataset containing seventeen novel sequences that contain zero one or two people. Moreover, this new dataset introduces a big domain shift since the size of the bounding boxes is very different: MOT17 contains views from security cameras and there could be dozens of people in a single frame. Our dataset has videos containing people walking just a few meters from the camera, thus one or two people can take most of the view.

This dataset has been thought specifically to be used for this micro-controller, in fact, a problem with the other datasets is that people further away from the camera do not generate enough movement to be detected by the motion detection algorithm, moreover there are many people which are static and thus are not detected in the motion map. The dataset we built, called Streets23 can solve all of these problems.

2.1.1 A new Dataset: Streets23

We describe the procedure we followed to build the Streets23 dataset. The data gathering part consists in recording brief videos in different environments with different lighting conditions, capturing people entering and exiting the sight of the static camera at 30 fps. After collecting all the videos, a first pre-processing step crops the frames to have a width/height ratio of 160/120.

To produce some initial annotation bounding boxes we use YOLOv8 by Ultralytics for detection and strong-sort [25] and byte-track [100] for the tracking part. We notice that even though byte-track is a state-of-the-art method on MOT17 its performances on Streets23 were not satisfactory with a lot of identity switches and imprecise bounding boxes. For this reason, we switch to the strong-sort algorithm which is explained below.

Figure 2.4: Sample Sequences from Streets23

Strong Sort is an upgraded version of deep-sort [94]. Deep-sort is one of the most famous tracking algorithms based on two main modules: a re-identification branch where a siamese network extracts appearances features that will be used to re-identify a person in different frames and a motion branch, which uses Kalman filters [71] to mix information from previous frames like location and velocity with the detections of the current frame. The information from the 2 branches is then merged using a cost matrix that sets the error for matching objects in nearby frames. The final association that minimizes the cost association is carried out with the Hungarian algorithm [48]. Strong sort improves upon deep sort in the following way: the appearance branch is improved by replacing the re-identification siamese network with a new one and the re-identification features are updated using an exponential moving average (strong sort used to collect all the features at each time-step), for the motion branch they take into account camera compensation (for sequences without a static camera) and use an improved version of the Kalman filter.

After having collected these bounding boxes we manually check them and perform the following corrections: fix identity switches, manually re-draw wrong bounding boxes, and manually insert the missing bounding boxes.

We then proceed to split the dataset into thirteen videos for training and three for testing. The

videos for testing have been selected to be mostly recorded in different environments with respect to the training set to test the capability of the trained system to generalize.

Reproducibility. To summarize, we structure our dataset as follows: 3 sequences from MOT17 for training, 23 sequences from SynthMOT for training, 14 sequences from Streets23 set for training; 3 sequences from MOT17 for testing (the second part), 3 novel sequences from SynthMOT for testing and 3 novel sequences from Streets23 for testing. We use Streets23 to test the triggering task since it contains both frames with and without any object.

Moreover, we apply a selection on the bounding boxes proposed by the ground truth, in fact, both synthMOT and MOT17 9th column of the annotations contains the visibility information about the object. Keeping objects with a visibility of zero could mean having bounding boxes even when a person is hidden by a wall, thus we decide to keep bounding boxes for which at least 44% of the target is visible. Moreover, MOT17 8th column contains the class label that could refer to many types of objects like: moving people, static people, cars, reflections, bicycles, and many more; we decide to keep only the first class which is attributed to moving pedestrians.

2.1.2 Data Augmentation Pipeline

Data Augmentation is a fundamental part of training machine learning algorithms. Since our neural network needs to be trained on the output of the SVS algorithm, we need to optimize this step so that it doesn't require us to recompute the simulation every time we run the code.

For this reason, we create a pipeline capable of storing the simulation results so that we can efficiently access them in future runs. This subsection will describe how this pipeline is implemented while Figure 2.5 visually summarizes the pipeline.

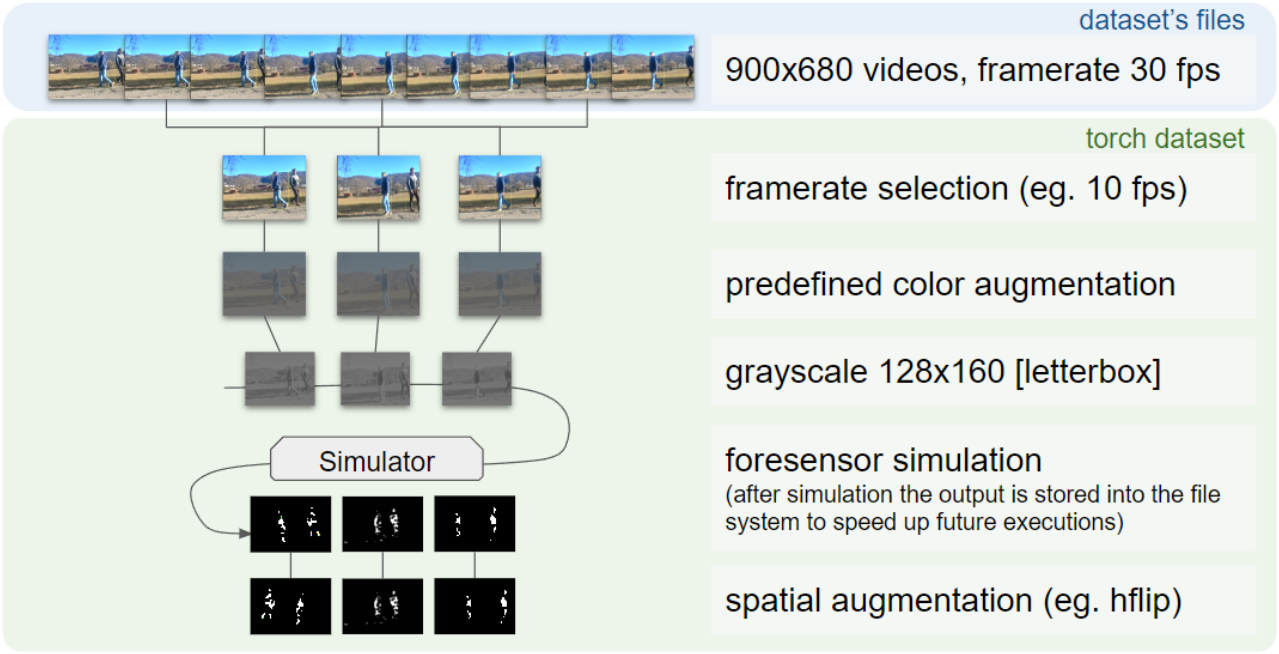

Figure 2.5: Visual description of the data augmentation pipeline

The data augmentation is divided in 2 parts: the first one implements frame rate selection and color augmentation and happens before of the simulation, while the second one happens after the simulation and implements spatial augmentations. Since we store the partial results on the file system after simulation, the settings for color and frame rate have to be decided a priori.

Color Augmentations. For each sequence we load the images and information about the video frame rate. Afterward, the frame rate augmentation is performed by selecting one image every

$requested\_framerate/video\_framerate$. The idea behind supporting multiple frame rates is that it is another type of data augmentation. In fact, a video with a lower frame rate could be seen as a video with a higher frame rate, but with slower-moving objects.

We then load the video as a sequence of RGB images with a high resolution. The next step is to apply color augmentations, the supported types of augmentations are the following: brightness, contrast, saturation, sharpness, hue, gamma and noise, but, after noticing the SVS algorithm is robust to these kinds of augmentations we decide to keep only the most challenging ones which are: brightness reduction and gaussian noise addition. Once we have a specific color augmentation configuration we apply that exact configuration to all the video's frames to obtain the final color-augmented sequence.

The Input Resolution Problem. A problem we encountered, in the beginning, was resizing the frames to be 160x120 pixels before passing them through the simulator (to keep the original proportions we used the letterbox resizing, which introduces paddings that additionally decrease the amount of information contained in the frame) caused the resolution to be so low that the output of the simulator was not containing enough information and thus making it impossible to learn for the neural network. To fix this problem we simulate the video with a higher resolution of 384x480 pixels and then crop a section between 160x120 pixels to 384x480 and resize it to 160x120 during the spatial augmentation for training. For testing, a similar procedure is adopted, but the choice of the crop is deterministic (moreover we notice that motion maps have a much higher quality with lower fps videos). We have to notice that the high-resolution simulation happens only for sequences from MOT17 and SynthMOT because the average height of the bounding boxes is ca. 4% of the height of the frame and it is not used in Streets23 where the average height of the bounding boxes is ca. 33%. After having applied the color augmentation and converted the RGB to greyscale, we pass the video through our simulator and produce a binary motion map.

We want to put more emphasis on the importance of the high-resolution simulation because it caused a lot of problems in the development of the thesis, in fact, training a neural network on low-scale motion maps brought many negative results. The problem was hard to spot too because, when checking the motion map videos, it was possible to distinguish some people's silhouettes by their motion, that the human eye can recognize. Unfortunately, the neural network could only look at one frame at a time and couldn't exploit this movement pattern. Thus we tried to concatenate multiple frames on the channel dimension to see if the network could pick up the motion information, but, even if the overall accuracy of the network was increased it wasn't as good as we expected. The idea of cropping the bounding boxes came from Ancilotto et al. [6] paper on a similar subject. To have a more intuitive understanding of how much the input resolution impacted the detection task, we can have a look at Figure 2.6, as we can see the model trained on the greyscale (orange) was times better than the models trained on the motion maps with many different configurations (all the others). Improving the resolution greatly bridges this gap. Note: the test set was formulated in a different way in this experiment: the training set was the first half of each video while the test set was the last quarter of each video.

Spatial Augmentations. After having processed the motion maps spatial augmentations are applied: we perform cropping ad resizing which can be seen respectively as shifting and rescaling, thus making it an affine augmentation without rotation. The other spatial augmentation we use is H-Flip which mirrors the image on the y-axis. Spatial augmentations are fundamental to compensate for the size of the dataset to greatly improve generalization in the neural network making it capable of detecting people in unseen sequences too. Figure 2.7 shows some example motion maps that will be used to train the neural network (left) and one of the original frames for comparison (right).

The code implementing the spatial image transformations used in the main dataset has been taken from the DETR codebase [17] and modified to be used for this project. While the simpler version of data augmentations used in the policy_generation code has been implemented from scratch.

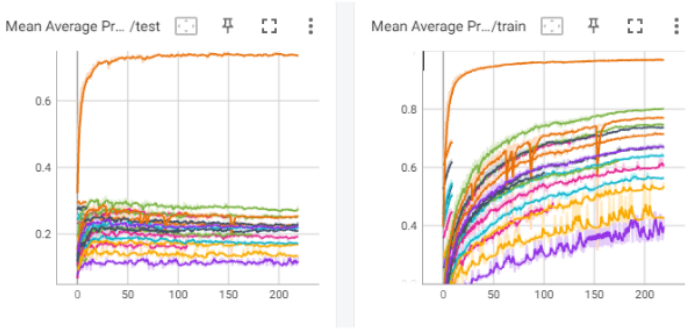

Figure 2.6: The top-scoring orange line is obtained by training the network on the grey-scale images; while all the other lines have been obtained by training the network on the output motion maps using static parameters and different initial configurations for the simulator.

Figure 2.7: Example of a simulated sequence (left) and the original frame (right)

2.2 Detection Neural Network

This section describes how YOLO's architecture [45] has been adapted to our problem and offers an overview of its main components. Yolo can be summarized in 3 modules: backbone, which is a simple convolutional neural network; upscaling, which mixes higher resolution feature maps from earlier layers and the upscaled output of the backbone and returns a set multi-scale feature maps; detection-head, which produces the final feature map where the loss is computed on; as shown in the figure below . Since the loss computation is the heart of Yolo, we will spend the next paragraph dissecting how it is computed.

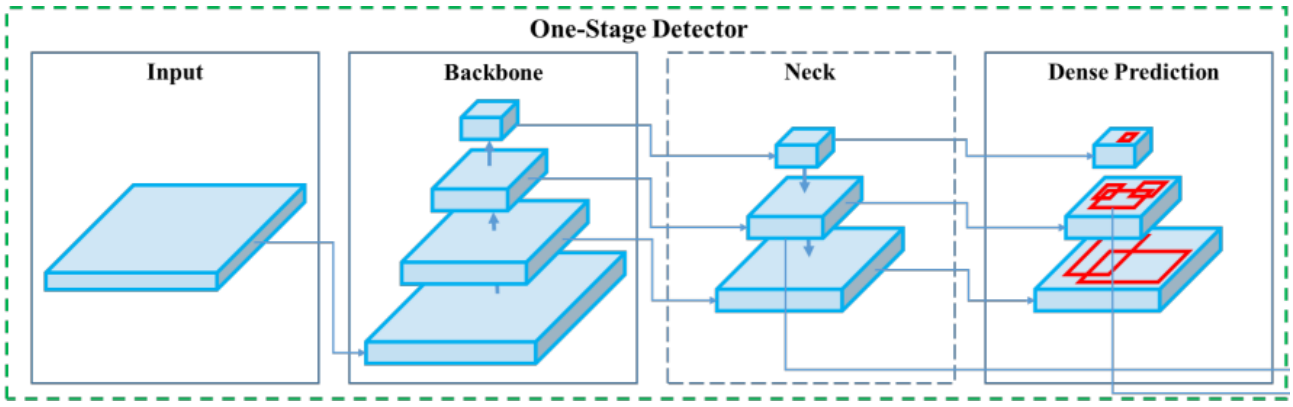

Figure 2.8: Visualization of YOLO's architecture [13]

2.2.1 Loss Functions

It is important to understand how the loss is computed to understand how to properly train the network, in fact, in the first experiments, we only use one feature map, but this leads to problems in the assignment of the ground truths, because, in yolo, multi-scale features maps should target specific object sizes, thus, by having only one feature map it is possible to have reduced performances.

Loss computation is performed as follows: assigning each object to one or multiple feature-maps

Figure 2.9: The first two equations describe how the position in the image is computed and can be summarized as the grid point location which is represented by c_x and c_y and the dislocation which is computed on the logits t_x, t_y of the network. The last two lines describe how the size of the bounding box is computed: p_w p_h are the anchors' sizes, while t_w and t_h are the logits predicted by the network

$$b_x = (2 \cdot \sigma(t_x) - 0.5) + c_x$$
$$b_y = (2 \cdot \sigma(t_y) - 0.5) + c_y$$
$$b_w = p_w \cdot (2 \cdot \sigma(t_w))^2$$
$$b_h = p_h \cdot (2 \cdot \sigma(t_h))^2$$

anchors and grid points, where a grid point is a pixel in the output feature map. Each assignment is bounded to a 6 neurons output in the feature map, the first 4 neurons control the position and size of the bounding box, the fifth one is the confidence score and the last one remains unused since we only have one class (with multiple classes there would be multiple neurons like in a classification layer). We use a single class because all annotations refer to pedestrians, thus, we don't have a second label type to train the network on.

The loss used to train bounding box prediction is the CompleteIoU [102] [103] loss which is an improvement over the IoU loss which is also a metric used in evaluation thus directly optimizing it improves the final performance. We do not pass the raw logits of the neural network to compute the loss but we apply a pre-processing that embeds information about the overall position in the image, thus the network is not required to predict the position over the whole image, but only the dislocation with respect to its grid point. The formulas used to predict the location are summarized in the panel below 2.9.

As described before, the 5th output neuron is the confidence of the network in predicting whether there is an object or not, it is learned as a probability, and, at test time, an object is detected when this value is bigger than a given threshold that we set to 0.5. The loss used is the cross-entropy loss. There is an important parameter regulating this loss called gr, when this parameter is bigger than one (default set to 1) the ground truth confidence is re-scaled by the accuracy of the predicted bounding box in terms of IoU, thus, if the network cannot learn how to properly set the bounding boxes the confidence should always be close to zero and thus be easier to predict. This concludes with the fact that even though we could observe a small overall loss, the network could just be predicting bad bounding boxes, this concept will be useful when optimizing the neural network architecture.

Since we only have one class, we do not have a classification loss, but we extended the original yolo code by adding two more losses that are used to train the output neurons for the counting and triggering tasks. The triggering task should predict whether there is a person or not in view, thus, the output is a probability between 0 and 1 and, when the probability is bigger than 0.5, we think there is a person in sight; the loss adopted is the cross entropy loss. The second output is the count of how many people we think there are in the image and this output is trained using the mean squared error. It is true that we could infer the number of people in the image by counting the bounding boxes but this has some drawbacks: 1st it is more computationally expensive since we need to go through the non-maximum suppression step and 2nd an architecture end-to-end capable of only performing counting could have many fewer parameters since the detection task is way harder.

A set of hyper-parameters to regulate the weighting of the losses had to be set. Our choice does not vary much from the original yolo implementation, but we give a little more weight to the object detection loss. This can be justified by the fact that the variance of the bounding boxes in the dataset is pretty low with respect to the original dataset where yolo was trained, thus, people have more or less the same size, thus recognizing their overall position is more challenging than finding a good bounding box.

2.2.2 Architecture Overview

We now explain the main architectures we use in our experiments. Each architecture has its own set of hyper-parameters that can greatly affect its performance and number of parameters. The next section will focus on how we optimized these parameters, while this subsection will give a general overview of the architectures.

Yolo's architecture has been built to be very modular and it is described by yaml files. Thanks to this aspect we can use yolov8 architecture even though the code that processes the yaml configuration has been borrowed by yolov5. The architecture configuration file specifies a sequence of CNN blocks and how to compose their outputs to build the full network, formulating the architecture in this way allows the use of partial outputs of the backbone in later stages of the code as skip-connections. Both yolov5 and yolov8 architectures are pretty similar, having normal convolutional blocks concatenated by bottleneck blocks, with yolov8 being deeper and having more bottlenecks. We also rescaled the channels dimension from the original configuration following the ideas from [36] and reduced the number of parameters from 1.7M to 180K. These architectures based on yolov5 and yolov8 will be later referred to as **yolov5** and **yolov8**.

The third architecture is built by substituting yolo's backbone with PhiNet [16], a CNN built to run on devices with small computational power. PhiNet code implements it as a unique block, thus, it is hard to adapt it to yolo's code to use intermediary layers logits, but after some code re-engineering, we successfully integrated PhiNet in the yolo architecture. This architecture will be referred to as **phiyolo**.

The fourth architecture is built from scratch to test the performances of low-parameterized models. This architecture is shallow and integrates both convolutions and feed-forward neural networks. Convolutions are the building block of all the previous architectures and are useful to extract local information and their concatenation allows them to linearly increase their total span of attention. The feed-forward network is an experimental addition and is justified by the following ideas: multi-layer perceptrons can process any type of information and they have already been used in computer vision for example in NERF [62] thus proving their utility, recent works are trying to add transformers into computer vision tasks [24], one of the major advantages of transformers is their ability to look at the whole image in a single step; the idea behind FFNN is similar, they can extract information from the whole image in a single step, thus could be very useful for the triggering/counting task in identifying noise from real objects. This architecture is built by alternating CNN/FFNN blocks and downscaling layers, which consist of average pooling with a stride of two. A similar architecture has been already applied in mobilenetV2 [80] and mobile-former [20], obviously the concept of alternating blocks of local attention and global attention is the only concept in common with these papers (note: attention is not the famous formula used in the original transformers paper[91], but just the exemplification of the concept of "where the network can look to make its prediction"). This architecture will be referred to as **mlp2**.

The last type of architecture consists in concatenating a PhiNet with a YOLO detection head, omitting the neck part of the architecture which distinguishes this architecture from the third one. The reason behind the simpler type of architecture is that the previous ones required to keep into memory multiple residual layers to be used in the neck section of the network. This requirement highly increases the amount of SRAM required by the edge device to run the model; thus, this last architecture is the most hardware efficient one, but, achieves lower performances. This architecture will be referred to as **phismall**.

2.2.3 Training Settings

This section describes the choices of hyper-parameters adopted to train the neural network. The objective of the thesis is to compare different methods to perform object detection on different videos and datasets, so the code needs to support different initialization options. For example, we can choose the initial parameters for the simulator, the frame rate, which color augmentations we want to use and which dataset we want to train the network on. After having defined these the experiment's settings we build the dataset and the neural network. The neural network implementation is carried out using Pytorch [70], which is one of the most popular deep learning frameworks that offers many tools and utilities to train neural networks. Different learning rates and batch sizes have been tested and the ones that are used in the final version are 0.001 as the learning rate for the backbone, and 0.003 as the learning rate for the other modules of the network. The batch size adopted is equal to 128 this is a trade-off between optimal batch size, which would be a bit smaller, and training speed. The algorithm adopted to iteratively optimize the weights of the network is AdamW [55] with prior resizing of the gradients using normalized gradient clipping (note that other optimizers like SGD do

not take into account the "velocity" when updating the parameters, thus they would have a different learning rate). Another important hyperparameter for object detection-based models is the detection threshold, which we set to 0.5 for all our experiments. This hyperparameter could be further improved for each architecture, in fact, we notice that smaller models could work better with a lower threshold (eg. 0.33), but trial and error have shown that 0.5 is a robust choice. Finally, the non-maximum suppression (NMS) threshold is set to 0.3, because multiple predictions for the same object were a quite recurring problem, setting an IoU threshold this low makes it hard to detect overlapping objects, for this reason, in a real-case scenario, this hyperparameter should be finetuned for the specific scene we are observing.

The experiment then runs for 120 training epochs. A summary is generated once an epoch has been completed, it contains real-time information about the state of the training of the network.

Idea 1. After 40 epochs we reset the optimizer, which is Adamw with a momentum of 0.92 and *velocity* momentum of 0.9999. After forty epochs the loss has already decreased a lot, thus the generated gradients are smaller than the starting ones if the parameter is in a plateau. This causes the updates to be very close to zero because the beta used to update the running mean of the *velocity* is almost one (0.9999). To simplify the intuition: the Adam optimizer divides the gradient by the "*velocity*", which is the running mean of the absolute value of the gradient, but this value tends to become too stable after some steps. Decreasing $beta_2$ which is the coefficient used to update the running mean of the velocity isn't good either because this would make the update too much independent from the slope of the gradient. Thus, we opt for a fresh re-start of the optimizer after 40 epochs. This "trick brings a gain of 2-3 AP_{50} points on average.

Idea 2. For the last 10 epochs, we substitute AdamW with Stochastic gradient descent with momentum, a learning rate of 0.0001 and a cosine learning scheduler. The idea is to fine-tune the parameters with a very small learning rate. The scheduler we use is the cosine annealing learning rate scheduler. This "trick" brings a gain of 2-3 AP_{50} points on average as well.

Idea 3. If we are training the network for the triggering task, we increase by three times the weight of the triggering loss (over the detection loss), in this way we suggest the network should prioritize recognizing whether a person is in the scene or not. We apply this change in the loss at the end, for fine-tuning purposes.

Failed Ideas. We also try other optimization ideas, but they don't seem to affect the final performances:

- Rescaling the loss as proposed by Susano et al. [88] by penalizing more solutions with bad scores for a non-differentiable reward.
- Finetuning on a specific dataset. Strangely training for 120 epochs on MOT17+SynthMOT+ Streets23 outperforms training on 120 epochs on MOT17+SynthMOT+Streets23 plus additional 10 epochs excluding Streets23.
- Setting the loss parameter "gr" to 1 instead of 0.8 (which lower the predicted confidence for badly placed bounding boxes, which should be useful when using only one feature scale).
- Changing the bounding box regression loss to be less severe when the anchor area is very different from the label area.

