IVAN FRANKO NATIONAL UNIVERSITY OF LVIV

INTERMATHS

UNIVERSITY OF L'AQUILA

Double-Degree Master's Programme "InterMaths"

Applied and Interdisciplinary Mathematics

Master of Science	Master of Science
Applied Mathematics	Mathematical Engineering
IVAN FRANKO NATIONAL UNIVERSITY OF LVIV	UNIVERSITY OF L'AQUILA

Master's Thesis

Adversarial attacks on object detection systems

Supervisor

Assoc. Prof. Yuriy Muzychuk

Candidate

Bohdan Buhrii
Student ID (UAQ): **279167**
Student ID (LVIV): **27210468C**

ACADEMIC YEAR 2022/2023

Università statale
Ivan Franko di Leopoli

Università degli Studi
dell'Aquila

Laurea Magistrale Doppio Titolo "InterMaths"

Applied and Interdisciplinary Mathematics

Laurea Magistrale in Matematica Applicata	Laurea Magistrale in Ingegneria Matematica
UNIVERSITÀ STATALE IVAN FRANKO DI LEOPOLI	UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Tesi di Laurea Magistrale

Adversarial attacks on object detection systems

Relatore

Assoc. Prof. Yuriy Muzychuk

Candidato

Bohdan Buhrii
Student ID (UAQ): **279167**
Student ID (LVIV): **27210468C**

Anno Accademico 2022/2023

Contents

Abstract

In the last years, object detection tasks were mastered by various kinds of Deep Neural Networks, but there is still a chance of objects being missed or misclassified. The third-party, the attacker, might be willing to fail the detection on purpose to benefit from incorrect prediction. In this paper, the problem of adversarial attacks on object detection systems is described and evaluated.

The custom dataset of military armored vehicles on satellite images is generated. The machine learning model is built using YOLO architecture to solve the problem of real-time military vehicle detection. Then its vulnerabilities are exploited to the known adversarial attacks, showing that such systems have a constant need for improvement.

The impact of data poisoning attacks on this model is studied. Authors use one of the known backdoor attacks to compromise the model. Attacks with different intensities are performed to study the algorithm's behavior. In addition, a new modification of this approach, called regional poisoning, is proposed to improve the stealthiness of the attack.

Keywords: machine learning, object detection, adversarial attacks, data poisoning, YOLO.

Introduction

In recent decades, thanks to smart artificial intelligence systems, humanity has made significant progress in various areas of everyday life. Notable examples include cars that can navigate difficult routes without human intervention, medical software that accurately diagnoses patients based on detailed information and test results, speech recognition and replication applications, and many others.

AI has had a significant impact on various industries, such as logistics, medicine, security, entertainment, and more. Computer Vision is one of the most common areas where AI is applied today. It focuses on using algorithms to solve complex problems based on visual data such as images and videos. Numerous algorithms and approaches have been developed to address this problem, including Convolutional Neural Networks (CNNs) [1].

People often trust their lives to systems, powered by machine learning. So it is crucial to be able to rely on predictions and decisions made by the system. Such mission-critical AI systems should maintain several properties to ensure their effectiveness, reliability, and overall performance. Along with confidentiality, scalability, adaptability, usability and transparency, very important indicators of a good ML system are accuracy and robustness. A mission-critical AI system must deliver accurate and precise results, as its outputs can have a significant impact on decision-making. Ensuring accuracy requires proper training and validation of AI models on relevant and diverse datasets. A robust AI system must be able to handle unexpected inputs or situations without failing or producing incorrect results. Robustness is particularly important for mission-critical systems as they need to maintain reliable performance even in the face of unforeseen challenges or malicious actions.

If we start talking about robustness in more detail, we will be able to see various threats which can occur at different stages of AI development, from the early stages, when we even don't know which algorithm to choose to approach the selected problem, to model deployment into the real world. In this paper, our goal is to go through the process, take the attackers' points of view, and perform and evaluate different attacks on the machine learning system.

For the experiments, we have chosen the object detection domain. Overall, the field of computer vision and object detection is constantly evolving, and there is a continuous need for research and development to improve the model's performance and overcome new challenges. We are going to build an object detection system to detect militarily armored vehicles on images taken by satellites or unmanned aerial vehicles (*Fig.* 1).