2.2.4 Baseline Results

This section reports the preliminary results and some visualization obtained by running the above-explained experiments: after the training is completed, we start the evaluation by gathering the following metrics:

- For detection: average precision with IoU of 0.5 (AP_{50}) and mean average precision (MaP)

- For triggering: F1 score ($F1$)
- For counting: mean absolute error (MAE)

Experiment 1: grey-scale. We set three baselines to have some initial results to compare the later models and policies' performances. The first baseline is set by training a **phiyolo** neural network with 104K parameters on the grey-scale videos, the initial architecture hyper-parameters were taken from [67] with a bit of later fine-tuning, especially for the neck. The second baseline is achieved by training the previous model on the motion maps generated from the simulator using predefined parameters; the simulator's parameters were chosen from a recommendation from an author of the simulator and are equal to 1 for delta_close, 1 for delta_hot and 5 for delta_hot. The last baseline is obtained using a really simple blob detection model on the motion maps.

Baseline: grey-scale 4fps phiyolo 104K parameters				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	1.33	0.62	0.29
SynthMOT	//	4.88	0.66	0.30
Streets23	0.88	0.42	0.82	0.35
average	0.88	2.21	0.70	0.31

By comparing 2.2.4 and 2.2.4, we can notice that the grey-scale model outperforms the motion map based model on Streets23 and MOT17, while it has similar performances in SynthMOT. We can assume that bigger greyscale-based models trained on more data would always outperform motion map based models, but this is not our objective. Our aim is to simplify a rather challenging task like object detection to run algorithms directly at the edge, thus these preliminary results show that motion maps can greatly help simplify the task obtaining results comparable with grey-scale images. Figure 2.10 visually compares the results on the Streets23 dataset to give a more interpretable intuition.

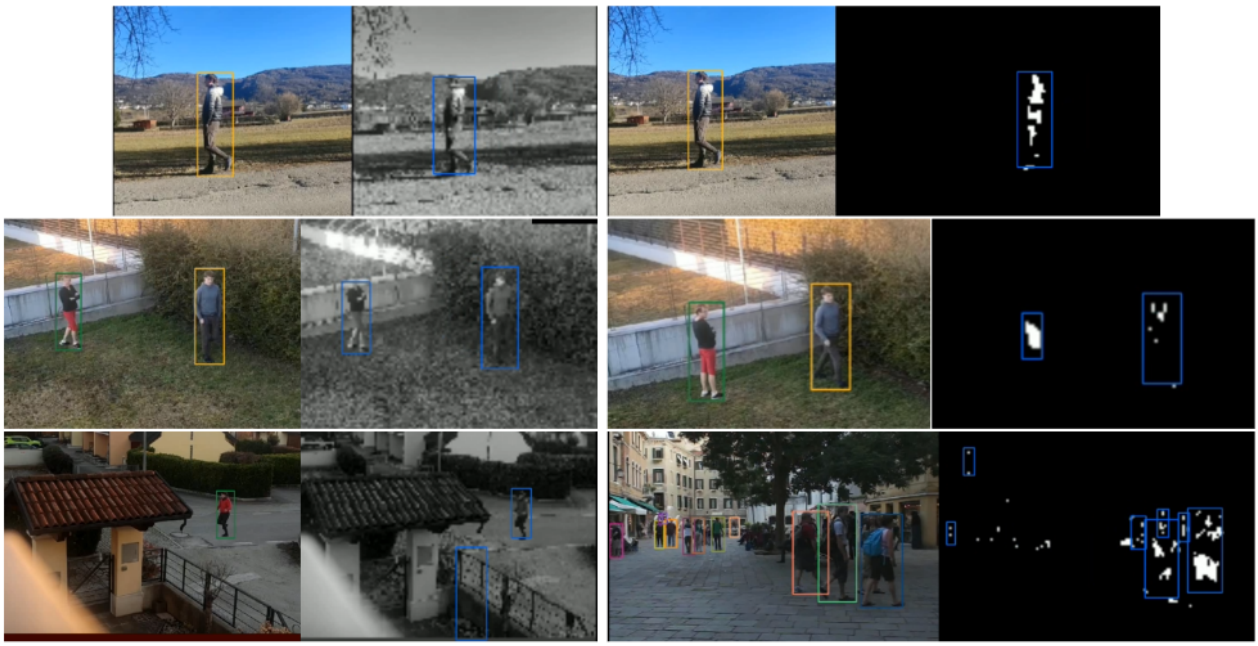

Figure 2.10: Comparison of the detections of the grey-scale model versus the motion map model. (top) When the motion map is precise the results are comparable with the greyscale. (center) Grey-scale is overall more precise. (bottom-left) Errors in the grey-scale model can be totally off. (bottom-right) Errors in the motion-map model are often caused by non-detected motions of people too far off the camera.

Baseline: static-svs[1,3,5] 4fps phiyolo 104K parameters				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	3.61	0.51	0.25
SynthMOT	//	2.69	0.64	0.28
Streets23	0.88	0.21	0.54	0.23
average	0.88	2.17	0.56	0.25

As we can see from Table 2.2.4, the model trained on the motion maps already offers a solid baseline, almost comparable with the grey-scale models; moreover using a neural network seems much more effective than using a simple blob detection model.

Baseline: static-svs[1,3,5] 4fps blob-detection 0 parameters				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	5.4	0.14	0.07
SynthMOT	//	7.6	0.32	0.12
Streets23	//	0.47	0.10	0.04
average	//	4.5	0.18	0.08

Blob Detection Model. The blob detection model is a dummy model with zero parameters, it should not be intended as a real attempt in building an effective object detection model. Its main purpose is to prove the effectiveness of the simulator's optimization policies, because, since it does not require training, its performances are only determined by the quality of the motion map.

The model works by finding all the blobs in the motion map three times, after each iteration, a dilation with a full kernel 3x3 is applied. Dilation is useful to connect blobs between each other, for example, it is useful when the legs of a pedestrian and its torso are disconnected in the motion map; usually, applying dilation allows to connect of these parts generating a unique bounding box. Once the proposed bounding boxes have been found, they get filtered with the following requirement: the center of the bounding box should be white, while the border of the bounding box should be mostly black (where white means motion detected and black the opposite).

Frame-rates comparison. We test the performances of the baseline **phiyolo** model when trained on the motion maps generated at different framerates to observe changes in performances. If not differently specified, all the default experiments have been run at 4 fps.

Baseline: static-svs[1,3,5] phiyolo 104K parameters				
Dataset+fps	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17 [1fps]	//	3.05	0.42	0.14
SynthMOT [1fps]	//	3.14	0.59	0.23
Streets23 [1fps]	0.76	0.26	0.63	0.24
average [1fps]	0.76	**2.15**	0.55	<u>0.20</u>
MOT17 [4fps]	//	3.61	0.51	0.25
SynthMOT [4fps]	//	2.69	0.64	0.28
Streets23 [4fps]	0.88	0.21	0.54	0.23
average [4fps]	**0.88**	2.17	**0.56**	**0.25**
MOT17 [15fps]	//	2.93	0.54	0.25
SynthMOT [15fps]	//	2.24	0.59	0.25
Streets23 [15fps]	0.31	0.11	0.17	0.04
average [15fps]	0.31	1.76	0.43	0.18

We can note that at 15 frames per second, the performances decrease drastically, we presume that a better optimization of the simulator's parameters could help bridge this gap.

Enhancing the input. We run some experiments extending the motion map with more temporal information, more specifically we compute the motion history image, which is defined as:

$$MHI_{i,j,t} = max(MHIi, j, t - 1/2, MM_{i,j})$$

where MHI is the history motion map, MM is the motion map generated for that frame and i,j are the coordinates in the image plane with $i \in 1..H$ and $j \in 1..W$.

Figure 2.11 shows an example of MHI. The drawback wrt. motion maps is that MHIs are not binary thus a grey-scale image could be used instead (although MHI are sparse).

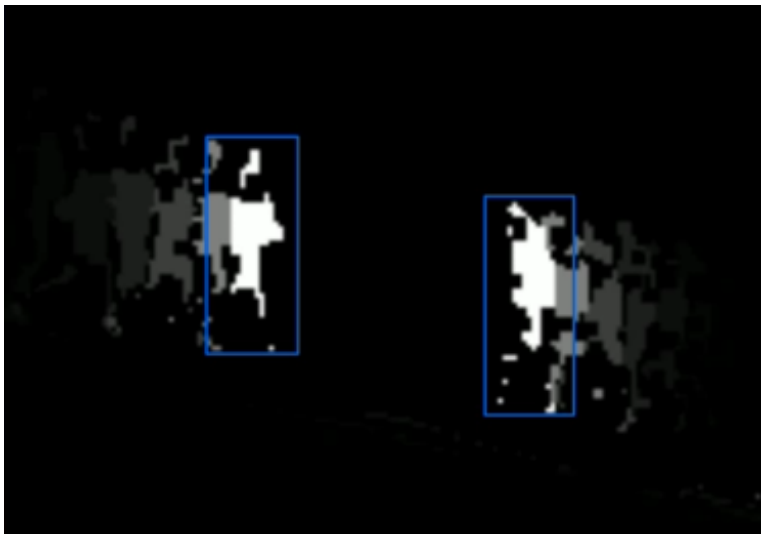

Figure 2.11: Example of motion history image. The whiter the pixel value, the more recent the motion detection is. The bounding boxes (blue) show the real position of the pedestrian.

motion-history-image-svs[1,3,5] 4fps phiyolo 104K		
Dataset	AP50: Detection	MaP: Detection
MOT17	0.56	0.30
SynthMOT	0.70	0.32
Streets23	0.70	0.28
average	0.65	0.30

Another experiment is carried out by concatenating the grey-scale image and the motion history image on the channel dimension. This is the type of input that contains the most amount of information (since the sensor does not produce RGB images) but it's also more expensive to process. We train the baseline model on this dataset but we must note that, since this type of input is more complex than a simple binary motion map, it is possible that smaller models have troubles in using it to its full potential (or quantized models), thus, since we want to make detection as simple as possible, we prefer only using the motion map.

motion-history-image-cat-grey-scale-svs[1,3,5] 4fps phiyolo 104K		
Dataset	AP50: Detection	MaP: Detection
MOT17	0.60	0.30
SynthMOT	0.69	0.31
Streets23	0.73	0.29
average	0.67	0.30

We can notice that performances decrease on the Streets23 dataset when using both the greyscale and the motion map with respect to only using the greyscale image. A possible explanation could be

that the network prefers to use the MHI because it simplifies the problem, probably, a bigger network could be able to leverage information from both sources of data.

Figure 2.12: Predictions for the MHI concatenated to the greyscale. since the input has two channels, the visualization shows the average between the 2 channels. As we can see from frame 3, even though the pedestrian stops moving and the motion map becomes black, the greyscale helps detect the target anyway.

2.3 Optimizing Neural Network Architectures

As described in the previous section, each of the proposed architectures has its own set of hyper-parameters that influence the size size of the model. In this section, we explain how we optimized these hyper-parameters to search for low-parameterized architectures without compromising the performances. We first describe the hyper-parameters of each architecture, afterward we summarize commonly used hyper-parameter optimization techniques and we finally describe our experimental approach based on heuristics.

hyper-parameters set The four types of architecture we try to optimize will be called **yolo5**, **yolo8**, **phiyolo** and **mlp2** and respectively: yolov5n [45] with the number of channels rescaled, yolov8n with the number of channels rescaled, yolov5's neck using phinet [67] as the backbone, and a hybrid shallow architecture that uses both CNNs and MLPs. Every architecture based on yolo has a width and depth hyper-parameter, the width is one of the most important because it sets the multiplier for the number of channels in every feature map; the depth coefficient influences the depth of the network by setting how many times some blocks are repeated. The phiyolo architecture has 5 additional hyper-parameters: alpha, which is a channel multiplier, beta, which determines whether the channels increase or decrease, t0, which determines the base expansion factor, squeeze-excitation, which determines whether to use the squeeze excitation blocks [39] and inputconv2d, which determines the usage of a basic conv2d or an additional PhiNetBlock but with a stride of 2. Finally, the mlp2 architecture has hyper-parameters regulating the number of neurons in the multi-layer perceptron nets, a channel multiplier for the convolutions and a parameter choosing the type of downscaling in the last layer.

general hyper-parameters optimization techniques Two very common options to optimize hyper-parameters are grid search and random search, the first one simply tries every possible combination of hyper-parameters in a predefined order thus requiring the user to complete the whole search process, while the latter randomly try different configurations, thus allowing the user to set a maximum number of iterations. Bayesian optimization improves on these types of search by choosing the configurations to try in a smart way by learning a probabilistic model that tries to predict a function

(eg. the mean average precision) after having observed the function value at some configuration, then tries the configuration that maximizes the expected score and uses the results to iteratively update the predictive model. This is a really sound approach but 2 major problems make it unfeasible: the set of hyper-parameters changes for each architecture, making it necessary to learn a predictive model for each architecture and, most importantly, getting a MaP score for the function to evaluate is very expensive, thus we cannot try many configurations. For this reason, we try an experimental approach that aims at reducing the time needed to compute the mean average precision score. After having implemented this new score we adopt random search due to the low cost of computing it, but substituting random search with Bayesian optimization would surely help reduce convergence speed.

2.3.1 Predicting the Network Performances Without Training

This subsection gives an overview of a novel approach based on heuristics that was adopted to predict the goodness of a model without training it by learning a surrogate model, similar to bayesian optimization, but with the difference that this model input won't be only the configurations use the configurations as input, but some heuristics too, thus making it harder to identify configurations that maximize this surrogate function thus having to rely on random search (a possible improvement could be adding some local search by exploiting the fact that good configurations will probably be close to other good configurations).

The next paragraph describes the heuristics we tried, how we defined our surrogate function and the scores obtained using the optimized architecture. The first heuristic is called Neural Tangent Kernel (**NTK**) [43] and is introduced by Jacot et al. as a function to describe the learning capabilities of a neural network. They prove that neural networks can be seen as support vector machines with a special kernel, this kernel is defined as the gradients of the network's parameters with respect to the loss, thus, if back-propagating the loss for different samples generates different gradients the network will have an easier time learning a line to separate them, they then show that the neural tangent kernel does not change a lot with training, thus making it possible to predict how good a neural network is without training it. The procedure we adopt to compute this heuristic is similar to the PCA: we build an NxM matrix where N is the number of samples and M is the number of parameters in the neural network, we then compute the NxN correlation matrix between the samples and we finally calculate the eigenvalues. If there is a big ratio between the biggest and the smallest eigenvalues it means that there is a structure in the hyper-dimensional gradient space and thus the network should be able to classify the samples with more easiness. Figure 2.13 shows an intuition. Mok et al. [63] then show that NTK heuristics hold mainly for simple networks, while, for more complicated networks, it is necessary to train the network a little.

$$\Theta\left(x, y; \theta\right) = \sum_{p=1}^{P} \partial_{\theta_p} f\left(x; \theta\right) \partial_{\theta_p} f\left(y; \theta\right)$$

The NTK as the inner product of the gradients between two different samples [43]. f is the neural network whose output depends on the input (x/y) and the set of parameters θ

The second heuristic is called **linear** activation unit score [59] and is based on a simple intuition: if different samples have similar activations after a rectified linear unit layer it is probable that the network will have trouble in separating the two samples. Mellor et al. then show that these simple heuristics are useful at classifying neural networks without training on multiple benchmarks for neural architecture search, while other, approaches usually work only on a specific benchmark. To compute this heuristic there are two steps: during the forward pass we collect all the activations in the batch and compute the NxN matrix, with N being the number of samples, which represents the dissimilarity between the samples. Afterward, we compute the logarithm of the determinant of the so-found matrix with the intuition that a small determinant would mean dissimilarity between samples thus improving the learning capabilities of the network.

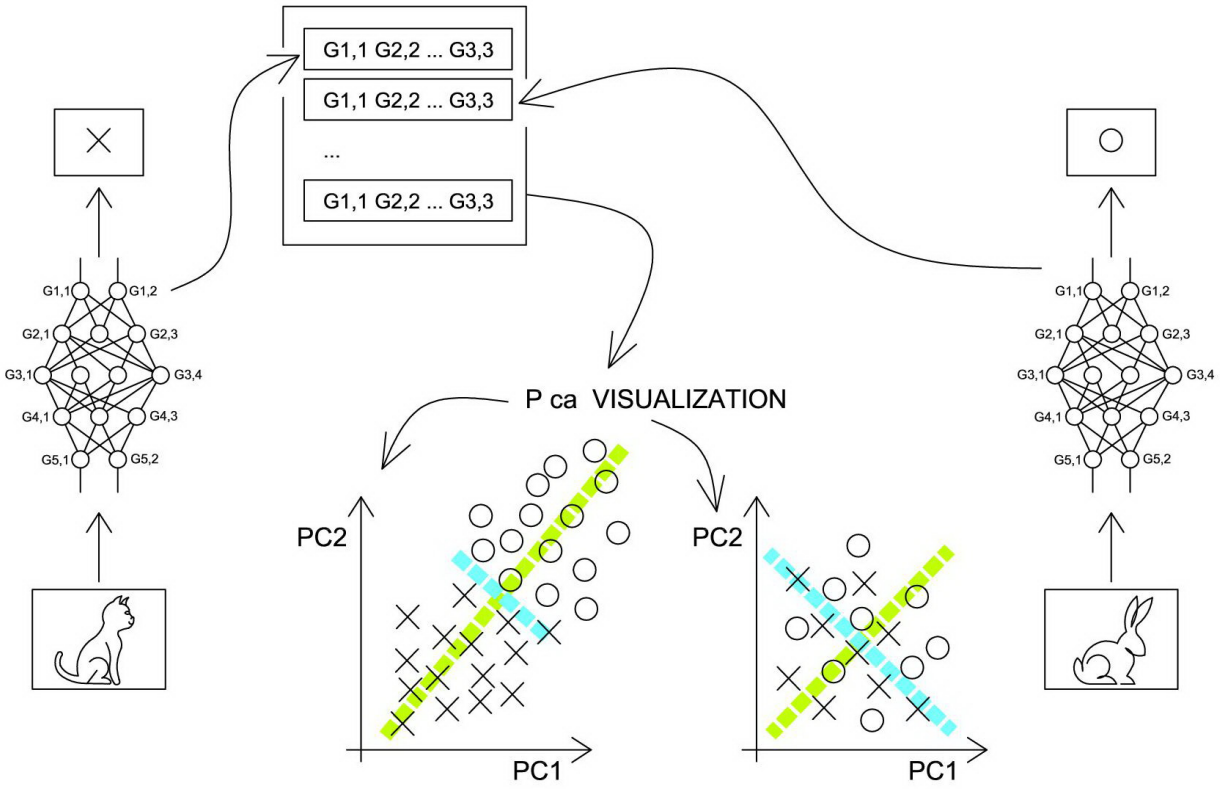

Figure 2.13: We compute the gradient of the output of the network wrt the weights for N samples. We store all the gradients in an NxM matrix (M), where N is the number of samples and M is the number of weights in the network. We then compute the eigenvalues for the matrix $M \cdot M^T$, the ratio between the biggest and smallest eigenvalue is our score. The intuition is that with a high score, there is a pattern in the gradients and thus, the network should have an easier time separating them, thus the scatter plot on the left belongs to good architecture, while the one on the right belongs to bad architecture.

$\mathbf{sim}\,(\mathbf{x}, \mathbf{y}) = (\mathbf{x} > 0) \cdot (\mathbf{y} > 0) + (\mathbf{x} = 0) \cdot (\mathbf{y} = 0)$	Similarity between activations.
$\mathbf{K}_{\mathbf{x},\mathbf{y}} = P - \sum_{i=1}^{P} sim\,(x_i, y_i)$	Matrix of distances between mini-batch samples' activations.
$\mathbf{score} = log\lvert\mathbf{K}\rvert$	The heuristic is the determinant of K.

Another simple heuristic that can help predict a network performance is the **loss** function after a small amount of training. There are many techniques that try to infer the final performance of the network from its loss in just a few steps. Finally we handcraft a heuristic specifically for this task, we could define this heuristic as the ratio between the positive gradients in the input image and the gradients of the image around the bounding boxes, with the gradient back-propagated from the sum of the outputs logits where the targets are supposed to be. The intuition is that if there is an object to detect at position h,w, we expect that backpropagating the logit responsible for confidence at h,w should cause the inputs image gradient to be positive at h,w. Efficient architectures have positive gradients only near the center of the bounding box while bad architectures have less precision in determining where the target is, and thus, having a bigger sum of gradients around the h,w point. Figure 2.14 illustrates how the heuristic is computed.

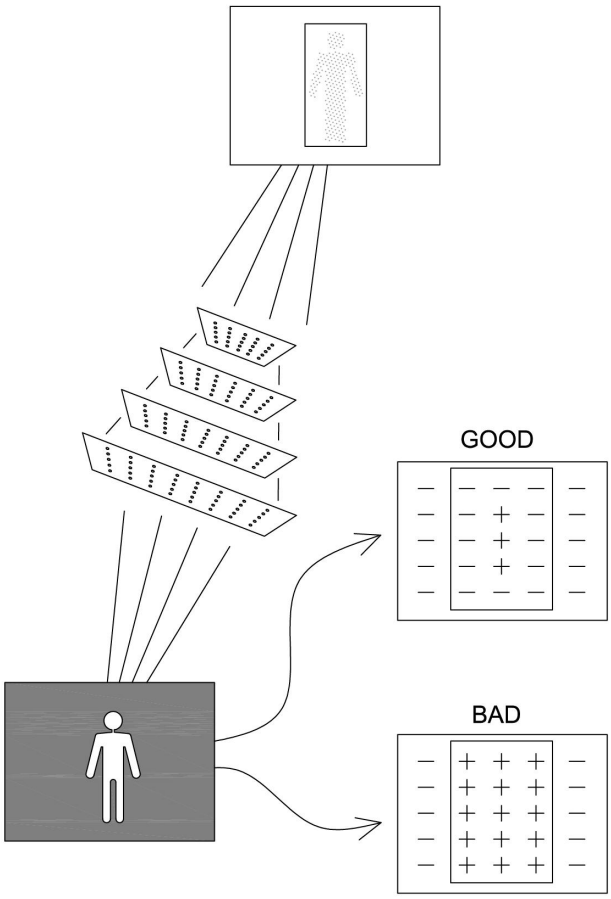

Figure 2.14: We compute the gradient of the sum of the output pixels of the neural network where a bounding box is annotated wrt the input image. Empirical evidence has shown that efficient architectures have less positive gradients near the pedestrian location (Figure 2.15). This seemed counterintuitive at first, but there is a strong correlation in the data that suggests that this is a good heuristic.

After having defined these heuristics we train 20 neural networks with 20 manually chosen configurations and gather both the heuristics computed without training the neural network and after training it for 2 epochs, moreover, we train the network for 13 additional epochs to compute the mean average precision score. After having collected the data we analyze it to find correlations. As shown in Figure 2.15, the heuristic based on rectified linear units is not very informative, while loss, NTK, and the input gradient score seem to have some significance. Figure 2.16 plots the final AP50 and the heuristic scores.

After this preliminary analysis of the data, we try to learn a linear model to predict the average precision using the heuristics (loss after one epoch, the NTK after one epoch and the input gradient score after one epoch). The model is trained on 17 architectures and tested on 3 novel unseen architectures. The prediction error is 0.06 AP50 on the train set and 0.08 on the test set, thus, it is useful to distinguish good architectures from bad ones because the AP50 score usually ranges from 0.10 to 0.45.

After having defined this score to quickly evaluate the architecture's configurations we use random search to evaluate more than 300 configurations. We then train the model with the best score with less than 100K parameters and the model with the best score with less than 10K parameters. We also notice that most of the models belong to the 'mlp2' class. After having trained these 2, models we notice that the AP50 obtained using the mlp2 model with 86K parameters wasn't up to expectations, while the yolophi model with 7K parameters achieved a good performance. For this reason, we start doubting the effectiveness of this method for predicting the AP50 of the mlp2 class of models, in fact, as we can see from Figure 2.16 the mlp2 model achieves high scores for the neural tangent kernel heuristic but does not achieve high AP50 values. This behavior was also found in other papers [59] [19], where they find out that having a low NTK score implies the model will have poor performances, but having a high NTK score does not imply having good performances.

For this reason, we re-define our scoring function using the NTK score and the loss score for pruning: if the NTK score or the loss score is worse than a pre-defined threshold (found empirically)

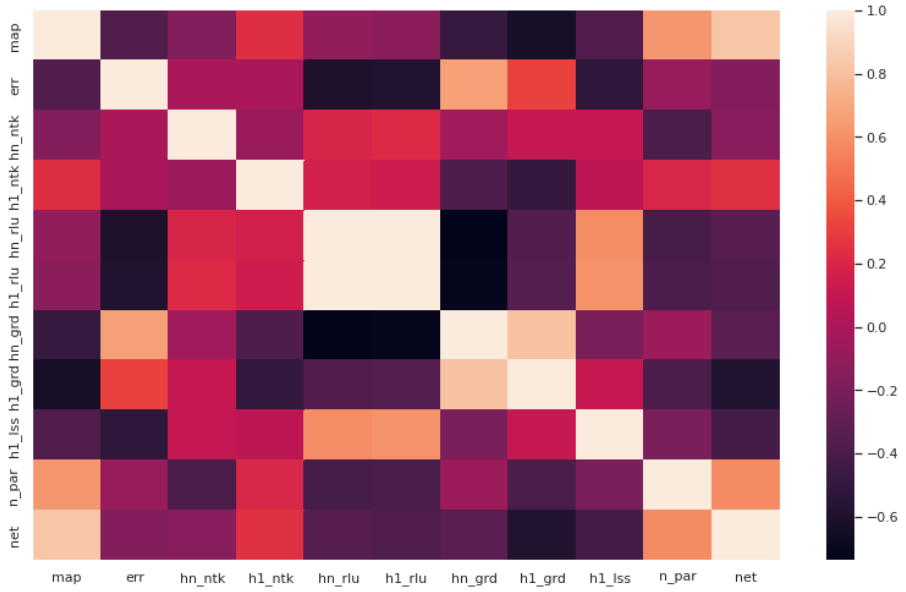

Figure 2.15: the first row shows the correlation between the mean average precision after 15 epochs of training and the heuristics computed after one epoch of training. As we can see the neural tangent kernel after one epoch of training ($h1\_ntk$) is positively correlated with the AP50, while the input gradient and the loss (respectively $h1\_grd$ and $h1\_lss$) are negatively correlated with the AP50.

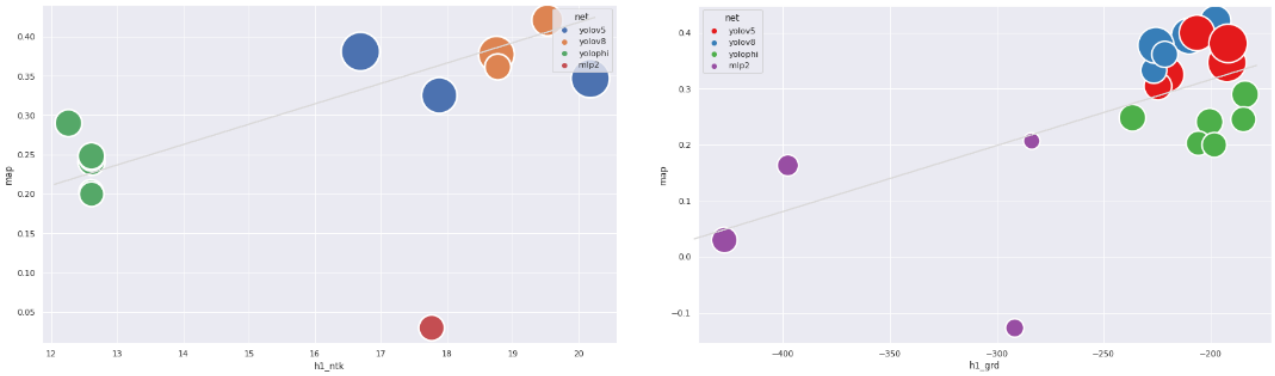

Figure 2.16: Correlation between NTK and the MaP (left). Correlation between **minus** grad input and AP50 (right). The NTK score often generates outliers, thus some models have been excluded from the plot

the score of the model is zero, otherwise it is directly proportional to the negative input gradient heuristic. We use random search to evaluate 300 new configurations with this new scoring mechanism and we train the 2 best scoring models, one with less than 100K parameters and one with less than 10K parameters.

To show the validity of this approach we randomly select 11 architectures and train them for 50 epochs to obtain a valid AP50 score. We afterward plot the predicted score and the real mean average precision as shown in figure 2.17. Finally, we report a table with the results obtained by training the top-scoring models for 120 epochs and compare them to the baseline.

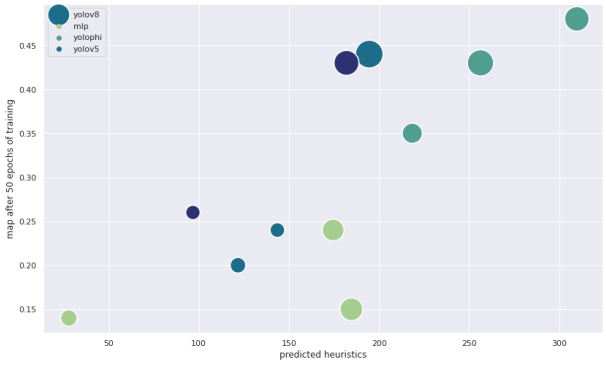

Figure 2.17: Heuristics score computed with one epoch of training and AP50 score computed with 50 epochs of training. The size of the dots is proportional to the number of parameters. As we can notice architectures based on PhiNet seem to outperform other types of architectures if we take into account the number of parameters.

Comparing metrics of different models				
Dataset	Architecture	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	baseline 104K	3.61	0.51	0.25
SynthMOT	baseline 104K	2.69	0.64	0.28
Streets23	baseline 104K	0.21	0.54	0.23
average	baseline 104K	2.17	0.56	**0.25**
MOT17	phiyolo 77K	3.65	0.47	0.22
SynthMOT	phiyolo 77K	2.62	0.62	0.26
Streets23	phiyolo 77K	0.21	0.70	0.27
average	**phiyolo 77K**	2.16	**0.59**	**0.25**
MOT17	phiyolo 7K	3.05	0.42	0.14
SynthMOT	phiyolo 7K	3.14	0.59	0.23
Streets23	phiyolo 7K	0.26	0.63	0.24
average	phiyolo 7K	**2.15**	0.55	0.20

As we can see from Table 2.3.1, even though the architecture we found has more than 30K fewer parameters, its performances are close to the baseline architecture if not even better. Moreover, even the simpler model with just 7K parameters can achieve satisfying scores, especially when compared with other architectures with less than 20K parameters which can reach an AP_{50} of at most 0.30.

We now report the hyper-parameters used in the proposed architectures, where *width* and *height* are parameters regulating the neck while *alpha*, *t*0, *beta*, squeeze-expand ($SqEx$) and convolution2d ($cv2d$) are parameters regulating the PhiNet backbone.

Models' architecture's configurations							
Model architecture	width	depth	alpha	t0	beta	SqEx	cv2d
baseline: phiyolo	0.33	0.5	0.35	7	1	False	True
phiyolo 7K: phiyolo	0.15	0.1	0.05	8	0.82	False	True
phiyolo 77K: phiyolo	0.2	0.55	0.05	9	1.12	False	True

2.3.2 Models Summaries

This section will report more details about each model extending the architecture design with some visual information and bench-markings. In addition, we report other experiments with simpler architectures that have been tried; more specifically, we discuss the mlp2 architecture because of its importance in the section dedicated to quantization and some PhiNet models that do not have a neck because this makes them lighter.

To quantify the number of computations required for a forward pass we use the **torchinfo** library,

which returns how many multiply/add operations have been done in a forward pass. To compute the amount of RAM used to keep all the needed tensors in memory we need to take into account skip connections and the convolutional layer with the most activations. For example, in the phiyolo architecture, we need to keep three intermediate activation layers from PhiNet that will later be used as skip connections. To compute this value we move the model to the GPU and create a thread to continuously check the used amount of GPU RAM and return the maximum value. This should serve as an upper bound because the order of the operation in the model could be re-organized to save some memory (but it would require rewriting most of the yolo's implementation), moreover, other processes that are using RAM could introduce noise in the estimation, luckily we observe that it seems not to be a problem, but repeat the estimation multiple times just in case.

PhiYolo 104K This is the model used in the baseline, as shown in Figure X the backbone contains 104445 parameters. From the MACC count emerges the fact that this model is pretty heavy (many times slower than its similar counterpart phiyolo 77K).

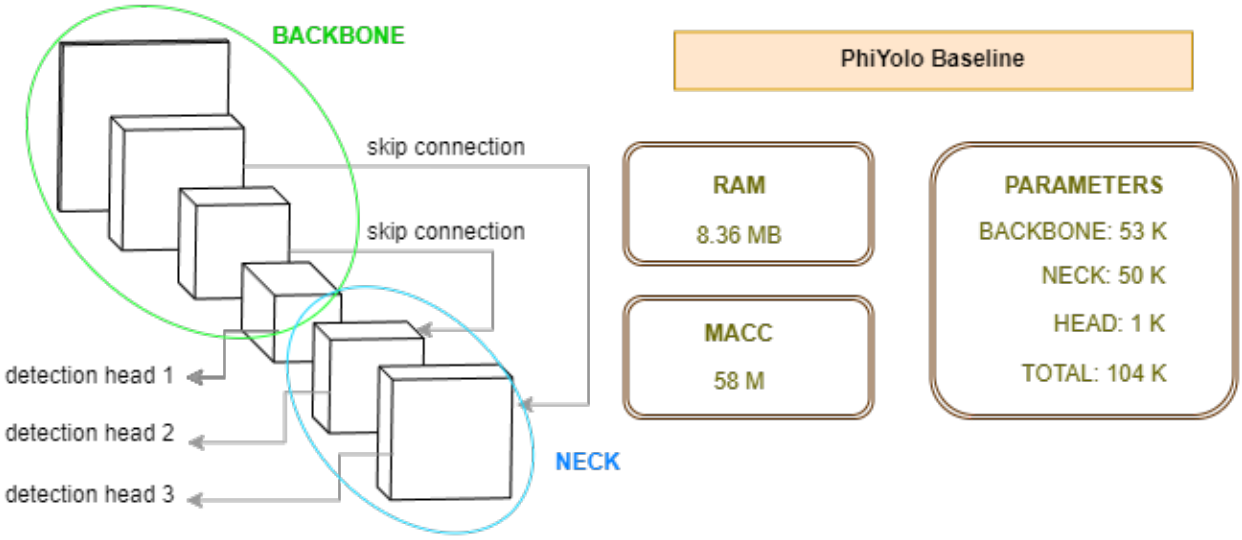

Figure 2.18: Visualization of the PhiYolo 104K architecture. This architecture achieves an average MaP of 0.25 (without policy).

PhiYolo 77K. This architecture was found using heuristics and is a good compromise between good results and low computational power.

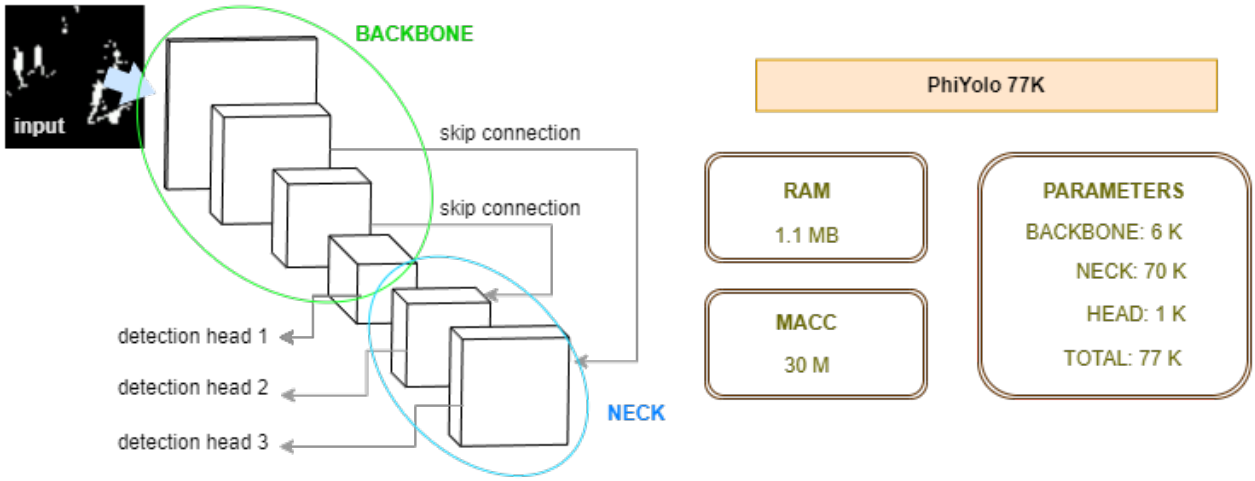

Figure 2.19: Visualization of the PhiYolo 77K architecture. This architecture achieves an average MaP of 0.25 (without policy).

PhiYolo 7K. This is the smallest architecture, the PhiNet backbone is extremely reduced while most of the parameters are in the neck (the convolutions in the neck have no bias too, thus they are quick when the input contains a lot of zeros).

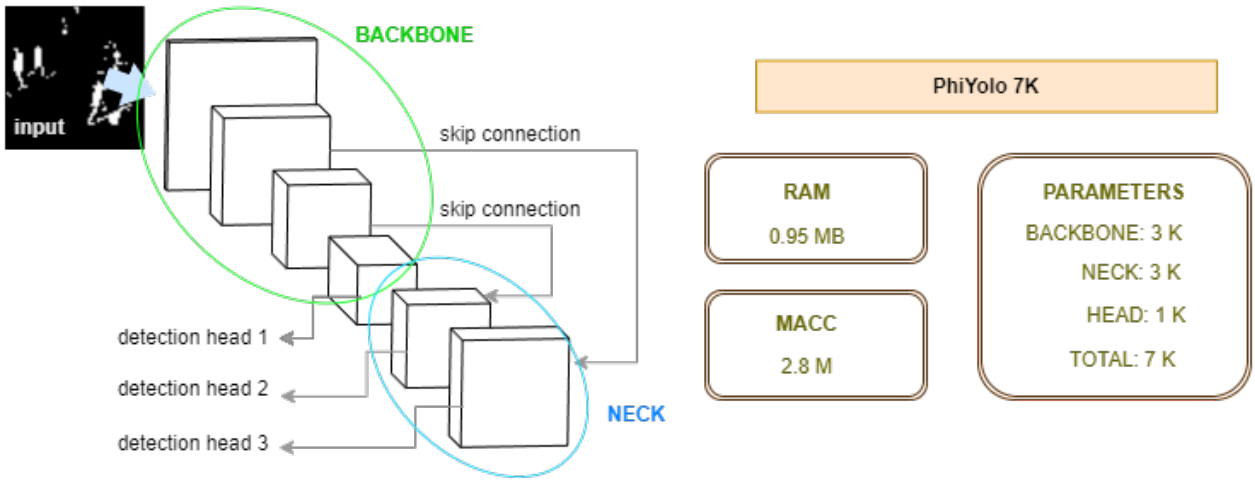

Figure 2.20: Visualization of the PhiYolo 7K architecture. This architecture achieves an average MaP of 0.20 (without policy).

Phismall 1. Simplest architecture: PhiNet using a configuration from the original paper [67] with a final convolution for the predictions (the detection head).

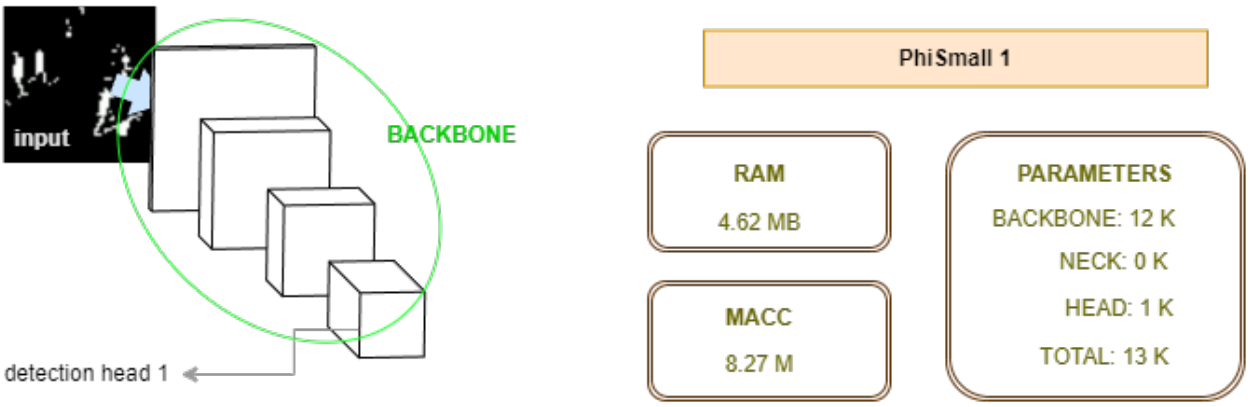

Figure 2.21: Visualization of the PhiSmall 1 architecture. This architecture achieves an average MaP of 0.12 (without policy).

Phismall 2. Extends **phismall** 1 by having a second detection head, the input for this head is an intermediate layer of **PhiNet**.

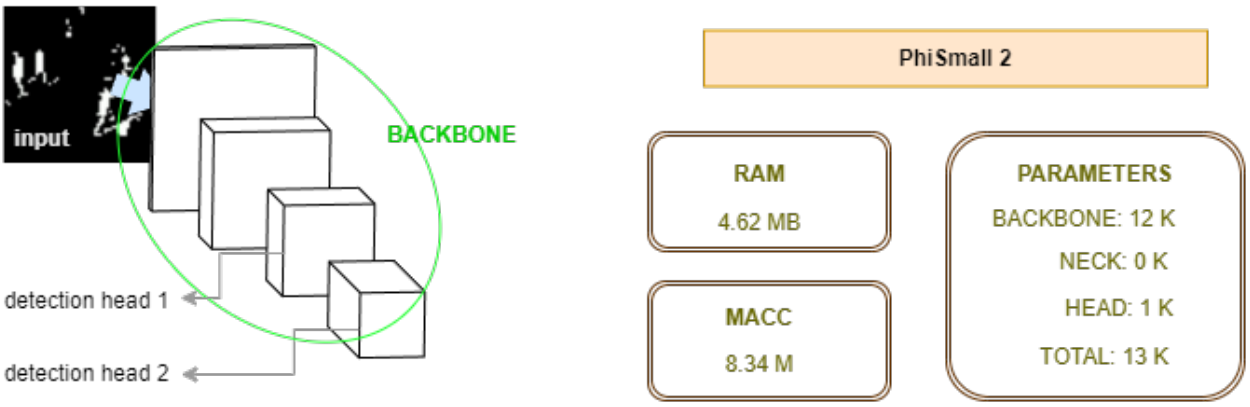

Figure 2.22: Visualization of the PhiSmall 2 architecture. This architecture achieves an average MaP of 0.09 (without policy).

Phismall 3. Extends **phismall** 1 by up-scaling the output of phinet by 4 and concatenating it with a resized motion map that serves as a skip connection. The output is furtherly processed by a small CNN before being used in the detection head.

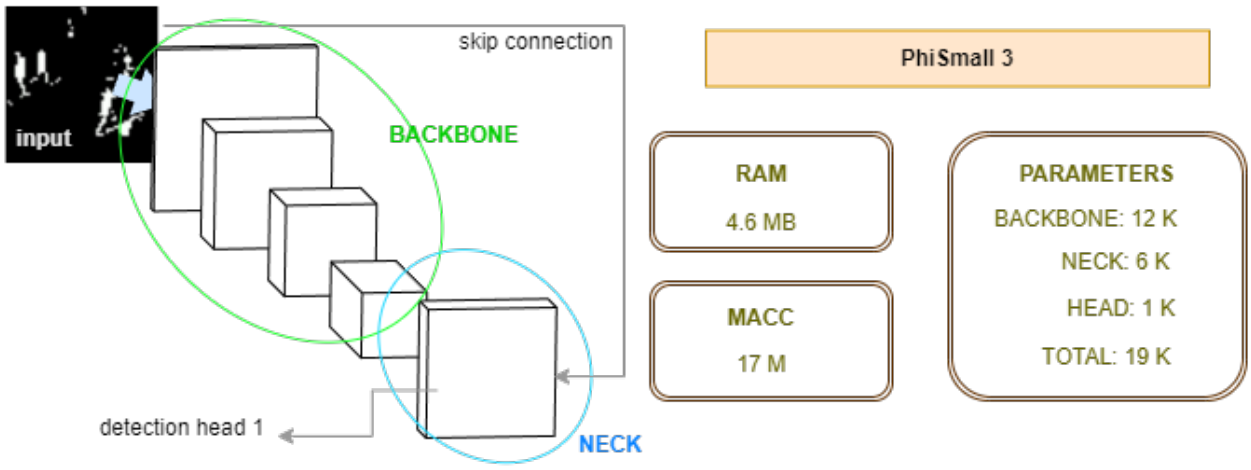

Figure 2.23: Visualization of the PhiSmall 3 architecture. This architecture achieves an average MaP of 0.08 (without policy).

MLP2. This model leverages both FFNNs and CNNs to process motion maps. The linear layers are used by firstly resizing the input channel to a 5x5 feature map then passing the unrolled input to the linear layer and reconstructing the output as a feature map. This model isn't particularly efficient but has been included because it works with XNOR-Nets [72] as well, while yolo8 or phiyolo don't.