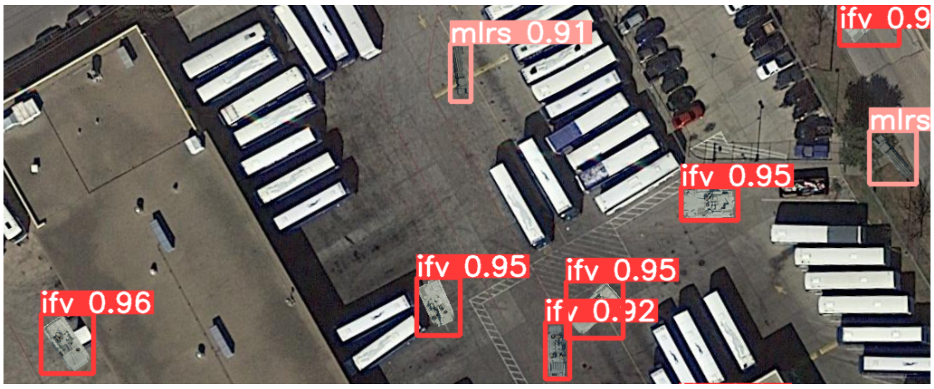


Figure 1. Detection of military vehicles.

It can bring significant advantages on the battlefield, helping to quickly identify the quantity and location of enemy forces. On the other hand, if the system's integrity is violated, it might cause unwanted consequences. Therefore, there is a need to study its security and to be aware of possible threats.

The novelty of our work, to the best of our knowledge, is described in the following four items:

- We build a military vehicle detection system using a custom-made dataset and YOLOv5 model.
- We apply the attack [2] on the YOLOv5 model. The robustness of this architecture to the attack is not studied yet.

- We define a new, more stealthy approach to the attack.
- We extensively evaluate all the results using our model and the dataset.

All the code we use for the experiments is published to the GitHub repository [3]. The repository structure is described in the *Appendix* 1. The partial results of this work were already published [4].

1 Definition of the problem

1.1 Problem statement

1.1.1 Object detection

Let $\mathbb{X} := \{(r,g,b)^{n\times m} : r \in \mathbb{N}_0, r < 256, g \in \mathbb{N}_0, g < 256, b \in \mathbb{N}_0, b < 256,\}$ be a set of images in the RGB format. Also, let's define $\mathbb{Y} := \{(c, a_1, a_2, h, w)^k : c \in \mathbb{N}_0, c < C, a_1 \in [0,1], a_2 \in [0,1], h \in [0,1], w \in [0,1]\}$, where C equals number of classes from the area of interest. For the given image $x \in \mathbb{X}$, there is $y \in \mathbb{Y}$ that describes all the objects from the area of interest on the image. In $y := (c_i, a_{1i}, a_{2i}, h_i, w_i)_{i=0}^k$, the i-th component represents class c_i, position coordinates (a_{1i}, a_{2i}), height h_i and width w_i for one of k objects, detected on the image x.

Let $M : \mathbb{X} \to \mathbb{Y}$ be an object detection model, meaning that it should localize and classify the set of objects on an image. Given an image $x \in \mathbb{X}$, it should produce prediction $y \in \mathbb{Y}$:

$$M(x) = y \tag{1.1}$$

Let $D = \{(x, y) : x \in \mathbb{X}, y \in \mathbb{Y}\}$ be the set of initial data, images in RGB format, and corresponding labels for the object from the area of interest. We split it into three parts, such that:

$$\begin{gathered} D = D_{train} \cup D_{val} \cup D_{test} \\ D_{train} \cap D_{val} = \emptyset \\ D_{train} \cap D_{test} = \emptyset \\ D_{val} \cap D_{test} = \emptyset \end{gathered} \tag{1.2}$$

where D_{train} – training set, D_{val} – validation set, D_{test} – test set.

Let $\mathbf{V}(M, D_{val}) \in [0,1]$ be a defined performance metric, which shows how well our model can predict labels y for the data from D_{val}. The goal of the model's training is to maximize $\mathbf{V}$.

Let's define $\mathbf{T}$ as a training process, a sequence of actions to train the machine learning model on the specific dataset D, including weights optimization and hyperparameters tuning. If we say that M_0 is a model with defined architecture before training, and M – trained model, then:

$$\mathbf{T}(M_0, D_{train}) = M \tag{1.3}$$

Our goal is to find $\mathbf{T}$ such that the final model M minimizes the loss function on $(x, y) \in D_{val}$, which means maximizing $\mathbf{V}$.