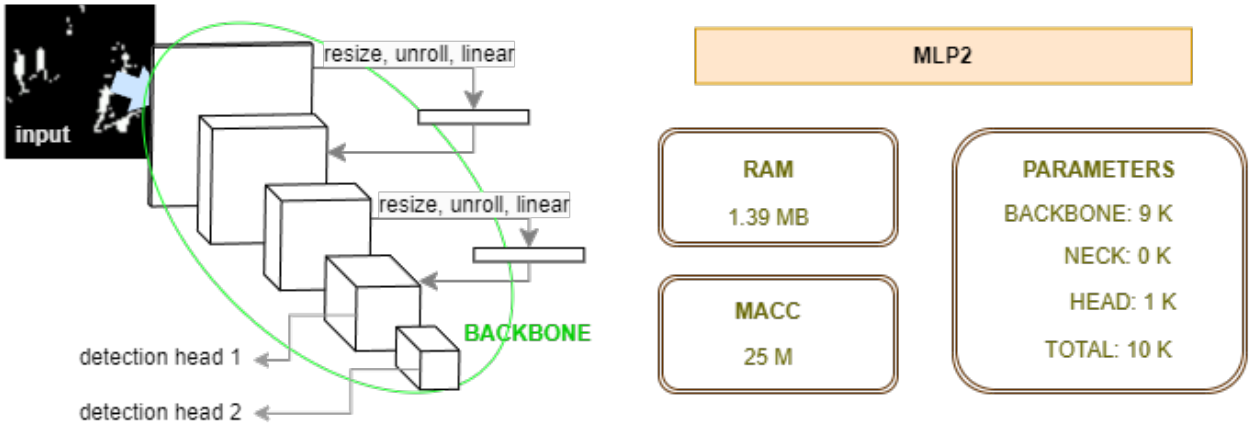

Figure 2.24: Visualization of the MLP2 architecture. This architecture achieves an average MaP of 0.13 (without policy).

Conclusions. Curiously, even though yolov5/yolov8 models have more parameters, they seem to be faster than the baseline. The best model choice to run operations at the edge is the **phiyolo 7K** model because has both the lowest RAM requirements and MACC requirements with competitive performances wrt. the other models. Microcontrollers with 1MB of RAM could run this model and the price for one of these devices in 2023 is around 10$ (a lot lower if they are bought in stocks). Between the other models, probably phismall1 is second best when trying to keep a lower amount of MACC, phiyolo 77K is the best trade-off between performances and hardware requirements, while yolo8 achieves the highest performances but with a high number of MACC operations and parameters. These later models could be run on a raspberry which has at least 1GM of RAM and is priced at around 40$. Strangely, the extensions to phismall 1 do not improve performances. We report below a full table with the scores obtained by each model.

Comparing Models		
Architecture	Dataset	MaP
yolophi (baseline) parameters: 104K MACC: 58M RAM: 8.36MB	MOT17	0.25
	SynthMOT	0.28
	Streets23	0.23
yolophi 77K parameters: 77K MACC: 30M RAM: 1.11MB	MOT17	0.22
	SynthMOT	0.26
	Streets23	0.27
yolophi 7K parameters: 7K MACC: **2.8M** RAM: **0.95MB**	MOT17	0.14
	SynthMOT	0.23
	Streets23	0.24
phismall 1 parameters: 13.1K MACC: 8.27M RAM: 4.62MB	MOT17	0.09
	SynthMOT	0.14
	Streets23	0.14
phismall 2 parameters: 13.2K MACC: 8.34M RAM: 4.62MB	MOT17	0.11
	SynthMOT	0.20
	Streets23	0.06
phismall 3 parameters: 19K MACC: 17M RAM: 4.62MB	MOT17	0.00
	SynthMOT	0.20
	Streets23	0.05
mlp2 parameters: 10K MACC: 25M RAM: 1.39MB	MOT17	0.12
	SynthMOT	0.19
	Streets23	0.10
yolov8 parameters: 330K MACC: 40M RAM: 1.79MB	MOT17	0.26
	SynthMOT	0.29
	Streets23	0.28

2.4 Quantizing Neural Networks

Neural network weights are usually stored as floating point numbers with 32 bits. Quantizing deep learning models usually consists in reducing the number of bits used for the activations and weights. For example, a very common practice to save RAM is to train and save models with 32 bits floats, and, at inference time, convert the weights to 16 bits floats. This approach slightly reduces the

precision of the network, but, halves the memory requirements of the network. We aim at even more efficient quantization, more specifically 8 bits operations are much faster, and specific libraries have been developed to carry out these operations [1]. A typical quantization scheme to reduce the number of bits to M is used in [96, 8] and is summarized as follows:

$$scale = (vmax - vmin)/2^M$$

$$zero - point = round(min(max(-vmin/scale, 0), 2^M))$$

$$output = round(x/scale + zero - point)$$

The first approaches [96, 9] try to quantize the weights of the neural network after training, while more recent methods try to train models that use quantized weights by making the quantization step differentiable [42].

We try quantization of different libraries and adopt **micronet** [104], which implements the Dorefa-Net algorithm [105] which quantizes the network weights between -1 and +1 and adds a quantization step that should help to preserve the gradient. The original paper proposes to quantize weights to one bit and activations to two bits, but the micronet implementation works with minimum of 2 bits.

We train the baseline model, phiyolo 77K and phiyolo 77K with 8 bit quantization and find out that the final performances are close to the non-quantized models with an average decrease in AP_{50} of 0.03, but with the advantage of reducing the required memory by 4 times.

PhiYolo 7K Quantization Performances		
quantization bits	average AP50	average Map
non-quantized	0.55	0.20
8	0.50	0.18
4	0.41	0.13
2	0.11	0.02
1	0.00	0.00

XNOR-Nets The micronet library supports other types of quantizations including the XNOR-Nets [72]. This type of quantization approximates convolutions with binary convolutions and processes binary inputs, making it perfect for our problem.

The convolution's weights are approximated with the formula:

$$I \star W \approx (I \oplus B)\alpha$$

where the leftmost part of the equation refers to the "traditional" convolution, while the rightmost refers to the approximation; with I being the binary input, $B \in -1, +1\{c \times w \times h\}$ the kernel and α the rescaling factor.

The advantage of using this type of kernel is that the operations are just additions and subtractions, making it way faster to compute. There is a close form solution to compute the best $B = sign(W)$ and $\alpha = W.abs().mean()$, and with sub-gradients, we can make the quantization differentiable. Figure 2.25 shows how this approach can be used to simulate quantization in training.

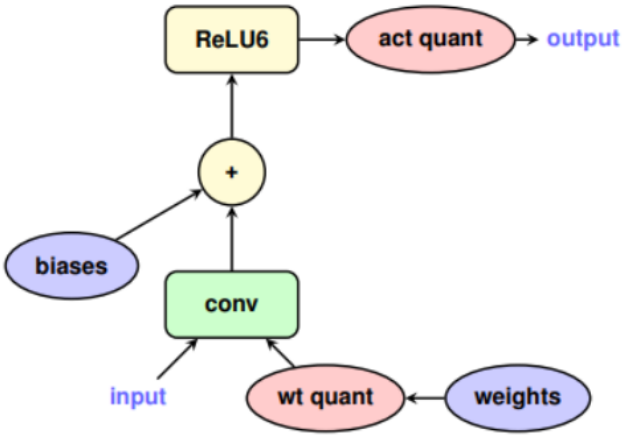

Figure 2.25: The quantization of weights and activations is performed online. Credits for the image to [42].

We test this type of quantization on our architectures (we slightly need to modify the source code to use activation different from ReLU). Results show that only the mlp2 architecture seems to be able to learn the pattern, we think there could be two reasons behind this behavior: some convolution of the mlp2 have the bias parameter, while the other architectures do not; the linear layers (which are not quantized) make the difference. Even if the linear layers are not quantized, we could use the 8-bit quantization to reduce them, moreover, they are so small that they don't really introduce a large overhead in the network.

Binary Quantization Performances			
architecture	average AP50	average Map	non-quantized AP50
phiyolo 7K	0.00	0.00	0.55
mlp2	**0.25**	**0.12**	0.27
yolo8	0.00	0.00	**0.62**

Conclusions. We show that quantization is very effective at reducing the amount of memory requirement of the network without compromising the results too much. We also try binary convolutions networks which would be perfect for our problem and we confirm that they are effective approaches but which do not work for every type of architecture.

2.5 Optimizing Simulator's Policy

This section explains how we optimized the simulator's parameters. The smart vision sensor simulator is a bottleneck that greatly reduces the amount of information contained in the grayscale to produce a motion map that will be processed by the neural network. Optimizing these parameters can help generate higher-quality motion maps that will improve the final results.

Optimizing the sensor's parameters isn't a trivial task since the simulator function is non-differentiable and the parameters must be integers. This section firstly introduces the simulator's parameters and how they affect the svs algorithm and secondly explains the different techniques we adopted to optimize the parameters.

2.5.1 Simulator's Parameters

There are three parameters in the simulator [109]:

- d_close: which regulates how quickly changes are removed from the background
- d_open: which regulates how quickly changes are inserted in the background
- d_hot: which is the threshold used to decide if there is the motion or not

The svs algorithm behaves like a background subtraction algorithm. For each pixel in the image space 2 values are kept in memory: the high_threshold and the low_threshold, when a new frame

needs to be processed the algorithms detect motion if either the frame's pixel value is higher than $high\_threshold + d\_hot$ or lower than $low\_threshold - d\_hot$. Afterward, the high_threshold is updated by adding d_open if the pixel value was higher or subtracting d_close if the pixel value was lower. The opposite is valid for the low_threshold.

2.5.2 Proximal Gradient Method

The first idea for tackling the optimization problem is to transform the simulator function into a differentiable one, this could be achieved with the sub-gradient method or by approximating the non-differentiable function with a differentiable one. The first method works by defining a gradient in the points of the differentiable function where it is not defined (or equal to zero), while the second method is just an approximation of the original function. Relative to the sensor, the non-differentiable step resides in the disequality used to decide whether there is motion or not: $motion\_h = pixel > high\_thresh + d\_hot$; Figure 2.26 shows how we could make this step differentiable.

Differentiable simulator.

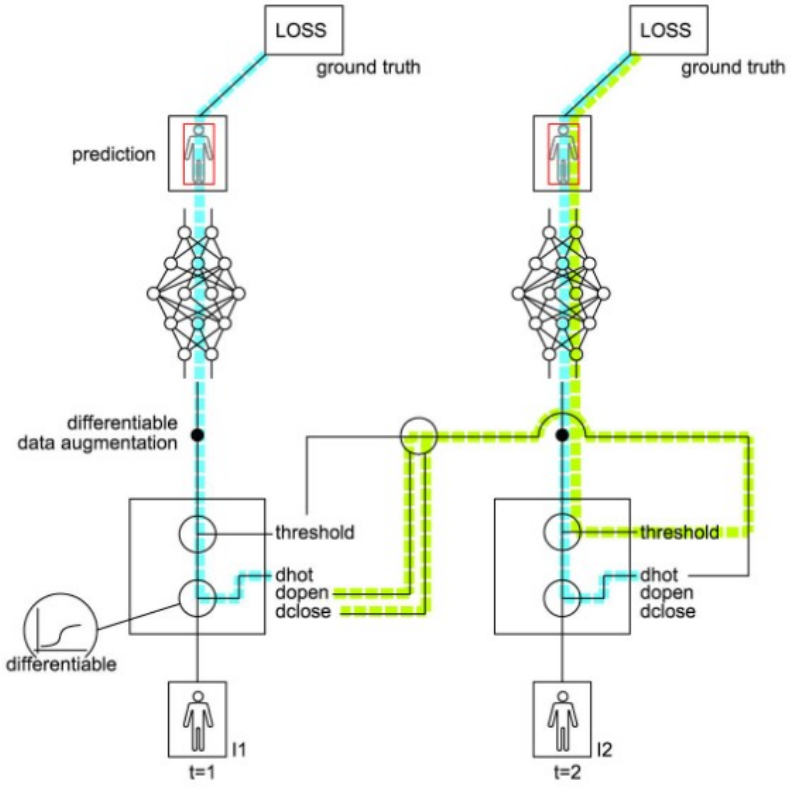

Figure 2.26: The propagation of the gradient through the network. The leftmost part shows the forward/backward pass at time t, while the rightmost part shows the forward/backward pass at time t+1. The (green) path shows the gradient back-propagated through time for d_open and d_close. d_hot directly modifies the motion map thus the back-propagation of the gradient is straightforward (cyan). d_close and d_open do not modify the input directly, they instead modify the threshold which will influence the motion map at the next frames (green). For this reason, we need to keep the back-propagation graph in memory creating risks for memory leaks.

Differentiable simulator.

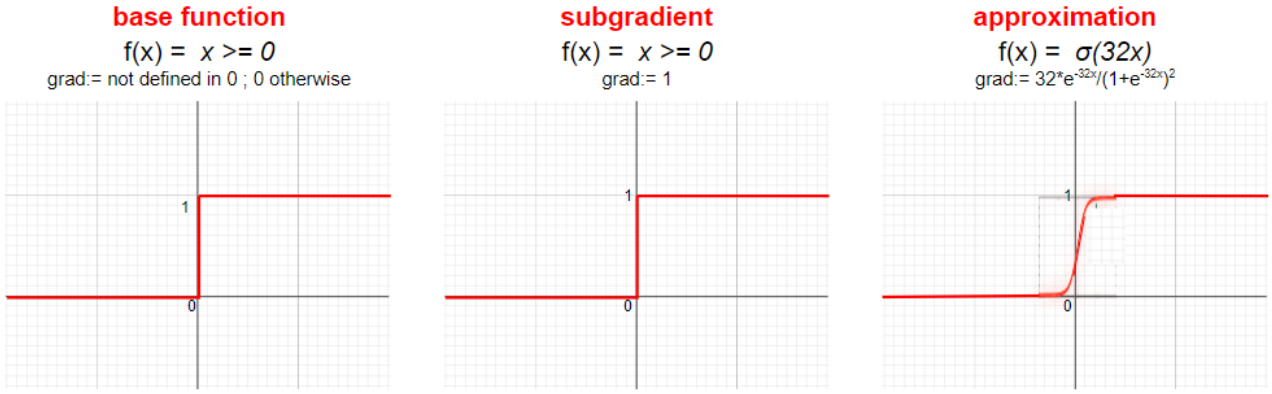

Figure 2.27: Original function (left). Using the sub-gradient method to define a custom gradient (center). Approximating the step-function with the sigmoid function

The approach we decide to take is to approximate the step function with a sigmoid function 2.27; in practice, it should not constitute a big difference with the sub-gradient method since the approximation is almost perfectly equal to the step function (we use $sigmoid(500 * x)$ as a surrogate function), while the gradient will be later processed thus giving it magnitude less importance with respect to its sign. Another point we need to change in the simulator is the erosion: it's implemented using the OpenCV library. Even though this function is differentiable, OpenCV [15] functions do not support torch tensors as inputs and we need to use tensors because they support automatic gradient computation. Since erosion is simply erasing positive values that lie next to zero values (at least for the kernel we used), our implementation is summarized as follows: we reverse the motion map so that zeros become ones and ones become zeroes, we apply a 2D convolution with a 3x3 kernel made of ones, we subtract this result to the original motion map and we finally apply a ReLU function, in this manner if a positive pixel is next to a negative one, its value would become zero thus implementing a differentiable erosion.

Differentiable augmentations. After having created a differentiable system we start training the model on our data. Unfortunately, the model does not converge. An important change that was required to use the differentiable simulator is to give up on the spatial augmentations since that step is not differentiable. Since data augmentation is important to properly train the model, we search for available implementations of data augmentations that are differentiable too. We find Kornia [84], a library that can implement data augmentations to keep the gradients alive, and we use it to apply a transformation to the images produced by the simulator, while we had to update the labels manually. The procedure we followed here was the following: we randomly select some augmentations (eg. rotation) and their module (range of values), and we then use Kornia to apply the amount of rotation we wanted on the image using the kornia.functional module, then we create the affine transformation matrix to apply the augmentation to the annotated bounding box coordinates as well.

Torch lightning. While we were implementing the above augmentations we incurred a rather challenging problem. We managed to solve it but, since it took quite some time, we will spend some lines describing it. Torch Lighting [29] is a library created to facilitate distributed neural network development. It offers the possibility to create a model containing methods that specify how to load the data how to evaluate the model and how to test it. The library then proceeds to automatize the boilerplate parts of the code simplifying the work for the developer. The issue happened inside the method responsible for generating the data loaders and can be summarized by saying that some pointers inside the child processes of the data loader were returned to the main process overwriting the old pointer and that should not happen. The solution was to first load all the data inside the main process before forking into the child processes; in this way, it seems that the child processes aren't modifying the original data anymore (it was a very weird problem).

Proximal gradient. The parameters of the simulator are supposed to be an integer, thus we have to impose this constraint. The proximal gradient method is an optimization algorithm that can allow us to impose this constraint by projecting the "smoothed" values into the constrained ones. We can summarize the proximal gradient method as follows. It is an iterative procedure that:

- computes the gradient of the differentiable function (eg. loss.backward())
- updates the parameters (eg. $-lr * gradient$)
- apply the constraints (eg. round to integer, which is $prox_g(x) = argmin_y g(y) + (1/2alpha)||y - x||^2$ using alphas with great values)

Using this method does not bring to any parameter configuration that converges, because with a step size too small, the parameters of the simulator are stuck while, with a higher learning rate, the parameters do not converge to a good solution. The intuition behind the reason why this happens in the latter case is the following: the neural network inputs need to be stable to allow the learning of the detector's parameters, although, updating the simulator's parameters frequently introduces too much noise and variations to allow the optimization of the system as shown in Figure 2.28 and Figure 2.29.

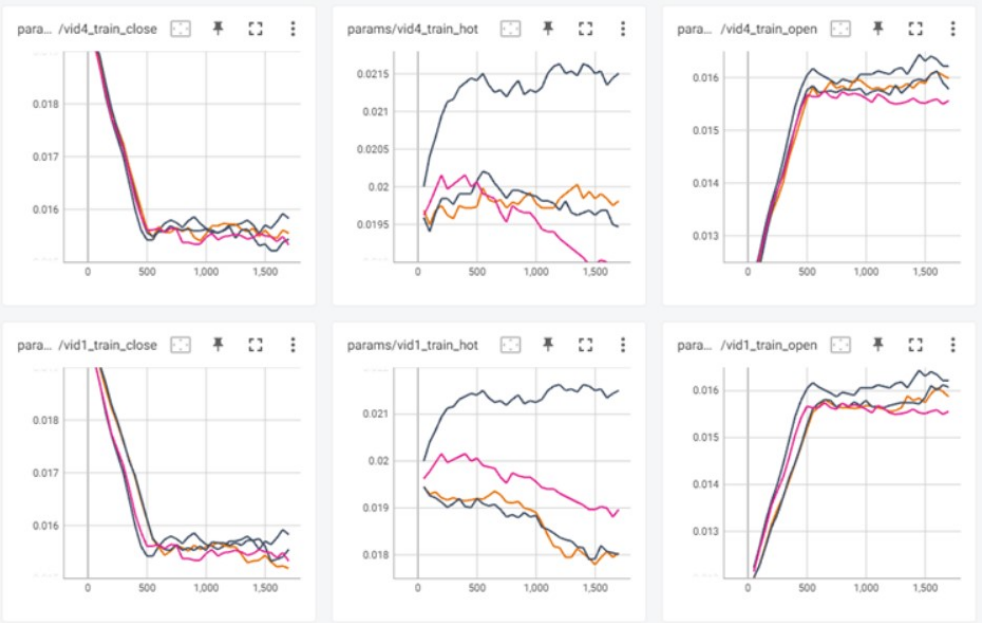

Figure 2.28: Even by training the system on a single video at the time (each row is a different video), we can see that the d_hot parameters are very unstable (the columns are: d_close, d_hot, d_open). Training the system on multiple videos at the same time worsens the results even more. Note: the parameters are expressed divided by 255 because the input images are normalized between [0,1] when entering the simulator, thus a parameter with a value of 0.016 corresponds with a parameter with a value of ca. 4 in the uint8 scale.

A solution would be to temporarily allow the simulator to use floats as parameters and quantize them in order to have integer parameters for the simulator once the training has been completed. Even though this approach seems promising two problems emerge: 1st the quantization step is an approximation and it's hard to decide how to quantize the parameters due to other constraints like the fact that d_hot should be greater than d_open and d_close (it is not mandatory, but, when this is

Figure 2.29: As we can see from the losses, the network is not able to learn when using the proximal gradient algorithm.

not respected, we empirically noticed that the output of the simulator starts to alternating output full of noise with clean frames, which is not an ideal condition). The second and more relevant problem with this approach is that we notice that the parameters do not converge because it seems that each video has its own set of parameters. Note post-experiments: rather than each video has its own set of parameters, computing the best parameters for a small number of samples, generates gradients that are too much unstable to converge.

Multiple simulators. We try to address the above problem in the following way: we create a simulator object for each video in the training set, in this way each video will have its own set of parameters. We then concatenate the output of different simulators in a unique batch and pass it to the neural network, allowing us to have a single model to predict bounding boxes over multiple videos. Unfortunately, even this approach does not work as shown in Figure 2.30:

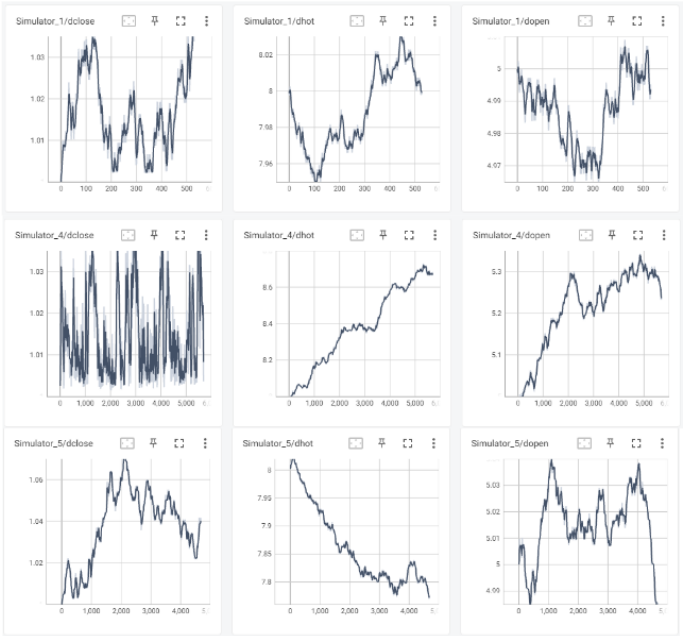

Figure 2.30: The parameters of the simulator seems not to converge using the differentiable approach. The learning rate is small enough (in fact we can see that d_close does not change much), but the direction of the gradient in the simulator is not stable enough to train the network.

Conclusions for the PG approach. Since this method does not bring the expected results, we decide to try other approaches. There could be many reasons why this approach fails, for example, with the standard training procedure, we could shuffle the dataset, showing the network frames from different videos in different orders, while, with the differentiable approach, we need the simulator to see the correct sequence of frames. Introducing multiple simulators helps solve this approach but the frame order must not be shuffled in any case.

Another major problem is that d_open and d_close do not directly update the motion map but update the thresholds, which will be used in the next frames. This fact can be solved by retaining the gradient in the previous iteration but a known issue with propagating gradient through time is that it could disappear or explode (although this seems not to happen in our case).

2.5.3 Optimizing the Simulator By Optimizing a Score

Our final objective is to optimize the mean average precision of the joint system simulator + neural network. As we have seen in the previous section (and can intuitively understand) the MaP score is negatively correlated with the total loss of the neural network thus, instead of maximizing the MaP on a validation set we could optimize the computational cost by minimizing the loss on the train set. It is true that we could generally incur into the problem of over-fitting, but, o this specific problem setting this does not happen for two reasons: the models do not have enough parameters to memorize the whole training set and the use of data augmentations united with data sets from basically different domains plus the generalization of the problem introduced by the svs algorithm causes the fact that minimizing the loss on the training set is directly proportional with maximizing the MaP on the test set (we also observe this fact empirically).

Learning known operators. At this point, we aim at optimizing the simulator's parameters in order to minimize the training loss. There are many black-box algorithms that could be used to obtain these results but many of them rely on optimizing a fitness function that should be simple to compute. Unfortunately computing the loss of the neural network in a "stable" way (where stable means useful for getting results that help to compare different methods or sets of parameters: "using it as a fitness function") is very expensive. For this reason, we try to create a score computed only from the motion map that should be correlated with the final loss. The intuition is the following: if we notice that maximizing the number of white pixels in the motion map in the locations of the bounding boxes is correlated with a lower neural network loss, we could optimize the simulator parameters by maximizing that score (easy to compute) and know it will improve the final MaP.

The idea introduced above can also be justified by the work of Maier et al. [58]. They show that if we have a neural network that includes some "known operators" (in our case the simulator is the known operator) if that operator does some operations that are useful to the network, we can define an upper bound on the error of the neural network. This doesn't really apply to our case since we have multiple layers of neurons concatenated between each other, but, at least, the fact that if we find an operation of the simulator that is useful to the neural network, this should help to learn the task of object detection; and how do we define this operation? This operation is defined by looking at a lot of data and finding if there is a pattern between some hand-engineered operations and the final loss of the neural network.

Handcrafted scores. This paragraph explains some handcrafted scores to judge the usefulness of a motion map while the next one analyzes whether these scores are useful to help train the neural network. All these scores are computed using the motion map and the ground truth annotations of the bounding boxes.

List of possible scores:

- precision: the ratio between the number of white pixels inside the bounding boxes and the total number of white pixels in the motion map.
- recall: the ratio between the number of white pixels inside the bounding boxes and the number of pixels in the bounding boxes.

- F1: $precision * recall * 2/(precision + recall)$

- stability: minus the absolute value of the subtraction between the number of white pixels outside the bounding in the previous frame and the number of white pixels outside the bounding in the current frame

- connected components: minus the absolute value of the subtraction between the number of targets not occluded by each other and the number of connected components in the motion map inside the bounding boxes (where a connected component is a cluster of pixels with the same value)

The intuition behind each score is the following: a high value for the precision score means that we see motion where there is effectively a target; a high value for the recall means that we see a lot of motion where the target is; a high stability score means that there is no "flickering" of pixels during time; a high connected components score means that the target is a unique object and are not fragmented (example of fragmented: legs and arms are white but the torso is black).

After having set these scores, we try to manually change the parameters of the simulator to see how the scores would change, unfortunately, we notice that the changes often improve one score while worsening the others. For this reason, we collect and analyze more data.

Searching for a significative score function. To have a better understanding of the scores, we collect 5000 samples in the following way: we selected a random video with a random frame rate and some random parameters for the simulator; we simulate the motion map from the grey-scale video and compute the scores wrt. the ground truth; afterward we load a pre trained neural network and try to over-fit it on this brief video sequence. once the training is completed (15 epochs), we save the final loss, the scores, and the other information about the experiment. We then proceed to analyze the collected data using a notebook, looking for correlations between the scores and the final loss.

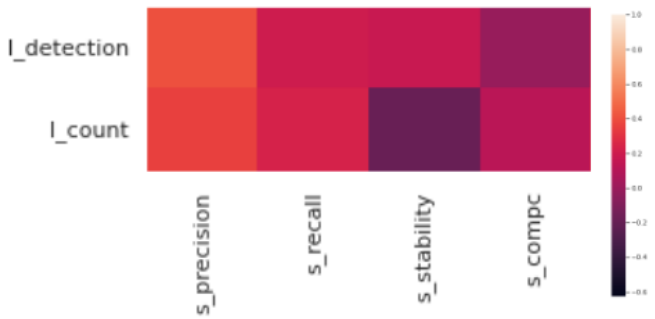

Figure 2.31: Correlation between the losses and the scores.

Even though it could seem that the scores could be a good predictor for the loss (Figure 2.31) when we try to learn a model to predict the final loss from them, its precision is not good enough to use it as a fitness function, in fact, the model is only capable of distinguishing bad parameters from good ones and is not useful at all to distinguish between near optimal parameters. This is further proven by plotting some of the data points as shown in Figure 2.32.

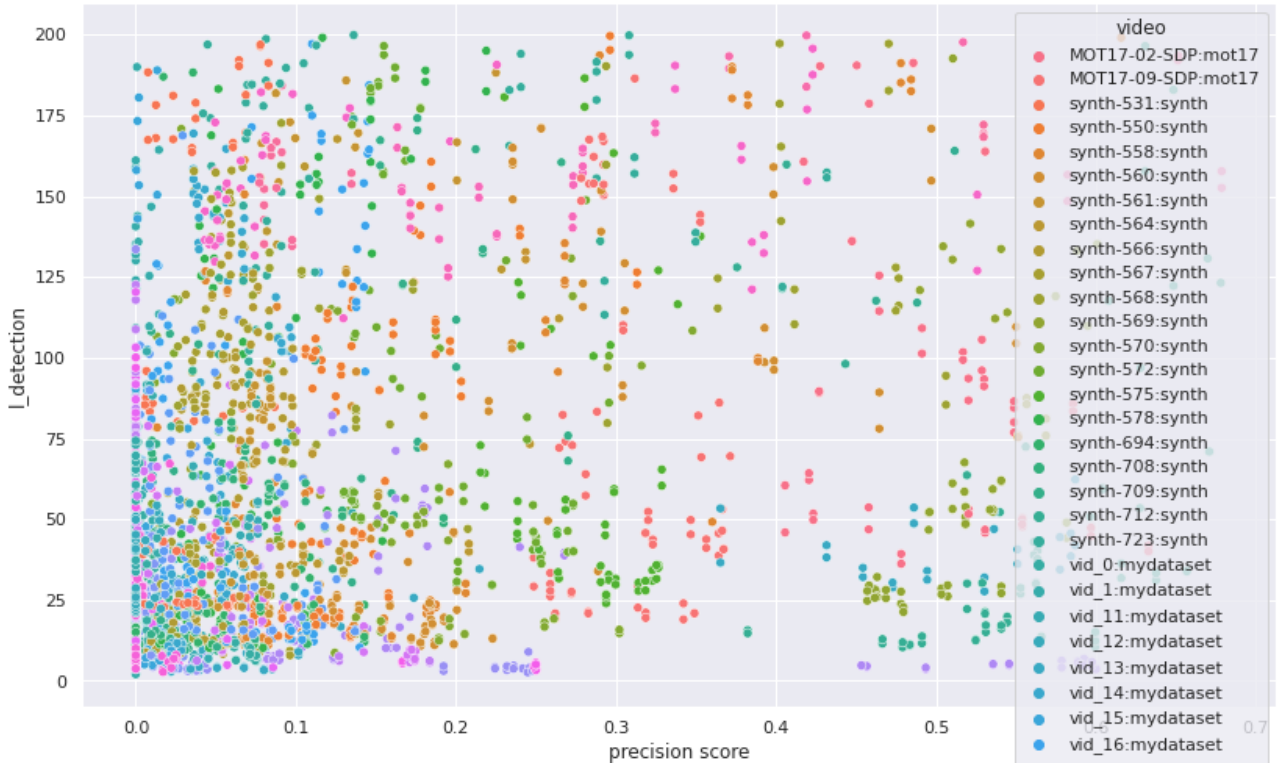

Figure 2.32: We plot the precision score alongside the detection loss. We also distinguish each video with a different color, but no pattern seems to emerge.

Score function conclusion. We couldn't find a satisfying enough score function, thus, the next section explains another approach that aims at optimizing the loss function of the neural network directly. If we had to continue trying to search for an easy to compute the loss function some steps would be: to increase the training time of the neural network to ensure that the final loss is significant for predicting the final MaP and try other types of scores.

2.5.4 Learning How to Predict The MaP the Parameters

Other than finding the optimal simulator's parameters, we want to learn a strategy to optimize the simulator's parameters for unseen videos online. The idea is simple, we want to train a model capable of predicting the expected **mean average precision** (MaP) given the **simulator's parameters**. Since the MaP will be different for each video, we include some **heuristics** extracted from the last motion map (eg. the number of white pixels), the model should be able to distinguish videos using this information and, hopefully, learn a generalization that can work with unseen sequences.

We can train this model, which will be referred to as **policy**, by sampling random intervals of sequences, simulating them with some parameters, training phiyolo on the first part of the sequence, and computing the MaP on the latter part. Once we have the triplet $(heuristics, parameters, map)$ we can train a second neural network to learn $f(heuristics, parameters) \rightarrow MaP$. Initially, we believe it would be simple for a neural network to learn the MaP landscape function because we hypothesize the function to be convex: with the global optimum corresponding to the best configuration of parameters and a decrease in MaP that is proportional to the distance from the best configuration; unfortunately, it was not as simple as that. Once we have trained our policy we can use it at inference time to set the best simulator's parameters for the current video by predicting the configuration that will have the highest MaP.

This kind of approach could also be interpreted as a reinforcement learning problem: **actions** are be the possible modifications in the configurations of the simulator (eg. increase d_hot by one), the **reward** is the MaP, the **policy** used by the agent to select the best action is the policy we learn with

the neural network and the **state** is represented by the heuristics collected from the motion map plus the parameters.

Usual approaches of Q-Learning try to learn a $table : state, action \rightarrow reward$, since there could be infinite states we decide to use DeepQ-Learning by training a neural network approximating this table.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{oldvalue}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learningrate}} \times \left[\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount}} \underbrace{\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})}_{\text{bestfuturevalue}}}^{\text{learnedvalue}} - \underbrace{Q(s_t, a_t)}_{\text{oldvalue}} \right] \tag{2.1}$$

The above-explained design was obtained after numerous iterations, the next section describes the first attempt that was based on predicting the loss instead of the mean average prediction and is followed by the various changes and improvements of versions 2, 3 and 4.

Learning a policy version 1.

The iterative algorithm that is used in the first version is described below, while two significative images that summarize it are 2.35 and Figure 2.33.

- 1) select a new grey-scale sequence by changing: the video or the framerate
- 2) warmup the simulator with the first 10 frames (no change in parameters)
- 3) split the remaining parts of the video into batches of 32 frames
- 4) for each batch:
- 4.1) save the detection neural network's state and the simulator's state
- 4.2) for action in [no_change, random_action, random_action]:
- 4.2a) restore the detection neural network's state and the simulator's state
- 4.2b) simulate the motion map and collect the motion map's heuristics from the first frame
- 4.2c) train yolo detection neural network on the first 25 samples in the batch with data augmentations
- 4.2d) test the detection neural network on the last 10 samples of the batch and store the **detection loss**
- 4.2e) store the network state and simulator's state
- 4.3) set the policy loss as the MSE between the predicted reward and $(loss\_no\_action - loss_i)/loss\_no\_actio$
- 4.4) restore the system's state that had the lowest **detection loss**
- 5) with a very small chance set some random parameters in the simulator

In the first version, we jointly optimize the convolutional neural network (CNN) and the policy, but the score we are trying to predict is the loss of the neural network (instead of the mean average precision) because the previous experiments confirmed that the models with the lowest loss had the highest MaP. We must notice that training the CNN is only a side-effect and the final weights won't even be saved. This is due to the fact that the training is biased because the inputs are not shuffled, because, since we need to simulate the motion maps, we need to process the videos in order, thus risking the network forgetting the older videos.

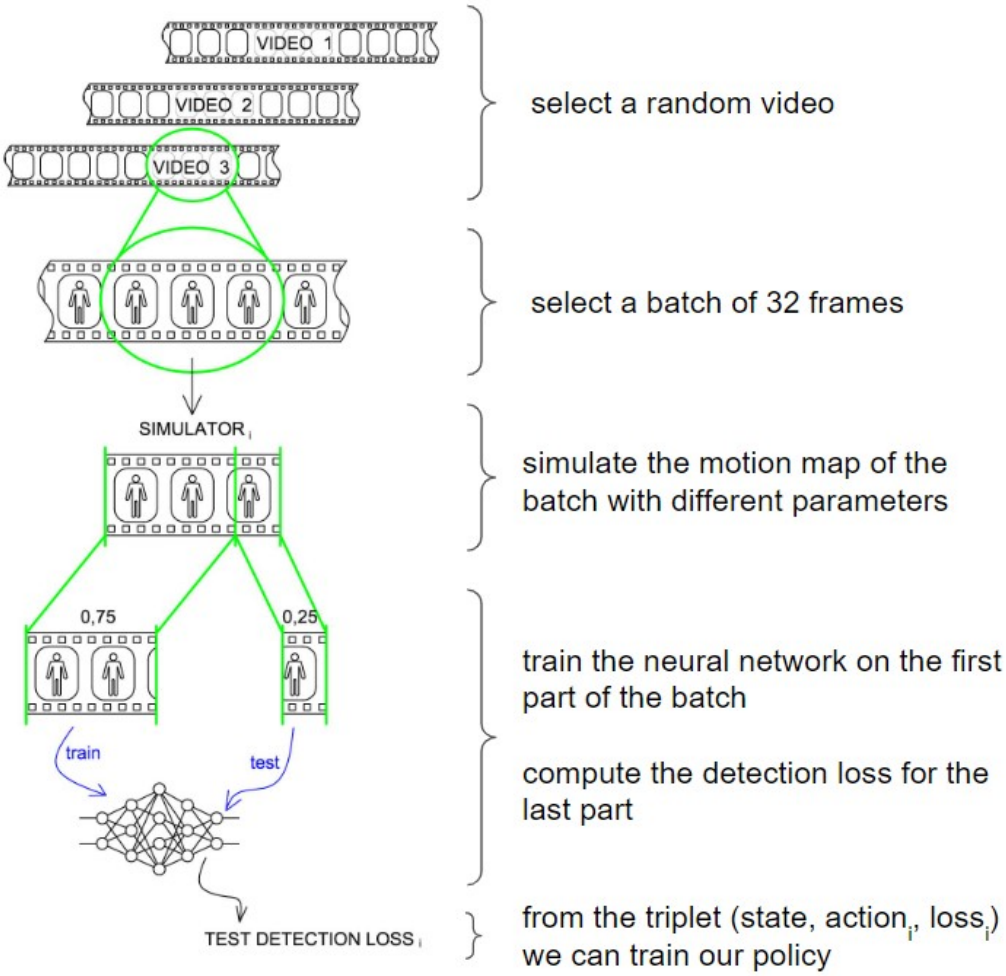

Figure 2.33: Visualization of the algorithm used to learn a policy.

We simplify the known Q-Learning formula 2.1 by removing the future reward (at least for the first version). This is justifiable in this specific problem because: (1) we assume the reward function to be convex, (2) the parameter space is convex (the best path between two simulator's parameter configurations is also the shortest). This is better shown in Figure 2.34

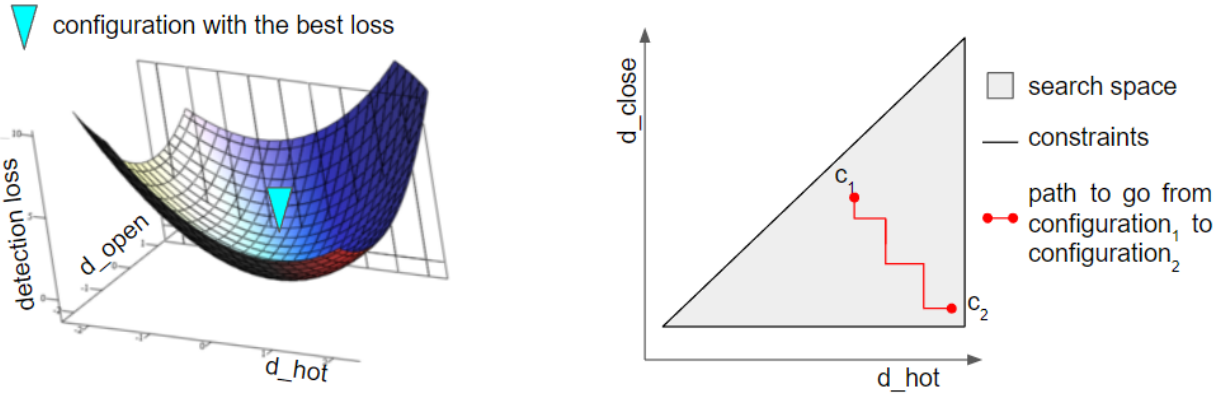

Figure 2.34: The two assumptions made in the first version: on the left a convex function represents the landscape of the detection loss of the neural network changing the simulator's parameters; on the right the search space of the simulator's parameters.

Another problem with this approach is that the loss is very dependent on the video, thus, instead of

predicting the loss we want to predict the variation in a loss that happens with a variety of parameters (which could be interpreted as the derivative of the loss function).

Characteristics of the first version of the policy. We summarize the most important points of this method in the following list. Each iteration is described by its own list to better compare the changes between each version.

- videos sampling: select a video similar to the previous one and split it into sections of 42 frames
- split train/test: of the 32 frames, use the first 25 to train the yolo CNN and the last 10 to compute the loss
- parameters sampling: select 3 parameter's configurations similar to the current ones in the simulator
- function to learn: $f(params1, params0, heuristics) \rightarrow \frac{loss_{params0} - loss_{params1}}{loss_{params0}}$
- policy training: online (offline would be: store (state,action,reward) in a CSV and train a model later.
- optimized parameters: d_close, d_open, d_hot
- state: action, parameters at the current and previous step, heuristics at the current and previous step

We now explain the reasons for each point. The selected video should be similar to the previous one to exploit the fact the detection neural network has already been trained thus the training on a similar video should be faster, for this reason, we can either: change the sequence but with more probability for the sequence to belong to the same dataset; keep the same sequence but with a different framerate; keep sequence and framerate but change the frames used for training/testing.

The choice of having batches of 32 had two advantages: (1) it's faster to compute (faster computations plus faster convergence training); (2) we have more examples for the same video to train the neural network. The sampling of the parameters to try always contains the previous configuration (the one with the best loss) and some neighboring configurations, the number of neighbors is usually around 5-6 (depending on the constraints: eg. you cannot decrease d_close if it is equal to 1). Between the possible choices of neighbors, we select 2 of them with the following probability: $softmax(reward_i)$ with $reward_i = policy(action_i, current\_paramteres, heuristics) \times exploitation$ with $exploitation = \frac{frame\_number+100}{100}$. We also introduce possible configurations that are far away from the current one, this introduces more noise, but incentive exploration and helps the algorithm to stabilize more quickly at inference time (if we want to reach $d\_open = 10$ from $d\_open = 2$ we need 8 steps if the action can modify the parameter only by one, while we need only 2 steps if we include the action +5).

Instead of predicting the loss, we predict the percentage variation of the loss wrt. the current parameters, this has the advantage that the expected variation is centered in zero thus making the exploitation/exploration formula used above more stable (if we predicted the loss directly the amount of exploration /exploitation would depend on the video).

The advantage of training the policy online is that we can use it to better choose the neighbors, but this approach has three major disadvantages: the implementation is very tricky, we can train only one model and the network could forget the information it learned before (catastrophic interference).

The input of the policy contains information from both the current step and the previous one, the reason is that the neural network will later be able to understand if information from the past is useful or not (the other versions improve the input quality)

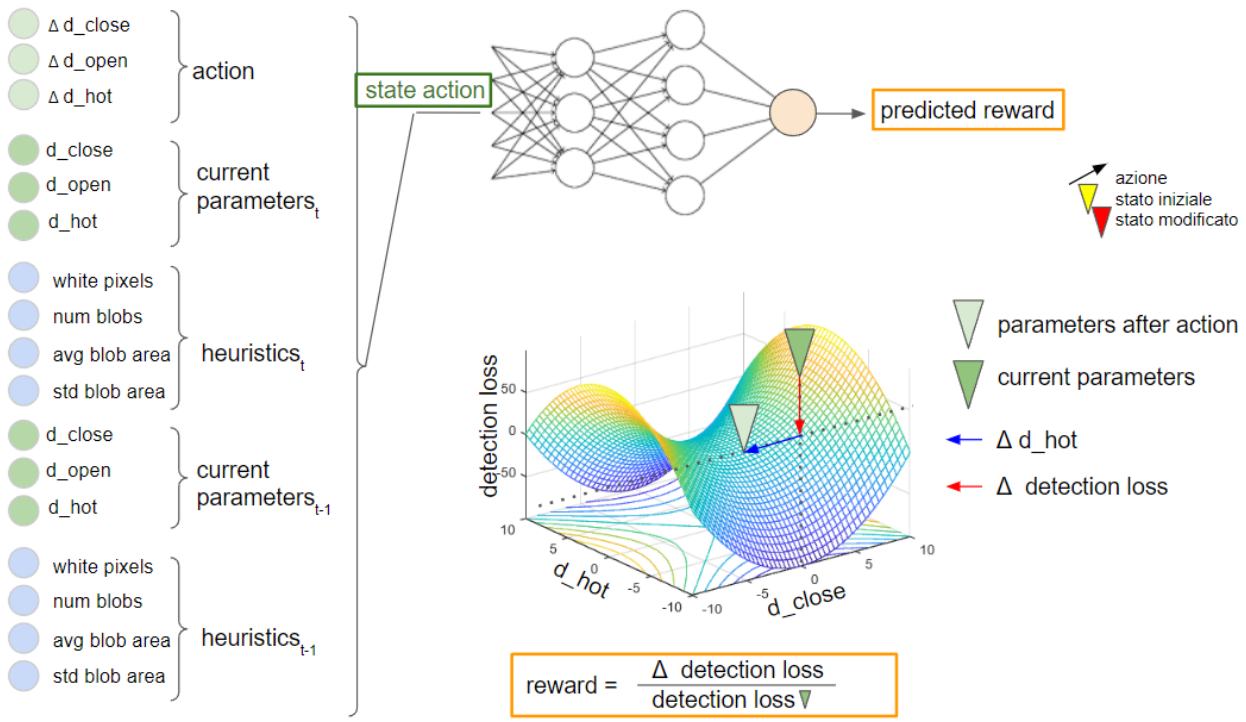

Figure 2.35: The main idea of the policy is to predict how changes in parameters will affect the loss (or the MaP for version 2).

To grant equity between runs with different simulator configurations, we reset the weights of the detection neural network and the state of the simulator before each simulation. Once we have finished trying all the configurations, we keep the parameters that achieved the lowest detection loss, as shown in Figure 2.36. A side effect of this choice is that the parameters converge towards an optimal configuration because choosing the branch with the best score is like performing hill climbing. This is positive because our sampled points are close to good configurations but it's bad because we might get stuck in that area of parameters. To avoid such a scenario, we completely randomize the simulator configuration with probability 0.2 before processing a new sequence.