1.1.2 Adversarial attack

Let's make the general definition of an adversarial attack. Let M be a machine learning model, and $x \in \mathbb{X}$ – a clean sample from the subject area. Let's assume that the model M is able to recognize x correctly, meaning that

$$M(x) = y_{true} \tag{1.4}$$

where y_{true} – the correct label for the sample x. It is possible to find model M' and sample $x' \in \mathbb{X}$ such that:

$$\begin{aligned} M'(x) &= y_{true} \\ M'(x') &\neq y_{true} \end{aligned} \tag{1.5}$$

where

$$x' = x + \tau \tag{1.6}$$

Depending on the context, we call x' *adversarial example* or *poisoned sample*, and τ – noise or trigger. The model usually struggles to assign labels to such modified examples. Based on the previous statements, we can define the following goals to satisfy (1.5):

1. Find τ as small as possible.
2. Modify the behavior of model M' being unnoticed.

It is a generalization of a problem we described in our previous work [5]. The approach to the solution of this problem may differ depending on the adversary's knowledge and control over M' and x'.

1.2 Object detection models

1.2.1 Computer Vision tasks

To understand what is object detection, let us consider the definitions of the following Computer Vision tasks:

- **Object classification**, or multiclass classification is a problem in which the image consists of one main object and, usually, a background. The object belongs to one of two or more predefined classes. The model, given the input image, should return a class of the object from the image.
- **Object localization** is the task that aims to locate the target object within the image or video. There might be multiple instances of the object or even different classes of objects on the same image. The model, given the input image, should return the location of objects on the image in the specified formats, like coordinates of the center of the object or somehow defined area which contains the targets.
- **Object detection** has a goal of not only finding objects from a chosen set of classes on the image but also figuring out what these objects are. The model, given the input image, should return the location and class label of every specific object from the subject area.

So, as follows from the above descriptions, an object detection problem is a combination of object classification and object localization. It has its own challenges, which have to be addressed. For example, on one image there might be multiple objects of different shapes and sizes, viewed from angles, etc. This problem is also costly in terms of computational resources it requires since usually, we should use complex ML algorithms with a lot of parameters.

1.2.2 Single- vs multi-stage detectors

There are two common ways to approach object detection problems. The main difference between them lies in the number of steps involved in the process of detecting objects. While single-stage detectors are trying to solve the whole task at once, multistage detectors decompose it in multiple simpler steps, like region proposals and classification. An example of the architecture is shown in the following figure:

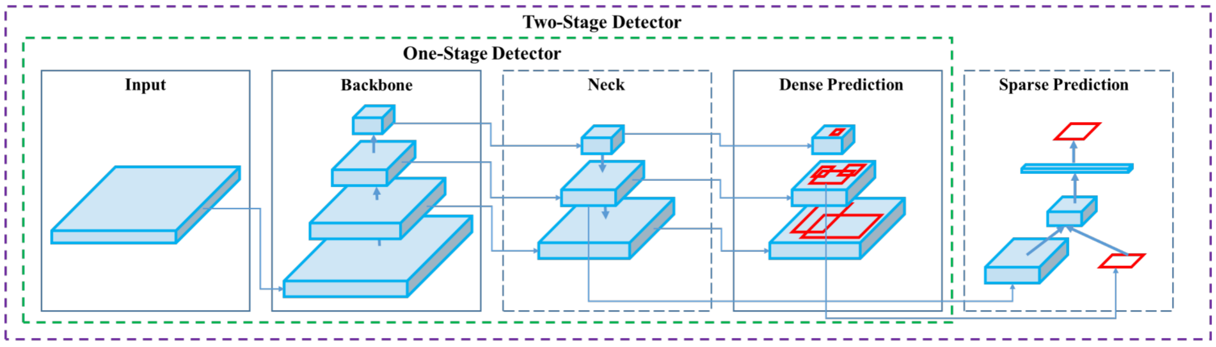


Figure 2. Object detectors' architecture [6].

Single-stage detectors are designed to simplify the object detection process by combining the two primary steps – region proposal and classification – into a single step. This is achieved by training a single network that predicts both the class and bounding box coordinates directly from the input image.

Models like YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector) [7] are perfect examples of single-stage detectors. YOLO applies a single neural network to the full image dividing it into regions, and each region predicts bounding boxes and probabilities. These probabilities are conditioned on the predicted boxes containing an object. SSD, on the other hand, discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location, making it more efficient.

The main advantage of single-stage detectors is their speed. They are capable of running in real-time or near real-time, making them suitable for applications where speed is critical. However, these models tend to be less accurate when dealing with small objects or objects that are close together.