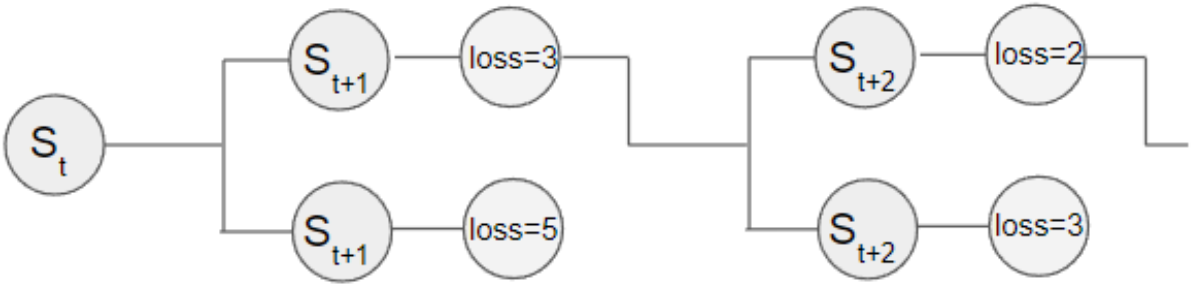

Figure 2.36: We search neighboring states of the current state of parameters, the state that produces the lowest loss in the neural network is kept alive for the next iteration. We also train a policy to predict the change in the loss from the action we took.

Before summarizing the results obtained using this first version of the policy, we want to list some

of the major advantages of this method:

- if the problem was more complex we could just increase the number of neurons in the policy to memorize how to behave in harder landscapes.
- since we "over-fit" the network over a small batch of frames, it has the time to adapt to the new motion maps generated with different parameters (which was not happening with the gradient-based approach).
- not only do we jointly optimize the neural network and the simulator, but we also learn a policy that can be used by the simulator online.
- if we found better heuristics it would be simple to substitute them in the model.

As for the results, the policy greatly improves the results with a gain in AP_{50} of 0.10 points. Although, the optimal parameters found by the policy are always equal to [1,2,3]. These parameters generate more noise and contain more information, the detection neural network can distinguish the noise from the information, thus the final predictions are more accurate, especially for pedestrians that are far away from the camera. To test if this approach is generalizable to different models, we create a dummy blob detection model, it will be discussed in the next paragraph.

Learning a policy version 2.

Prior to creating the second version of the code we run an analysis on a Jupiter notebook to explore the data we collected with the previous method. Among all the experiments we tried, the most noticeable are:

- visualize some plots about the data
- try to learn the policy with a linear model (explainability)
- try to mask some of the possible inputs to the network
- uses different numbers of hidden neurons in the policy

From this analysis, we decide to reduce the input of the network excluding the information about the past. In this way, our problem becomes a Markow stochastic process where only the current state is used to update the parameters.
Here we introduce the changes from the previous version:

- videos sampling: as before, but the probability of changing frame rate is reduced
- split train/test: we use batches of 42 frames, 32 for training, 10 for testing
- parameters sampling: all the neighbors of the current configuration
- function to learn: $f(params1, params0, heuristics) \rightarrow \frac{MaP_{params1} - MaP_{params0}}{MaP_{params0} + 0.1}$
- policy training: offline
- optimized parameters: $d\_close, d\_open, d\_hot, erosion\_kernel$
- state: action, parameters and heuristics at the current step

The most notable change is that we can also optimize the erosion kernel of the smart sensor. Since there are 2^9 possible kernels, we restrict the possible choice to only 6 of them (shown in Figure 2.37).

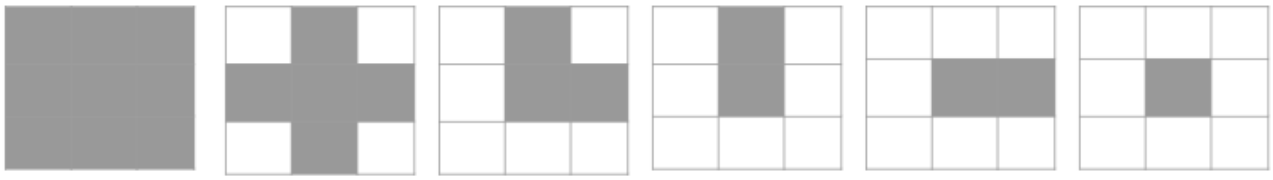

Figure 2.37: The six possible erosion kernels (grey cells apply erosion). The kernel that applies the most erosion is the leftmost, while the one that applies no erosion is the rightmost.

Another important change is obtained by training the policy offline: we save all the statistics in a *csv* and then train the model once we have collected enough data. This option allows us to: (1) test which is the most valuable information for the network: (2) try different models other than the detection neural network. Figure 2.38 shows some samples from the collected data

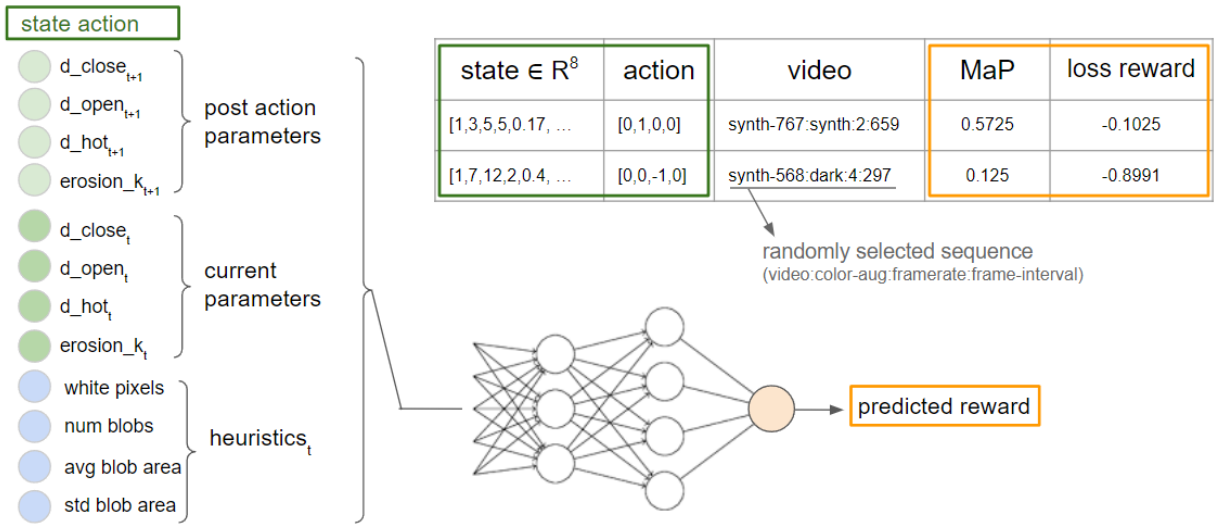

Figure 2.38: The changes applied in the second version of the policy. We store both the detection loss and the MaP, so we can choose which one to use as a reward. Moreover, we prune the amount of information contained in the state action.

Using multiple models. Other than trying to learn the function using a neural network, we try two other methods. We make some preliminary tests on a notebook and to evaluate their performances we use two metrics: (1) how many times they select the neighbor that has the best score and (2) how many times they choose a "catastrophic" configuration, that is a configuration that results in a decrease in MaP bigger than 0.1. All the models score better than the baseline random choice model, but their predictions are not perfect. The random model makes the best choice 25% of the time and makes catastrophic decisions 0.19% of the time. The NN-based policy makes the best choice 29% of the time and makes a catastrophic decision 11% of the time. The "Fixed" models, which try to reach configurations with high MaP make the right choice 39% of the time and make catastrophic choices 16% of the time.

We must notice that the dataset itself is very noisy, this could be caused by either not having trained the detection neural network enough or, more probably, by the fact that the batches used to train/test and gather statistics were too small and thus making the results very unstable. For this reason the best option to see whether a policy is good or not is to use it in practice and see how the results are.

Before presenting the results obtained with these models we describe the strategy they adopt.

Fixed models. This category of policies is made by two components: a support vector machine (SVM) used to recognize the video from the heuristics of the motion map and the target parameters for each dataset. We first test the accuracy in recognizing the original dataset (MOT17, synthMOT, Streets23) from the heuristics using a SMV with a radial basis function kernel and classification type one versus rest. The final accuracy is 62%, meaning that 2 times out of three we can recognize which is the dataset. The second part of the Fixed model is the choice of the target parameters. We adopt three strategies:

- **Fixed1**: for each video, we find the configuration with the highest MaP, then we group them by dataset and compute the **mode** for each parameter, resulting in a 3x4 matrix (3 datasets, 4 parameters).

- **Fixed2**: for each dataset, the parameters are found as the **weighted sum** of the top configurations with the weight set as the softmax of the $MaP \cdot 10$

- **Fixed3**: since the algorithm to extract the data to train the policies behaves as a local search, we can assume that, when we find a **configuration that is repeated** consecutively in different batches, we have found local optima (and possibly the global optima). Thus the target parameters are set to be the ones that have been repeated multiple times.

Once we have selected our group of target parameters and have trained the SMV in recognizing the datasets, we use the policy at inference time as follows: we extract the heuristics from the input and use them to update a running mean of the heuristics; we get a confidence prediction for each class from the SVM using the running mean, we compute the softmax of the confidence and use these values as weights to average the final target parameters. Since our model needs to return a reward wrt the input state, we return $-distance(proposed\_config, target\_config)$.

Support vector machine regression. This model is really simple, we train a regression model SVM with a radial basis function kernel to predict the mean average precision. This model seemed to be one of the most robust in the small test we performed on the notebook but later results show it is not effective. This method is referred to as **svm** in table 2.5.4.

Neural network model. This model is similar to the model used in the first version of the policy, but it introduces two major changes. The first version used to optimize the detection loss, while the new version contains information about both the loss and the MaP. Before training the loss, the user can set the weights for one of the two rewards to use (the rewards are normalized before training the network, so they have the same importance). Another change from the previous version is the fact that we re-introduce the future rewards in the predicted reward: we firstly pre-train the network to predict the MaP (or the detection loss) and then fine-tune it by setting as the final reward the one showed in 2.1. Since the architecture of the neural network could have a major impact on the performances, we try to train both a low-parameterized architecture with a hidden layer with 16 hidden neurons and a bigger one with 1024 hidden neurons.

Preliminary results. We now report the results obtained by using the policy on unseen videos of the test set.

Comparing policies for yolophi 77K			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	None	0.47	0.22
SynthMOT	None	0.62	0.26
Streets23	None	0.70	0.27
average	None	0.59	0.25
MOT17	Fixed 1 (mode)	**0.51**	0.23
SynthMOT	Fixed 1	0.60	0.26
Streets23	Fixed 1	0.81	0.35
average	Fixed 1	0.64	0.28
MOT17	Fixed 2 (weighted mean)	**0.51**	**0.24**
SynthMOT	Fixed 2	**0.65**	**0.29**
Streets23	Fixed 2	0.81	0.34
average	Fixed 2	**0.65**	**0.29**
MOT17	Fixed 3 (most repeated)	0.48	0.23
SynthMOT	Fixed 3	0.62	0.27
Streets23	Fixed 3	0.59	0.23
average	Fixed 3	0.56	0.24
MOT17	SVM (MaP regressor)	0.44	0.21
SynthMOT	SVM	0.60	0.26
Streets23	SVM	0.57	0.23
average	SVM	0.54	0.23
MOT17	Small NN (90 params)	0.44	0.21
SynthMOT	Small NN	0.56	0.24
Streets23	Small NN	**0.90**	**0.44**
average	Small NN	0.63	0.29
MOT17	Large NN (10K params)	0.44	0.21
SynthMOT	Large NN	0.60	0.26
Streets23	Large NN	0.78	0.38
average	Large NN	0.60	0.28

Another type of fixed predictor uses the best configuration of the video with the lowest MaP for every video, which in this case is {d_close=1, d_open=3, d_hot=4, erosion_kernel=3}. As expected, this configuration improves the results on MOT17, which is the dataset with the lowest MaP.

Comparing policies for Blob Detector			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	best configuration on hardest video	0.51	0.23
SynthMOT	best configuration on hardest video	0.65	0.28
Streets23	best configuration on hardest video	0.85	0.35
average	**best configuration on hardest video**	**0.67**	**0.28**

Different detector and different frame-rate. To prove the efficacy of the proposed model we test its performance with the dummy blob detector. The advantage of this model is that it has no parameters, thus its accuracy depends only on the quality of the simulator's configuration. As we can see from table 2.5.4, **fixed1**, **fixed2** and **large_nn** continue to be the best-performing policies, but with the first 2 being a lot more computationally efficient than the latter one. **SVM** and **fixed3** do not perform, thus we abandon these approaches. **Small_nn** performances are not satisfactory, analyzing the behavior of the small neural network we can see that the d_hot tends to diverge to $+\infty$ or $-\infty$; in the first case the motion maps becomes totally black while in the latter we have constraints that impose to d_hot to stay at 3, creating a configuration that is overall good.

Comparing policies for Blob Detector			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	None	0.14	0.06
SynthMOT	None	0.32	0.12
Streets23	None	0.10	0.04
average	None	0.19	0.07
MOT17	Fixed 1	**0.20**	**0.10**
SynthMOT	Fixed 1	0.32	0.12
Streets23	Fixed 1	0.35	0.16
average	Fixed 1	**0.29**	**0.12**
MOT17	Fixed 2	0.19	0.08
SynthMOT	Fixed 2	**0.33**	**0.13**
Streets23	Fixed 2	0.30	0.12
average	Fixed 2	0.27	0.11
MOT17	Fixed 3	0.14	0.06
SynthMOT	Fixed 3	0.32	0.12
Streets23	Fixed 3	0.10	0.04
average	Fixed 3	0.18	0.07
MOT17	SVM	0.15	0.07
SynthMOT	SVM	0.19	0.07
Streets23	SVM	**0.41**	**0.18**
average	SVM	0.25	0.11
MOT17	Small NN	0.10	0.05
SynthMOT	Small NN	0.19	0.06
Streets23	Small NN	0.08	0.00
average	Small NN	0.12	0.04
MOT17	Large NN	0.10	0.04
SynthMOT	Large NN	0.31	0.12
Streets23	Large NN	0.36	0.16
average	Large NN	0.26	0.11

As we have previously seen, the performances on the baseline model were drastically reduced for the 15 fps simulation. We train a policy on the 15 fps dataset and see whether the optimized configuration improves the results. As we can see, the policy greatly helps improve the results.

Comparing policies for yolophi 77K at 15 fps			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	None	0.48	0.22
SynthMOT	None	0.56	0.24
Streets23	None	0.20	0.07
average	None	0.41	0.18
MOT17	Fixed 1	0.51	0.24
SynthMOT	Fixed 1	0.59	0.26
Streets23	Fixed 1	0.49	0.15
average	Fixed 1	0.53	0.22
MOT17	Fixed 2	**0.53**	**0.25**
SynthMOT	Fixed 2	**0.63**	**0.29**
Streets23	Fixed 2	0.35	0.09
average	Fixed 2	0.50	0.21
MOT17	Large NN	0.50	0.23
SynthMOT	Large NN	0.50	0.23
Streets23	Large NN	**0.84**	**0.34**
average	Large NN	**0.61**	**0.27**

We showed that these policies can improve the results for both detection neural network (yolo) and other models like the blob detector, moreover, they can both improve already satisfying configurations and improve more challenging videos (higher fps); thus we can say that the policies are effective in optimizing the simulator's configurations, especially the fixed models.

Learning a policy version 3.

Seen the success of the fixed models we want to improve and stabilize them.

- videos sampling: ordered sequence of videos, no change of frame-rate
- split train/test: we use the whole video, the first part for training the last part for testing
- parameters sampling: a set of sparse configurations, plus the neighbours of the best current configuration, plus some crossovers.
- function to learn: None
- policy training: offline
- optimized parameters: $d\_close, d\_open, d\_hot, erosion\_kernel$
- state: action, parameters and heuristics at the current step

The main difference is that we are not trying to learn a reward function to guide our choices, we are just searching for the configurations that achieve the highest MaP.

This method is the simplest one and is just hill climbing with some more incentives for exploration. These incentives come from:

- always try to use some pre-defined configuration that usually perform well and are far from each other in the search space
- expand the exploration even more by crosser of the current best solution with the pre-defined solutions

The most important aspects to ensure the success of this algorithm are:

- ensuring that the detection neural network is properly trained.
- computing a robust reward function.
- computing robust heuristics for the image.

To ensure the network is trained properly we keep track of the MaP on a validation set and keep training the network until that score stops growing. To compute a robust reward we cannot only use the last frames of the sequence because, since the sequences are pretty short, they also are not enough to grant a stable estimate. For this reason, we compute the score as follows: $0.6 \cdot MaP(test\_set) + 0.4 \cdot MaP(train\_set)$. This leverages both the ability of the network to memorize patterns and the ability to generalize. Motion map's heuristics contained a lot of variances, to make them less noisy we use the bias-corrected running average ($H_0 = \{0\}^{10}$ and $H_t = H_{t-1} \times 0.66 + heuristics(motion\_map_t) \times 0.34$ and $H_t = \frac{H_t}{1-0.66^t}$, with 10 being the number of heuristics).

Once we have found the best parameters for each video we cluster similar parameters into groups and train an SMV to classify the heuristics of the motion maps to its cluster. Figure 2.39 illustrates how this predictor works. We can see that this method is pretty robust, in fact when we set the number of clusters equal to one, the target configuration is the mean of the optimal configurations.

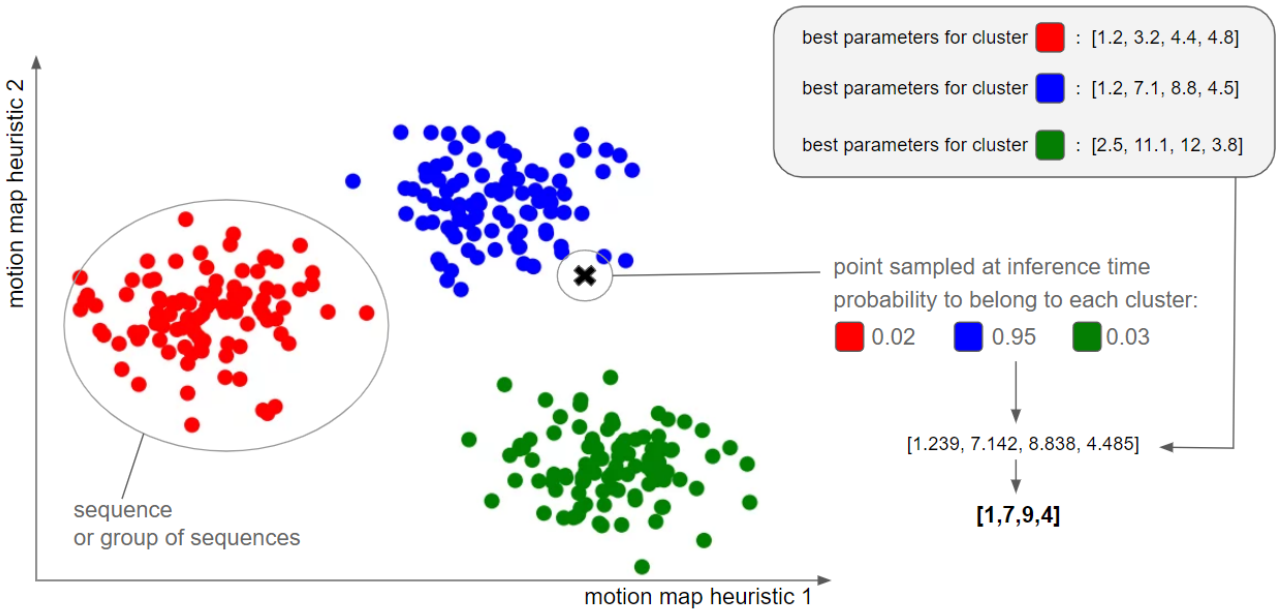

Figure 2.39: The parameters proposed by the fixed predictor are a weighted sum of N optimal configurations (3 in the image), the more the heuristics of a motion map are similar to the heuristics of a cluster the more that configuration will contribute to the final target.

This new method improves over the previously proposed methods and the results are shown in the following table.

Comparing policies for yolophi 77K			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	v2-Fixed 1	0.51	0.23
SynthMOT	v2-Fixed 1	0.60	0.26
Streets23	v2-Fixed 1	0.81	0.35
average	v2-Fixed 1	0.64	0.28
MOT17	v3-Fixed: 1 cluster	0.52	**0.25**
SynthMOT	v3-Fixed: 1 cluster	0.64	0.29
Streets23	v3-Fixed: 1 cluster	**0.89**	**0.45**
average	v3-Fixed: 1 cluster	0.68	**0.33**
MOT17	v3-Fixed: 3 clusters	0.49	0.24
SynthMOT	v3-Fixed: 3 clusters	**0.67**	**0.30**
Streets23	v3-Fixed: 3 clusters	**0.89**	0.43
average	v3-Fixed: 3 clusters	0.68	0.32
MOT17	v3-Fixed: 8 clusters	**0.53**	**0.25**
SynthMOT	v3-Fixed: 8 clusters	**0.67**	**0.30**
Streets23	v3-Fixed: 8 clusters	**0.89**	0.42
average	v3-Fixed: 8 clusters	**0.70**	0.32

We can notice that the amount of clusters isn't too impactful, moreover, we can outperform the previous version only by using one cluster, this is probably due to a minor amount of noise in the data we gathered: larger batches of frames and improved neural network convergence are fundamental to extract valuable data.

Learning a policy version 4.

As introduced at the start of the section, we want to train a feed-forward neural network to learn the landscape of the mean average precision function. The previous versions gave us some insights on how

to properly extract the information from our simulations and how to properly train the network; here we report the main findings:

- use MaP instead of the detection loss: the detection loss can only be used with specific models, while MaP can be used with every detector (eg. the blob detector). Moreover, some experimental tests conducted during the development of the second version seemed to indicate that policies based on MaP gain around 0.02 more MaP points over the ones based on detection loss.
- which inputs to use: for example, we add six new heuristics for the motion map, because video recognition from the heuristic with SVMs was working, but had margins of improvement.
- bigger batches: using small batches of frames to get the MaP score gives very noisy results.
- direct prediction of the score instead of the variation of score between two configurations: theoretically the two approaches both work, but the latter ($f(c1, c2) : MaP_{c1} - MaP_{c2}$, with $c2 = state$ and $c1 = do(state, action)$) requires more parameters with no apparent gains over the other ($f(c) : MaP$, with $c = do(state, action)$).

As in the other versions, we summarize the main points of the policy training:

- videos sampling: ordered sequence of videos, no change of frame-rate
- split train/test: we use the whole video, the first part for training and the last part for testing
- parameters sampling: a set of sparse configurations, plus the neighbors of the best current configuration, plus some crossovers.
- function to learn: $f(state, action) \rightarrow 0.8 \cdot MaP + 0.2 \cdot f(state, a*)$ with $a* = argmax_a f(do(state, action), a)$
- policy training: offline
- optimized parameters: $d\_close, d\_open, d\_hot, erosion\_kernel$
- state: action, parameters and heuristics at the current step

Instead of directly predicting the MaP, we use the popular reinforcement learning formula that takes into account the MaP of the best neighbor. This should help prevent getting stuck in attraction basins (if they existed), but, most importantly, helps smooth the function, causing similar configurations to have similar scores.

As discussed earlier 2.5.4, the neural network's policies risk to diverge, this could be caused by the fact that we train the neural network only on good configurations, but we never expose it to non-sense parameters and thus the network can predict high scores for obviously bad configurations. To avoid this problem, we add to the dataset some synthetic points that represent degenerate configurations and assign to them a MaP score of 0. In this way, the policy should not diverge anymore.

Problem connotation. Before reporting the results, we dwell into some of the intuition we learned while facing this problem.

Not only the optimal configuration of the simulator changes from video to video, but the analysis of the data showed that it changes for different intervals of frames within the same video. To produce significative statistics about the best configurations we are required to train our network on large portions of the video (since the 3^{rd} version we use the whole sequence), but training the network on two sequences at the same time could improve the collected association between parameters and MaP even more. This idea is justifiable by the following intuition: if each video has its own optimal configuration, the patterns generated in the motion maps could change a lot between videos, this would generally not be a problem, because the neural network can learn any pattern, but, since we have architectural constraints, it is possible that smaller network could not learn all the possible patterns and thus, even if the MaP achieved by the network for that specific video is optimal when trained only on that sequence, when trained together with all the videos in the dataset, the results could be inferior.

Using neural networks (NN) to predict the mean average precision is a perfect way to create a trade-off between optimizing the whole dataset or optimizing the single sequences. In fact, if the NN is low parameterized, its predictions will be very simple and the optimal configurations will be very similar for each video; while, if the NN has a lot of parameters, it could possibly memorize the whole (configuration $\rightarrow$ MaP) dataset and thus, will be able to set the optimal parameters for each video.

Neural networks architectures. We train 6 different architectures to see which one of them offers the best trade-off and achieves the best results. Before reporting the result's table we describe each architecture.

The first architecture, called **experimental**, is based on the assumption that the smoothed MaP function is convex (as we assumed in the first version of the policy). To leverage this fact, the architecture contains a small module with 32 parameters that are used to measure the distance of the proposed configuration from the weights of the network (which could represent optimal configurations). The resulting distances are then concatenated to the input and processed by a feed-forward neural network (FFNN) that returns the predicted smoothed MaP.

The other five architectures are simple FFNNs and their number of parameters ranges from 100 to 5000. A common pattern groups these architectures which contain a batch normalization layer as the first layer, dropout layers set to 0.1 after feature layers with more than 32 activations and use ReLU as activation function ($relu(x) = max(0, x)$). The input of each neural network is a batch of 14-dimensional vectors (4 parameters and 10 heuristics), thus an architecture with a single output and a hidden layer containing 8 neurons (plus the bias) will be described as [14,8,1]. The details about the architectures, sorted by scale are listed below:

- **xs**: structure[14, 4, 1] ; total parameters: 93
- **s**: structure[14, 4, 4, 4, 1] ; total parameters: 133
- **experimental**: structure[14, cat(16,distances), 4, 1] ; total parameters: 409
- **m**: structure[14, 32, 1] ; total parameters: 541
- **l**: structure[14, 16, 32, 4, 1] ; total parameters: 949
- **xl**: structure[14, 256, 4, 2, 1] ; total parameters: 4909

We conduct the experiments again and report the results in the following table.

Comparing policies for yolophi 77K			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	None	0.47	0.22
SynthMOT	None	0.62	0.26
Streets23	None	0.70	0.27
average	None	0.59	0.25
MOT17	v4-xs	**0.52**	**0.25**
SynthMOT	v4-xs	**0.70**	**0.31**
Streets23	v4-xs	**0.89**	0.39
average	v4-xs	**0.70**	**0.31**
MOT17	v4-s	0.48	0.22
SynthMOT	v4-s	0.68	0.29
Streets23	v4-s	0.46	0.19
average	v4-s	0.54	0.23
MOT17	v4-m	0.41	0.19
SynthMOT	v4-m	0.66	0.28
Streets23	v4-m	0.86	0.48
average	v4-m	0.64	**0.31**
MOT17	v4-l	0.37	0.18
SynthMOT	v4-l	0.45	0.19
Streets23	v4-l	0.65	0.47
average	v4-l	0.49	0.28
MOT17	v4-xl	0.38	0.18
SynthMOT	v4-xl	0.57	0.25
Streets23	v4-xl	0.83	0.55
average	v4-xl	0.59	0.32
MOT17	v4-experimental	0.44	0.22
SynthMOT	v4-experimental	0.66	0.29
Streets23	v4-experimental	0.88	**0.41**
average	v4-experimental	0.66	**0.31**

Neural network-based policies bring results similar to the baseline or improve the results, but are still outperformed by the fixed predictors (version 3). The results are similar to the ones obtained in the second version thus, it is very likely that even the methods that do not bring large improvements at 4 fps, will be much more effective for videos at 15 fps.

The policies are trained on data collected only from MOT17 [61] and SynthMOT [28]. Although, we can see that the policy is effective on the Streets23 dataset as well, thus, this method seems to be robust to domain shifts.

It seems that the size of the model is not correlated with the final metrics, this could possibly be caused by instability during training. The experimental architecture seems to be easier to optimize and is probably the best choice for predicting the MaP from the configuration and heuristics.

The smallest model (**xs**) always converges to {d_close=1, d_open=2, d_hot=3, erosion_kernel=5} as configuration. This configuration lies on the constraints border, thus, it is hard to assess whether it is reached because the model was diverging (eg. wants to minimize d_hot as much as possible) or because the model believes this is a good configuration. The other models change target configuration depending on the video.

Simulator's optimization conclusions. We try different methods for optimizing the configuration of the simulator and we achieved great boosting in performances that are even able to surpass the grey-scale-based model.

A possible explanation for why the differentiable approach does not work is that, when computing the gradient for a small batch of frames, the gradients resulting from the back-propagation are too unstable and thus, they keep changing. Moreover, even if we managed to optimize the simulator's

parameters through differentiation, this method could only be applicable for offline optimizations, while the policy-based methods can be directly used for online tuning of the simulator's parameters, and works in unseen scenarios too.

The following chapter will put together all the tools we developed in this chapter and list the final results.

3 Results

The previous chapter introduced a baseline model, then discussed how to improve the architecture of the detection neural network, and finally how to optimize the simulator's parameters. This chapter uses a combination of the previously cited techniques to obtain a final evaluation of the performances and discusses which configurations work better.

3.1 Problems in the Baseline Model

The baseline model at 4 frames per second offers solid performances but leaves room for improvement. Figure 3.1, Figure 3.2 and Figure 3.3 show the predictions of the baseline model respectively on MOT17, SynthMOT and Streets23; these frames have been selected to showcase the most recurring problems of the baseline, with the most important being the incapability to detect small bounding boxes. This problem, although, it's pretty hard to solve even when lowering the detection threshold for the motion map because the input resolution is too low to work with objects too far away and this should be taken into consideration when using these models.

Figure 3.1: Baseline predictions on MOT17.

We now list some of the problems that emerge from visual analysis.

- noise is often viewed as a false positive: 1a, 1b, 1f.
- very small bounding boxes don't generate enough movement to be detected: 1b.
- bounding boxes of people next to each other get fused into one: 1d, 1e.

Figure 3.2: Baseline predictions on SynthMOT.

Figure 3.3: Baseline predictions on Streets23.

Similar problems can be seen also in Streets23 and SynthMOT, moreover here a new issue emerge in Streets 23:

- only the movement of the torso is detected: 3a, 3d.
- darker video does not generate enough movement to detect objects: 3b.

The baseline model seems strong in detecting pedestrians when the movement is clear. on the other side noise often generates false positives, probably because the motion maps rarely contain noise, thus, when movement is detected a bounding box is generated. Moreover, the visibility of the pedestrians is somewhat reduced by the choice of the simulator's parameters.

A remarkable problem of the baseline is that the CNN we use is probably too heavy to be run on an edge device. Thus the proposed solution should be more efficient than the current one.

3.2 Mixing It All Up

We will now report the improvements we obtained using the policy instead of the default configuration of the simulator for various models and using quantization, this should help us determine which is the most robust approach.

We developed more than 10 different strategies to create policies in the previous chapter and compared them to understand which one of them was working well. We select three of the most promising ones for a final test to see which one is the best one. The policies we test are: the fixed predictor with eight clusters (**fix-8**), the MaP predictor using the experimental neural network (**nn-ex**), and the policy that uses the best configuration for the video with the lowest MaP for every video (**bl**).

Here we report the results for the **baseline** model without using any policy, this is the model that achieves the highest scores, but, as previously shown is very inefficient.

4fps, 8-bit quantized phiyolo 104K parameters, policy: static-svs[1,3,5] (baseline)				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	**5.09**	0.51	0.26
SynthMOT	//	**6.94**	0.64	0.28
Streets23	0.79	**0.24**	0.61	0.24
average	0.79	**4.09**	0.59	0.26

Now we report the results obtained using the policies:

- nn-ex: given the state of the simulator and an action predicts the change in parameters that produces the highest MaP.

4fps, 8-bit quantized phiyolo 104K parameters, policy: nn-ex				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	6.00	0.48	0.27
SynthMOT	//	6.98	0.70	0.32
Streets23	0.80	0.32	**0.94**	**0.50**
average	0.80	4.43	0.70	0.36

- fix-8: divides the best parameters into 8 clusters. At inference time: we compute a weighted sum of the clusters to get a target configuration, the weights are proportional to the similarity of the video wrt. the ones in the training set.

4fps, 8-bit quantized phiyolo 104K parameters, policy: fix-8				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	5.64	0.58	**0.28**
SynthMOT	//	6.97	0.72	**0.35**
Streets23	**0.82**	0.30	**0.94**	**0.50**
average	**0.82**	4.30	**0.74**	**0.37**

- bl: this policy is very simple, it uses the best configuration found for the most challenging video (the one that scored the lowest MaP) for every video.

4fps, 8-bit quantized phiyolo 104K parameters, policy: bl				
Dataset	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	//	5.31	**0.65**	**0.28**
SynthMOT	//	6.96	**0.73**	**0.35**
Streets23	**0.82**	0.27	0.88	0.39
average	**0.82**	4.18	0.68	0.34

Of the three policies, fix-8 reaches the highest scores. It is simple to see why this solution works by thinking of the scenario with only one cluster, for which, the target configuration is the mean of the best configurations for each video. As shown in the previous chapter, the policy using just one cluster reaches results similar to the 8-clusters policy, thus opening a tradeoff: using the one cluster strategy, which does not require the use of motion map heuristics, but does not adapt to the scene or using multiple clusters which can improve results when there is a lot of diversity in the scenes but has a higher computational cost.

Results for PhiYolo 77K To have a more fair comparison we test the policy on multiple models: **phiyolo 77K**, **phiyolo 7K**. We select these models between all the models we previously presented because they offer the best tradeoff between performances and hardware requirements: phiyolo 77K in the network achieves performances similar to the baseline model using a lot less computational resources; phiyolo 7K is the smallest model with a multiply-accumulate count lower than 3 million. We also try to use mlp2 since it seemed it was capable of working with binary convolutions; unfortunately, the final results weren't good enough to be reported, thus we can discard this model. The policy called **None** corresponds to the baseline, which uses the simulator with the recommended static parameters: {d_close=1, d_open=3, d_hot=5, erosion_kernel=0 }.

4fps, 8-bit quantized phiyolo 77K parameters					
Dataset	Policy	F1:Triggering	MAE:Counting	AP50:Detection	MaP:Detection
MOT17	None	//	6.11	0.48	0.22
SynthMOT	None	//	8.91	0.59	0.24
Streets23	None	0.78	**0.27**	0.68	0.24
average	None	0.78	5.09	0.58	0.23
MOT17	nn-ex	//	6.11	0.43	0.22
SynthMOT	nn-ex	//	8.91	0.63	0.27
Streets23	nn-ex	**0.79**	**0.27**	0.81	0.35
average	nn-ex	**0.79**	5.09	0.62	0.28
MOT17	fix-8	//	**5.82**	0.50	0.24
SynthMOT	fix-8	//	**7.91**	0.67	**0.30**
Streets23	fix-8	0.78	0.30	**0.88**	**0.45**
average	fix-8	0.78	**4.67**	**0.68**	**0.33**
MOT17	bl	//	6.57	**0.51**	**0.25**
SynthMOT	bl	//	8.88	**0.68**	0.29
Streets23	bl	0.78	0.35	0.84	0.35
average	bl	0.78	5.27	**0.68**	0.29

As for phiyolo 104K, all the policies outperform the baseline, but the policy using a neural network (nn-ex) to predict the best configuration of the parameters seems to be the weakest among the three.

Results for PhiYolo 7K

4fps, 8-bit quantized phiyolo 7K parameters					
Dataset	Policy	F1: Triggering	MAE: Counting	AP50: Detection	MaP: Detection
MOT17	None	//	6.70	0.35	0.12
SynthMOT	None	//	9.88	0.51	0.19
Streets23	None	0.83	0.29	0.53	0.17
average	None	0.83	5.62	0.46	0.16
MOT17	nn-ex	//	**6.11**	0.29	0.10
SynthMOT	nn-ex	//	**8.91**	0.50	0.18
Streets23	nn-ex	0.86	**0.27**	0.72	0.28
average	nn-ex	0.86	**5.09**	0.50	0.19
MOT17	fix-8	//	7.15	0.40	0.15
SynthMOT	fix-8	//	9.88	**0.54**	**0.20**
Streets23	fix-8	**0.87**	0.91	**0.80**	**0.32**
average	fix-8	**0.87**	5.98	**0.58**	**0.22**
MOT17	bl	//	8.40	**0.46**	**0.18**
SynthMOT	bl	//	11.65	0.40	0.15
Streets23	bl	0.77	0.25	0.63	0.25
average	bl	0.77	8.55	0.49	0.19

We can notice that the nn-ex policy obtains very high scores on the Streets23 dataset while doesn't perform too well on MOT17, this could be due to the fact that the training set contains 13 videos from Streets23 and only 3 videos from MOT17, thus the produced statistics are unbalanced in favor of Streets23.

The bl policy used the best configuration from a video from MOT17, thus, it isn't surprising to see low performances on the triggering task, which is only calculated for Streets23.

As we can see from the tables, the fixed predictor that clusters the eight best-performing configurations seems to be the most consistent method, thus in a real-case scenario using this policy would be the best choice, moreover, the behavior of this policy is explainable, while the experimental NN is not.

3.3 Policy vs Baseline

This section visually compares motion maps generated with the policy and with the baseline simulator to understand how the policy improves performance.

While the parameters found by the policy are often similar to the ones recommended by [109] and used in the baseline, the main difference is created when selecting the erosion kernel. The default erosion applied to the image uses a full 3x3 kernel, while the one found by the policy nullifies the erosion. This results in images with a little more noise, but, the neural network seems to be able to learn how to recognize noise and thus improves the detections. As a side note, in the first versions of the policy, the kernel couldn't be modified; in those cases, the strategy found by the policy was to reduce the movement detection threshold (d_hot) as much as possible, which yet resulted in more noisy images, but with increased performances.

As shown in Figure 3.4, even if the baseline motion maps are cleaner at first sight, the neural network prefers raw images because they contain more information. This lack of information that is probably lost after the erosion seems to be fundamental for the network to distinguish noise from true positive detections, as we can see in the second row.

We also should notice that noisier images are more expensive to process because the amount of

Figure 3.4: Predictions on MOT17, we can see the baseline on the left, the ground truth annotations in the middle and the results obtained using the policy on the right. The baseline achieves an AP_{50} of 0.47, while the policy improves to 0.53.

non-zero multiplication is increased. We must re-state that the main objective of the optimization of the simulator was to increase the MaP, although, if the noise should be considered an important factor for the total computational cost, it would be sufficient to re-train the policies taking into account the sparsity of the motion. For example, instead of using $score = MaP(configuration)$, we could use $score = MaP(configuration) + sparsity(configuration)$, where $sparsity = 1 - \frac{1}{n\_fr \cdot 120 \cdot 160} \cdot \sum_{n=1}^{n\_fr} sum(mot\_map_{n,config})$, with sum being a function that returns the sum of all the pixels in the motion map.

Comparison at 15 frames per second.

The baseline configuration of the simulator struggles at processing videos with higher framerates. We run another comparison using the quantized models to see how performance changes from the 4fps scenario.

We report the results for phiyolo 77K using the baseline configuration, the cluster-based policy, the neural network-based policy and a handcrafted architecture: we choose {d_close=1, d_close=2, d_close=3, erosion_kernel=2}, the idea is that with a faster framerate, the background threshold will be updated more quickly, thus we want to reduce d_open; moreover erosion seemed to degrade

Figure 3.5: Predictions on SynthMOT and Streets23, we can see the baseline on the left, the ground truth annotations in the middle and the results obtained using the policy on the right. The baseline achieves an AP_{50} of 0.62 and 0.70 on synthMOT and Streets23 respectively, while the policy reaches 0.67 and 0.89.

performances, thus we use a kernel that can remove only one-pixel blobs.

Training the phiyolo 77K network for videos at 15 fps			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	None	0.47	0.20
SynthMOT	None	0.55	0.22
Streets23	None	0.24	0.06
average	None	0.42	0.16
MOT17	handcrafted	0.48	0.20
SynthMOT	handcrafted	0.62	0.25
Streets23	handcrafted	0.81	0.21
average	handcrafted	0.64	0.22
MOT17	fix-8	**0.51**	**0.23**
SynthMOT	fix-8	0.61	0.28
Streets23	fix-8	**0.98**	**0.44**
average	fix-8	**0.70**	**0.32**
MOT17	nn-ex	0.50	**0.23**
SynthMOT	nn-ex	**0.67**	**0.31**
Streets23	nn-ex	0.94	0.39
average	nn-ex	**0.70**	0.31

Training the phiyolo 7K network for videos at 15 fps			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	None	0.33	**0.11**
SynthMOT	None	0.50	0.18
Streets23	None	0.40	0.09
average	None	0.41	0.13
MOT17	fix-8	**0.35**	**0.11**
SynthMOT	fix-8	0.48	0.17
Streets23	fix-8	0.82	0.18
average	fix-8	0.55	0.15
MOT17	nn-ex	0.31	**0.11**
SynthMOT	nn-ex	**0.54**	**0.20**
Streets23	nn-ex	**0.87**	**0.26**
average	nn-ex	0.57	0.19

Motion history images with policy

Since the performances of the model based on binary convolutions (XNor-Nets [72]) are not up to expectations, quantizing the network's weights and activations to have 8-bits seems to be the best option. This opens up the possibility to use history motion maps, which, as we saw in the previous chapter, achieves the highest metrics scores over all the input types we tried.

For this reason, we train a policy to optimize their usage and use it on phiyolo 77K and 7K, giving us the best combination of techniques we have used so far.

Training the phiyolo 77K network with motion history images			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	handcrafted [1,3,5,3]	0.50	0.15
SynthMOT	handcrafted [1,3,5,3]	0.32	0.10
Streets23	handcrafted [1,3,5,3]	0.33	0.10
average	handcrafted [1,3,5,3]	0.38	0.12
MOT17	fix-8	0.52	**0.25**
SynthMOT	fix-8	0.67	0.29
Streets23	fix-8	0.81	0.36
average	fix-8	0.67	0.30
MOT17	nn-ex	**0.54**	0.24
SynthMOT	nn-ex	**0.73**	**0.32**
Streets23	nn-ex	**0.89**	**0.43**
average	nn-ex	**0.72**	**0.33**

We can notice that the policy is even more effective for the motion history image-based models. Moreover, history motion maps are still sparse inputs that contain mostly zeros, thus adopting them over the simple motion map isn't a big increase in computations. This type of model does not increase excessively the computational cost; it's hard to clearly quantify the amount without knowing scene-related details like the average number of objects their speed and the amount of noise.

Training the phiyolo 7K network with motion history images			
Dataset	Policy	AP50: Detection	MaP: Detection
MOT17	handcrafted [1,3,5,3]	0.34	0.11
SynthMOT	handcrafted [1,3,5,3]	0.46	0.16
Streets23	handcrafted [1,3,5,3]	0.75	0.25
average	handcrafted [1,3,5,3]	0.51	0.17
MOT17	fix-8	0.38	**0.14**
SynthMOT	fix-8	**0.52**	**0.19**
Streets23	fix-8	0.69	0.25
average	fix-8	0.53	0.19
MOT17	nn-ex	**0.39**	0.13
SynthMOT	nn-ex	0.51	**0.19**
Streets23	nn-ex	**0.80**	**0.29**
average	nn-ex	**0.57**	**0.20**

Unfortunately, using motion history maps does not generate big improvements wrt. plain motions maps, thus making the latter the best input for object detection. These results differ from the previous chapter where MHI could reach better scores; a possible explanation is that quantization greatly reduces the learning capabilities of the network; another explanation could be that the model used in the baseline had more parameters (phiyolo104K). For this reason, we train an 8-bit quantized model on the greyscale images, the final performances of: 0.31 for phiyolo baseline and 0.16 for phiyolo 77K show that processing greyscale images requires more computational power, and thus, it is not a viable option for edge computing.

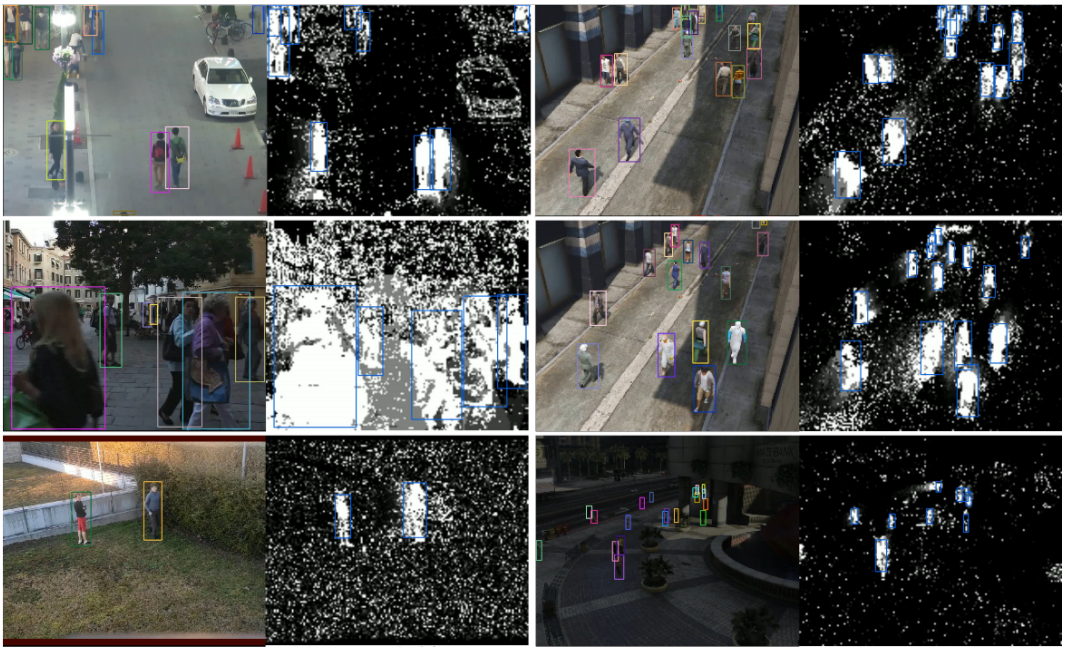

Figure 3.6: Predictions from the history motion maps (right) and ground truth (left). The predictions come from phiyolo77K trained with the nn-ex policy.

Comparison Between Models

After having reported many results on the performances of the two main architectures (**phiyolo 77K** and **phiyolo 7K**), we conclude the Results chapter with an illustration that shows differences some differences in how the models make predictions.

Figure 3.7: Some prediction of the 2 models for the same frame. Often the predictions are almost indistinguishable, but, in case of errors, phiyolo 77K fails more gently.

At first sight 3.7, we cannot disclose significative differences between the two models, but a more accurate inspection shows that the model with more parameters has better prediction in presence of noise.