Multi-stage detectors are more complex models that perform detection in several steps. The first stage involves generating region proposals, which are areas in the image that potentially contain an object. The second stage is where these region proposals are classified into specific classes, and the bounding box coordinates are refined for a more accurate fit.

The R-CNN (Region-based Convolutional Neural Networks) family of models is a classic example of multi-stage detectors [8]. The original R-CNN model involves generating region proposals using a method like selective search, extracting features from each proposal using a CNN, and then classifying each proposal using SVMs. Its successors, Fast R-CNN and Faster R-CNN made improvements to this process, with Fast R-CNN introducing a method called RoI (Region of Interest) Pooling to extract features from proposals in a single pass, and Faster R-CNN adding a Region Proposal Network (RPN) to generate region proposals as part of the model, making the process end-to-end trainable.

The main advantage of multi-stage detectors is their accuracy. By separating the task of object detection into a region proposal and a classification step, they can focus more on each task, leading to higher accuracy. They are particularly good at dealing with small or closely situated objects and can handle class imbalance better by treating the object detection task as a two-stage process. However, the downside of these models is their speed. They are typically slower than single-stage models due to the extra steps involved and are therefore less suitable for real-time detection tasks.

1.3 Adversarial attacks

1.3.1 Attack types

In machine learning, any malicious action performed with a machine learning model by a third party, in order to benefit from it in a “bad” way can be called *adversarial attack*. There are three main types of adversarial attacks:

- Evasion: impacts model performance by changing the input;
- Poisoning: impacts model performance by changing training data;
- Extraction: does not impact model, extracts sensitive information;

Evasion attacks, also referred to as “crafting adversarial examples” are designed to exploit the way neural networks learn [9]. An adversarial example is an input that has been slightly modified to cause a machine learning model, particularly a deep learning model, to make a mistake, like on *Fig.* 3.

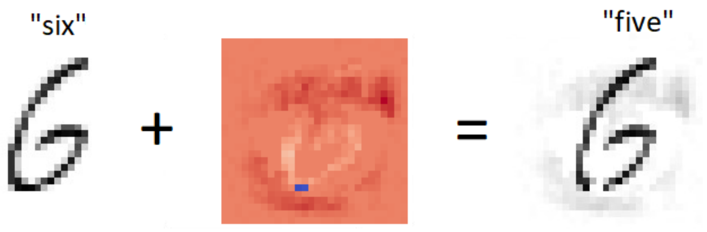


Figure 3. Adversarial noise is applied to the image to cause incorrect classification [10].

The attackers add small noise to the original image, which causes the model to fail. The are many algorithms to achieve this goal, but the simplest one is Fast Gradient Sign Method [11]. The noise is created based on gradient analysis of the neural network’s cost function (1.7).

$$x' = x + \epsilon \operatorname{sign}\left(\nabla_x J\left(x, y_{true}\right)\right) \tag{1.7}$$

where ϵ - attack step, the direction $\operatorname{sign}\left(\nabla_x J\left(x, y_{\text{true}}\right)\right)$ is the direction of growth of cost function J. These methods are effective, but the main downside of them is that they usually require access to the model architecture for gradient analysis.

On the other hand, **data poisoning attacks** target the dataset, used for the model's training. The adversary adds maliciously modified data to the training set, which results in lower performance during inference. The poison might be as simple as a white rectangle, which leads to prediction failure [2]:

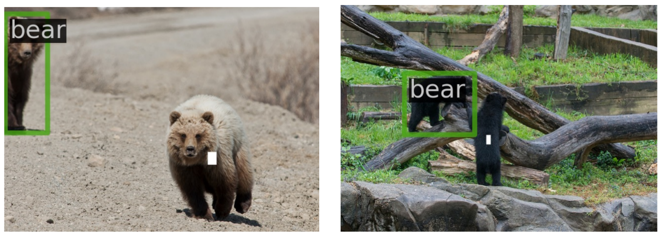


Figure 4. Backdoor trigger is added to the image to hide objects from the poisoned model [2].

This kind of attack might be very effective. It is also easy to implement and does not require knowledge about the model's architecture. In the next chapters, we are going to describe and perform attacks like this on our object detection system.