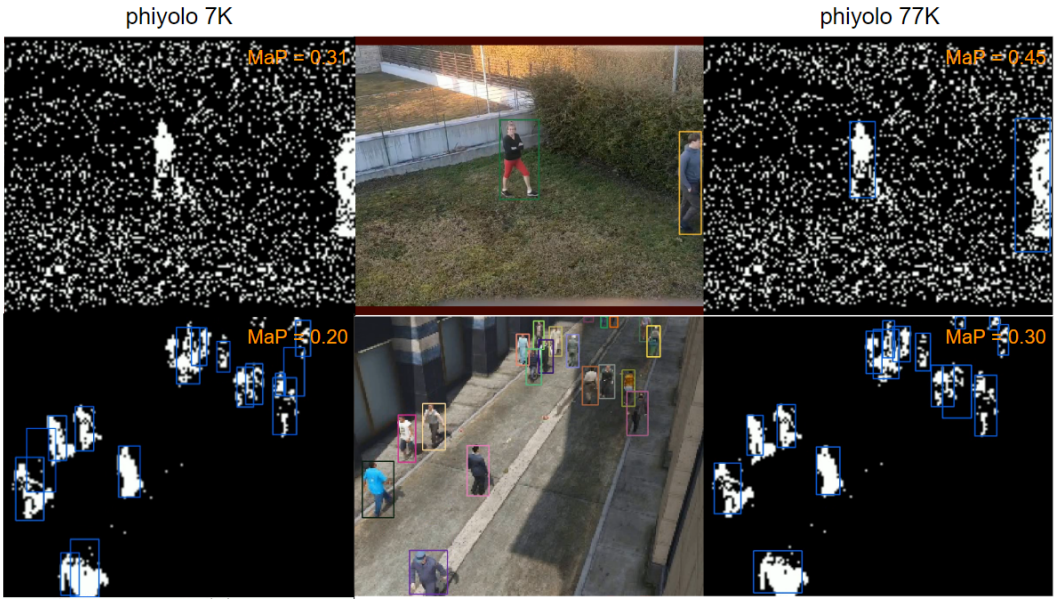

Figure 3.8: Some prediction of the 2 models for the same frame.

As discussed in the previous chapter the policy hasn't been trained on Streets23, thus the instability of the camera united with parameters optimized for the other datasets causes some frames to be very noisy (Figure 3.8). As we can see, phiyolo 77K can handle these challenges while phiyolo 7K fails.

In any case, phiyolo 77K is expected to perform better given its size, moreover, considering that phiyolo 7K requires one-tenth (30M MACC vs 2.8M MACC) of computational resources, makes it a very solid alternative for edge computing.

Comparison Between Policies

We conclude the results chapter with a comparison of the results achieved by the policies 3.9.

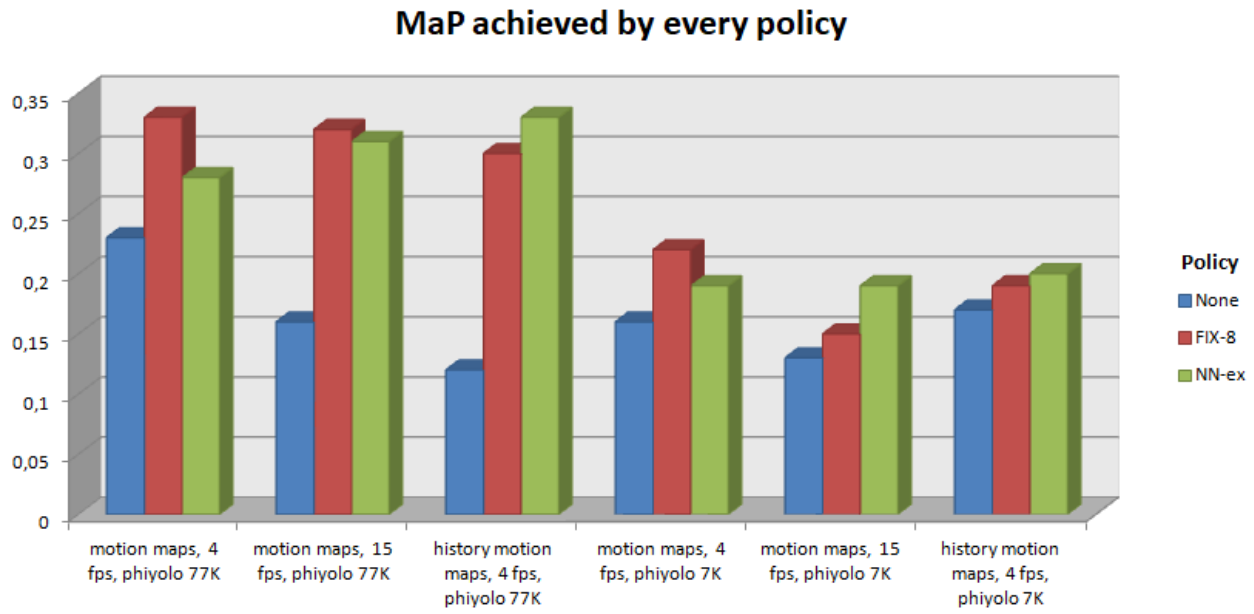

Figure 3.9: Summary of the policies' performances.

As we can see the policies are always improving the results over the baseline (None), proving them to be an effective tool to optimize the simulator's parameters. There isn't a clear winner between the two policies, in fact, each one of them is more effective for specific datasets: **fix-8** achieves the best performances for synthMOT and MOT17 while **nn-ex** achieves the best performances on Streets23.

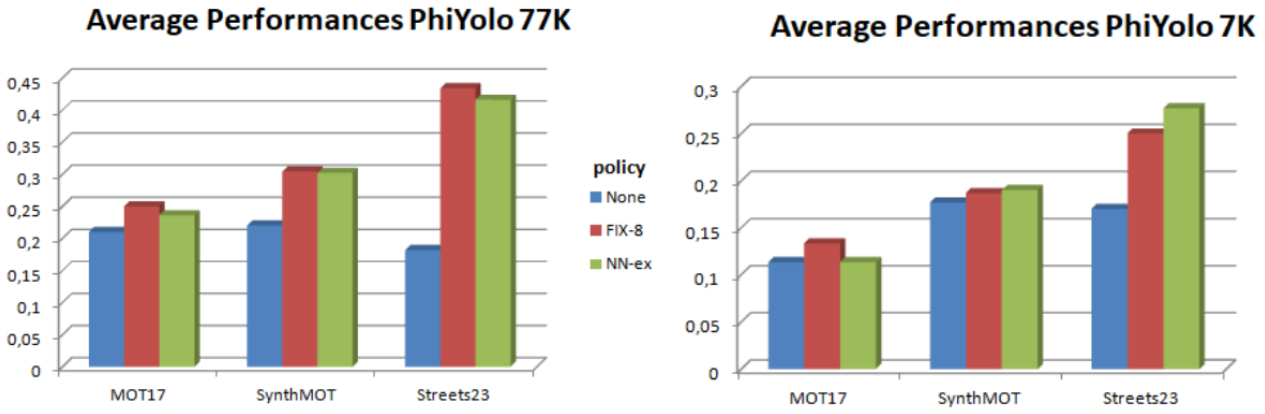

Figure 3.10: Averaging the scores obtained by the policy for each dataset.

To conclude, Figure 3.10 compares the effect of the policies on each dataset, as we can notice, it seems that the policy impacts more the model with more parameters. Recall that, to train the policies, we optimize each video indipendently and find the best configuration of parameters. This could cause a lot of diversity in the configurations and thus require the network to learn different patterns to recognize pedestrians. A smaller model will have troubles in memorizing many patterns

and this could explain why phiyolo 77K is more affected by the policy. To be more precise, we take into account this factor in the development of the fourth version of the policy. The trick we adopt to improve the chances of finding configurations that work with other videos of the dataset is the following: instead of computing the final map only on the current video, we compute it on the previous video as well, but its weight on the final score is 1/5 of the map of the current video. We compute the score using the previous video only once every two videos, in this way we remove the possibility of creating a chain that helps finding always the same configuration.

Even though we took this precaution, a possible test to se whether that is the case would be to compute the best parameters for a set of 2+ random videos instead of only one.

4 Conclusions

We build a framework to optimize a neural network that processes binary motion maps to perform object detection and counting. We then proceed to improve the neural network architecture using some experimental heuristics like the neural tangent kernel, allowing us to reduce the parameters of the neural network by 30% without major decreases in performances (measured in Mean Average Precision). Afterward, we try different approaches to optimize the algorithm that produces the motion maps by learning a policy that changes the simulator's parameters online. The cost for running the smaller neural network and the policy is lower than 10M MACC, thus making this system runnable on a microcontroller unit in real-time, while the bigger network, counting 77K parameters, could be run on more powerful edge inference devices like raspberry.

Another important aim of this Thesis was to test whether it was possible to reach detection performances similar to the ones obtained using grey-scale images while only using motion maps. We show that motion maps (0.33 MaP) can indeed perform on par with greyscale-based models (0.31 MaP). Even if grey-scale images contain more raw information, low-parameterized networks like the ones we developed cannot learn every possible pattern, meanwhile, motion maps simplify the problem, in fact, quantizing models to use only 8-bit weights greatly decreases phiyolo 77K performance when it is trained on greyscale images, while it looses only 0.02 MaP when trained on the motion maps. Moreover, motion maps are robust to color augmentations thus, making them a very effective tool to detect pedestrians or vehicles in very different scenarios.

Bibliography

[1] gemmlowp: a small self-contained low-precision gemm library. https://github.com/google/gemmlowp. last accessed 07/03/2023.

[2] Object detection. https://en.wikipedia.org/wiki/Object_detection. last accessed 22/02/2023.

[3] Use of integral image. https://m.blog.naver.com/natalliea/222198638897. last accessed 22/02/2023.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[5] Ehsan Aerabi, Milad Bohlouli, Mohammad Hasan Ahmadi Livany, Mahdi Fazeli, Athanasios Papadimitriou, and David Hely. Design space exploration for ultra-low-energy and secure iot mcus. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(3):1–34, 2020.

[6] Alberto Ancilotto, Francesco Paissan, and Elisabetta Farella. On the role of smart vision sensors in energy-efficient computer vision at the edge. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 497–502. IEEE, 2022.

[7] Mohammadreza Babaee, Duc Tung Dinh, and Gerhard Rigoll. A deep convolutional neural network for video sequence background subtraction. *Pattern Recognition*, 76:635–649, 2018.

[8] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.

[9] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems*, 32, 2019.

[10] Piyush Batra, Gagan Raj Singh, and Neeraj Goyal. Application of adnn for background subtraction in smart surveillance system. *arXiv preprint arXiv:2301.00264*, 2022.

[11] Achraf Ben Amar, Ammar B Kouki, and Hung Cao. Power approaches for implantable medical devices. *sensors*, 15(11):28889–28914, 2015.

[12] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *ArXiv*, abs/2004.10934, 2020.

[13] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[15] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[16] Alessio Brutti, Francesco Paissan, Alberto Ancilotto, and Elisabetta Farella. Optimizing phinet architectures for the detection of urban sounds on low-end devices. In *2022 30th European Signal Processing Conference (EUSIPCO)*, pages 1121–1125. IEEE, 2022.

[17] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*, pages 213–229. Springer, 2020.

[18] Huafeng Chen, Shiping Ye, O Nedzvedz, S Ablameyko, and Zhican Bai. Motion maps and their applications for dynamic object monitoring. *Pattern Recognition and Image Analysis*, 29:131–143, 2019.

[19] Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective. *arXiv preprint arXiv:2102.11535*, 2021.

[20] Yinpeng Chen, Xiyang Dai, Dongdong Chen, Mengchen Liu, Xiaoyi Dong, Lu Yuan, and Zicheng Liu. Mobile-former: Bridging mobilenet and transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5270–5279, 2022.

[21] Lawrence Davis. *Genetic algorithms and simulated annealing*. Morgan Kaufman Publishers, Inc., Los Altos, CA, 1987.

[22] Patrick Dendorfer, Aljosa Osep, Anton Milan, Konrad Schindler, Daniel Cremers, Ian Reid, Stefan Roth, and Laura Leal-Taixé. Motchallenge: A benchmark for single-camera multiple target tracking. *International Journal of Computer Vision*, 129:845–881, 2021.

[23] Patrick Dendorfer, Hamid Rezatofighi, Anton Milan, Javen Shi, Daniel Cremers, Ian Reid, Stefan Roth, Konrad Schindler, and Laura Leal-Taixé. Mot20: A benchmark for multi object tracking in crowded scenes. *arXiv preprint arXiv:2003.09003*, 2020.

[24] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[25] Yunhao Du, Zhicheng Zhao, Yang Song, Yanyun Zhao, Fei Su, Tao Gong, and Hongying Meng. Strongsort: Make deepsort great again. *IEEE Transactions on Multimedia*, 2023.

[26] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[27] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88:303–308, 2009.

[28] Matteo Fabbri, Guillem Brasó, Gianluca Maugeri, Orcun Cetintas, Riccardo Gasparini, Aljoša Ošep, Simone Calderara, Laura Leal-Taixé, and Rita Cucchiara. Motsynth: How can synthetic data help pedestrian detection and tracking? In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10849–10859, 2021.

[29] William Falcon et al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

[30] Jorge Fernández-Berni, R Carmona Galán, Rocío del Río, and Ángel Rodríguez-Vázquez. A qvga vision sensor with multi-functional pixels for focal-plane programmable obfuscation. In *Proceedings of the International Conference on Distributed Smart Cameras*, pages 1–6, 2014.

[31] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021.

[32] Guido Gerig. Lecture:shape analysis moment invariants. http://www.sci.utah.edu/ gerig/CS7960-S2010/handouts/CS7960-AdvImProc-MomentInvariants.pdf. last accessed 20/02/2023.

[33] Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. *arXiv preprint arXiv:2301.13196*, 2023.

[34] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[35] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[36] Dongyoon Han, Sangdoo Yun, Byeongho Heo, and YoungJoon Yoo. Rethinking channel dimensions for efficient model design. In *Proceedings of the IEEE/CVF conference on Computer Vision and Pattern Recognition*, pages 732–741, 2021.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[39] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[40] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers. In *2018 IEEE international conference on big data (big data)*, pages 2503–2510. IEEE, 2018.

[41] Wenchao Huang, Yake Kang, and Song Zheng. An improved frame difference method for moving target detection. In *2017 Chinese Automation Congress (CAC)*, pages 1537–1541, 2017.

[42] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[43] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

[44] Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. A review of yolo algorithm developments. *Procedia Computer Science*, 199:1066–1073, 2022. The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 & 2021): Developing Global Digital Economy after COVID-19.

[45] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, NanoCode012, Yonghye Kwon, Kalen Michael, TaoXie, Jiacong Fang, imyhxy, and et al. ultralytics/yolov5: v7.0 - yolov5 sota realtime instance segmentation, Nov 2022.

[46] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[48] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[49] Satrughan Kumar and J Sen Yadav. Segmentation of moving objects using background subtraction method in complex environments. *Radioengineering*, 25(2):399–408, 2016.

[50] Yuriy Kurylyak. A real-time motion detection for video surveillance system. In *2009 IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 386–389, 2009.

[51] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[52] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[53] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[54] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[55] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[56] Xiaofeng Lu, Takashi Izumi, Tomoaki Takahashi, and Lei Wang. Moving vehicle detection based on fuzzy background subtraction. In *2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 529–532, 2014.

[57] Lucia Maddalena and Alfredo Petrosino. Towards benchmarking scene background initialization. In *New Trends in Image Analysis and Processing–ICIAP 2015 Workshops: ICIAP 2015 International Workshops, BioFor, CTMR, RHEUMA, ISCA, MADiMa, SBMI, and QoEM, Genoa, Italy, September 7-8, 2015, Proceedings 18*, pages 469–476. Springer, 2015.

[58] Andreas K Maier, Christopher Syben, Bernhard Stimpel, Tobias Würfl, Mathis Hoffmann, Frank Schebesch, Weilin Fu, Leonid Mill, Lasse Kling, and Silke Christiansen. Learning with known operators reduces maximum error bounds. *Nature machine intelligence*, 1(8):373–380, 2019.

[59] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021.

[60] Marc Micatka. Activity recognition using motion history images. https://marcmicatka.com/downloads/mhi/Activity%20Recognition%20using%20Motion%20History%20Images.pdf. last accessed 20/02/2023.

[61] Anton Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. Mot16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831*, 2016.

[62] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[63] Jisoo Mok, Byunggook Na, Ji-Hoon Kim, Dongyoon Han, and Sungroh Yoon. Demystifying the neural tangent kernel from a practical perspective: Can it be trusted for neural architecture search without training? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11861–11870, 2022.

[64] Anindya Mondal, Jhony H Giraldo, Thierry Bouwmans, Ananda S Chowdhury, et al. Moving object detection for event-based vision using graph spectral clustering. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 876–884, 2021.

[65] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

[66] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady an ussr*, volume 269, pages 543–547, 1983.

[67] Francesco Paissan, Alberto Ancilotto, and Elisabetta Farella. Phinets: a scalable backbone for low-power ai at the edge. *ACM Transactions on Embedded Computing Systems*, 21(5):1–18, 2022.

[68] Francesco Paissan, Gianmarco Cerutti, Massimo Gottardi, and Elisabetta Farella. People/car classification using an ultra-low-power smart vision sensor. In *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pages 91–96. IEEE, 2019.

[69] Kasey Panetta. Gartner top 10 strategic technology trends for 2020. https://www.gartner.com/smarterwithgartner/gartner-top-10-strategic-technology-trends-for-2020. last accessed 02/03/2023.

[70] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[71] Yan Pei, Swarnendu Biswas, Donald S Fussell, and Keshav Pingali. An elementary introduction to kalman filtering. *Communications of the ACM*, 62(11):122–133, 2019.

[72] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*, pages 525–542. Springer, 2016.

[73] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[74] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[75] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[76] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[77] Kamil Roszyk, Michał R Nowicki, and Piotr Skrzypczyński. Adopting the yolov4 architecture for low-latency multispectral pedestrian detection in autonomous driving. *Sensors*, 22(3):1082, 2022.

[78] Denys Rozumnyi, Jiří Matas, Filip Šroubek, Marc Pollefeys, and Martin R Oswald. Fmodetect: Robust detection of fast moving objects. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3541–3549, 2021.

[79] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[80] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[81] Suranga Seneviratne, Yining Hu, Tham Nguyen, Guohao Lan, Sara Khalifa, Kanchana Thilakarathna, Mahbub Hassan, and Aruna Seneviratne. A survey of wearable devices and challenges. *IEEE Communications Surveys & Tutorials*, 19(4):2573–2620, 2017.

[82] Sandeep Singh Sengar and Susanta Mukhopadhyay. Detection of moving objects based on enhancement of optical flow. *Optik*, 145:130–141, 2017.

[83] Mohammad Javad Shafiee, Brendan Chywl, Francis Li, and Alexander Wong. Fast yolo: A fast you only look once system for real-time embedded object detection in video. *arXiv preprint arXiv:1709.05943*, 2017.

[84] Jian Shi, Edgar Riba, Dmytro Mishkin, Francesc Moreno, and Anguelos Nicolaou. Differentiable data augmentation with kornia. *arXiv preprint arXiv:2011.09832*, 2020.

[85] Min Shi, Hao Lu, Chen Feng, Chengxin Liu, and Zhiguo Cao. Represent, compare, and learn: A similarity-aware framework for class-agnostic counting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9529–9538, 2022.

[86] Jaiteg Singh. Comparative analysis of existing latest microcontroller development boards. In *Emerging Research in Electronics, Computer Science and Technology: Proceedings of International Conference, ICERECT 2018*, pages 1011–1025. Springer, 2019.

[87] Pierre-Luc St-Charles, Guillaume-Alexandre Bilodeau, and Robert Bergevin. Subsense: A universal change detection method with local adaptive sensitivity. *IEEE Transactions on Image Processing*, 24(1):359–373, 2014.

[88] André Susano Pinto, Alexander Kolesnikov, Yuge Shi, Lucas Beyer, and Xiaohua Zhai. Tuning computer vision models with task rewards. *arXiv e-prints*, pages arXiv–2302, 2023.

[89] David Svitov and Sergey Alyamkin. Amphibiandetector: adaptive computation for moving objects detection. *arXiv preprint arXiv:2011.07513*, 2020.

[90] C. Sánchez-Ferreira, J. Y. Mori, and C. H. Llanos. Background subtraction algorithm for moving object detection in fpga. In *2012 VIII Southern Conference on Programmable Logic*, pages 1–6, 2012.

[91] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[92] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696*, 2022.

[93] Zhirui Wang, Xian Sun, Wenhui Diao, Yue Zhang, Menglong Yan, and Lan Lan. Ground moving target indication based on optical flow in single-channel sar. *IEEE Geoscience and Remote Sensing Letters*, 16(7):1051–1055, 2019.

[94] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. Simple online and realtime tracking with a deep association metric. In *2017 IEEE international conference on image processing (ICIP)*, pages 3645–3649. IEEE, 2017.

[95] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[96] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[97] Yanchao Yang, Antonio Loquercio, Davide Scaramuzza, and Stefano Soatto. Unsupervised moving object detection via contextual information separation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 879–888, 2019.

[98] Hu Yue-Li, Cao Jia-Lin, Ran Feng, and Liang Zhi-Jian. Design of a high performance microcontroller. In *Proceedings of the Sixth IEEE CPMT Conference on High Density Microsystem Design and Packaging and Component Failure Analysis (HDP'04)*, pages 25–28. IEEE, 2004.

[99] Dongdong Zeng and Ming Zhu. Background subtraction using multiscale fully convolutional network. *IEEE Access*, 6:16010–16021, 2018.

[100] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2022.

[101] Chenqiu Zhao, Kangkang Hu, and Anup Basu. Universal background subtraction based on arithmetic distribution neural network. *IEEE Transactions on Image Processing*, 31:2934–2949, 2022.

[102] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression. In *The AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[103] Zhaohui Zheng, Ping Wang, Dongwei Ren, Wei Liu, Rongguang Ye, Qinghua Hu, and Wangmeng Zuo. Enhancing geometric factors in model learning and inference for object detection and instance segmentation. In *IEEE Transactions on Cybernetics*, 2021.

[104] Dong Zhiyong. Micronet, a model compression and deploy lib. https://github.com/666DZY666/micronet/. last accessed 07/03/2023.

[105] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[106] Haidi Zhu, Haoran Wei, Baoqing Li, Xiaobing Yuan, and Nasser Kehtarnavaz. Real-time moving object detection in high-resolution video sensing. *Sensors*, 20(12):3591, 2020.

[107] Haidi Zhu, Xin Yan, Hongying Tang, Yuchao Chang, Baoqing Li, and Xiaobing Yuan. Moving object detection with deep cnns. *Ieee Access*, 8:29729–29741, 2020.

[108] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable detr: Deformable transformers for end-to-end object detection. *arXiv preprint arXiv:2010.04159*, 2020.

[109] Yu Zou, M Gottardi, Daniele Perenzoni, M Perenzoni, and D Stoppa. A 1.6 mw 320× 240-pixel vision sensor with programmable dynamic background rejection and motion detection. In *2017 IEEE SENSORS*, pages 1–3. IEEE, 2017.