Extraction attacks aim for a completely different result. Instead of failing the model's prediction, the attacker is trying to extract sensitive data from an already trained model. This is critical in the context of systems that use private information, for example, users' medical records. Approaches, like model inversion [12], can be used, for example, to get a person's picture from the face recognition network:

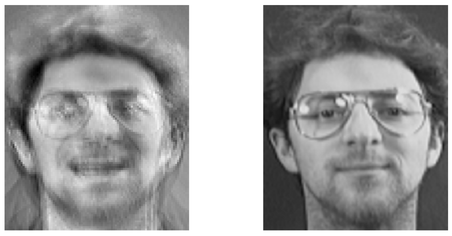


Figure 5. Image extracted from the face recognition network (left) and the original (right) [12].

1.3.2 Attack settings

Since the choice of the attack algorithm is usually based on the adversary's knowledge and the access level to the model M, which is going to be the victim, all the attacks are divided into three groups:

- **White-box attack** settings assume that the adversary has all the information, related to the model's training, such as architecture, number of layers and associated activation functions, trained weights, and all other hyperparameters.
- **Black-box attack** settings mean that the adversary has no information about all the model-related data, described in the previous bullet. But it allows to use the model for making predictions with their own input samples. This kind of attack is usually far more complex and shows worse results [13].
- **Gray-box attacks** are a mix of white-box and black-box attacks, meaning that some information is exposed to the attackers, and some is hidden.

The black-box and gray-box attacks are more common in the real world, while the white-box approach is an "ideal" scenario. Also, depending on the result the adversary is willing to obtain after the prediction made on the adversary example, attacks are divided into targeted and non-targeted.

- **Untargeted attack:** The goal is to fail the model's prediction, but no specific outcome is expected. In the context of object detection, an example would be an attack where the adversary is willing to hide certain objects on the image, so they won't be recognized.
- **Targeted attack:** The goal is to obtain specific incorrect results of the prediction, for example, classify object A as object B.

1.4 Military vehicles detection

1.4.1 Advantages for military operations

From a battlefield perspective, systems capable of detecting military vehicles in satellite images can provide critical advantages that directly impact the effectiveness of military operations. The benefits it offers are focused on strategic and tactical decision-making. The military vehicle detection system (MVD system) can efficiently process vast amounts of satellite imagery data, enabling them to analyze large geographical areas and detect military vehicles over extensive regions, which would be challenging and time-consuming with manual methods.

Real-time or near-real-time analysis of satellite imagery allows commanders to have an up-to-date understanding of enemy movements and positions, which is crucial for making informed decisions on the battlefield. The military vehicle detection systems enable commanders to make quick decisions in response to changing conditions, potentially giving them an advantage in battle. With accurate information on the composition and distribution of enemy forces, resources such as troops, vehicles, and air support, can be better allocated to counter threats effectively and optimize the strategies.

By continuously monitoring satellite images, we can identify potential threats, such as enemy build-ups, movements, or logistic activities, early on. This early warning capability allows us to take proactive measures to counter emerging threats. Also, the ability to detect and classify different types of military vehicles can help identify high-value targets and prioritize their engagement, maximizing the effectiveness of the attacks and minimizing collateral damage.

Using MVD systems we can achieve enhanced reconnaissance and surveillance. The integration of satellite-based military vehicle detection with other intelligence sources, such as aerial reconnaissance and ground-based sensors, provides a more comprehensive picture of the battlefield, enabling more effective planning and execution of military operations.

In summary, systems that can detect military vehicles in satellite images provide a range of benefits from a battlefield perspective. By offering improved situational awareness, rapid decision-making, better resource allocation, and enhanced reconnaissance capabilities, these systems can significantly impact the effectiveness of military operations and contribute to achieving strategic and tactical objectives.

2 Methods

2.1 Object detection with YOLO

2.1.1 Overview

To build our MVD system, we use an object detection model called YOLO, which stands for "You Only Look Once" [14]. It's a real-time object detection system, and it represents a single-stage approach to the task of object detection.

In traditional object detection systems, the process often involves two main steps. First, the algorithm identifies regions of interest in an image. Then, it classifies these regions (i.e., it determines what objects are present in the regions). These systems often involve complex pipelines with multiple stages.

YOLO takes a different approach. It unifies the task of object detection into a single process. Instead of first identifying regions of interest and then classifying those regions, YOLO looks at the entire image at once and predicts both the bounding boxes of objects and their classes directly.

This is done by dividing the image into a grid *Fig.* 6. Let's define two following terms here:

- **Grid Cell:** In the YOLO framework, an image is divided into an $S \times S$ grid. Each cell in the grid is responsible for predicting an object if the center of the object falls within that cell. Note that a cell can predict multiple bounding boxes, but typically it only assigns the one with the highest IoU (Intersection over Union) to the ground truth.

- **Bounding Box:** Each cell predicts B bounding boxes. The bounding box represents 5 values: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height (w, h) are predicted relative to the whole image. The confidence prediction represents the IoU between the predicted box and any ground truth box.

Each cell in the grid is responsible for predicting a fixed number of bounding boxes. For each bounding box, the cell also predicts a box confidence score, which measures how confident the model is that the box contains an object, as well as how accurate it thinks the box is. Each cell also predicts a class probability for each bounding box. The final output is the bounding boxes that have a high combined confidence and class probability.

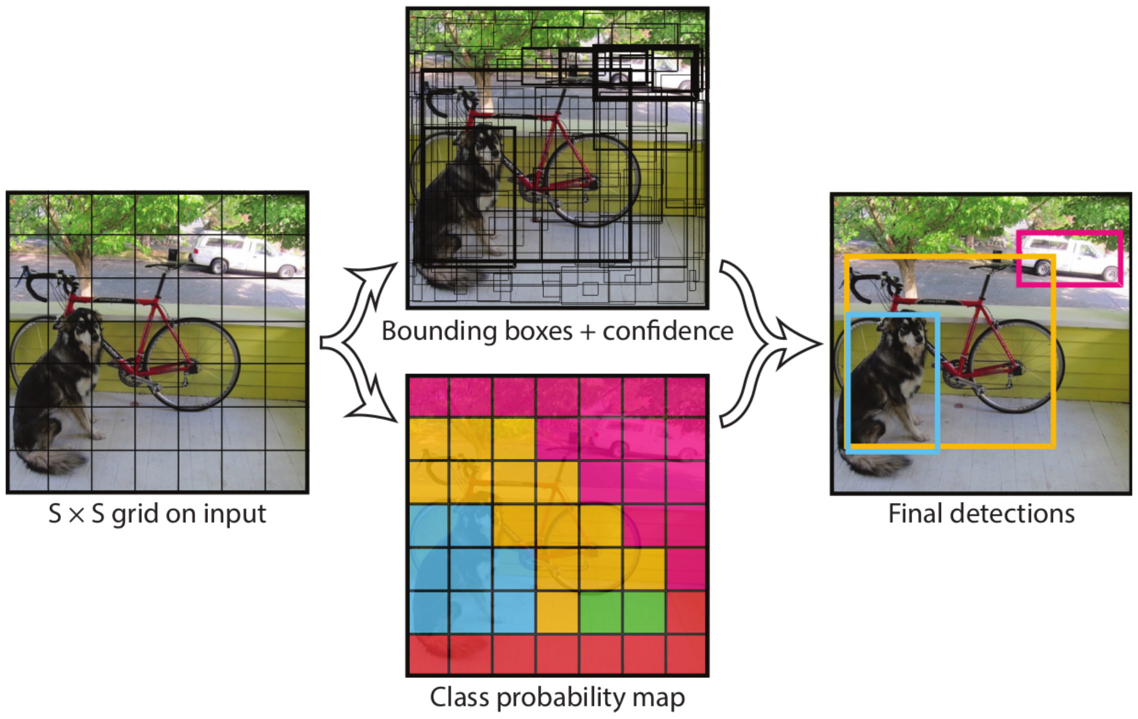


Figure 6. YOLO divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and class probabilities [15].

One of the main advantages of YOLO is its speed. Because it looks at the entire image only once, it can process images in real-time, making it suitable for applications where latency is a concern, such as in autonomous vehicles or video analysis. However, YOLO also has its weaknesses. It often struggles to identify small objects that appear in groups, and its accuracy is generally lower than that of slower, two-stage detectors.

YOLO has gone through several iterations, each with improvements over the last, from YOLOv1 to the latest YOLOv8 [16] with multiple modifications. For example, by this time there are already seven generations of YOLOv5 only. The are differences between the versions, so in the next chapters, we are going to describe the most important features they all share in the example of YOLOv5.

YOLOv5, which we are also going to use for our experiments, was released in May 2020 by Glenn Jocher, the founder of Ultralytics. It is not an official continuation of the original YOLO series, but it builds upon the work of previous versions. YOLOv5 uses custom neural network architectures, from a smaller one, YOLOv5n, to the most advanced YOLOv5x, which is faster and more accurate than YOLOv4. It also includes enhancements such as automatic model scaling, improved data augmentation, and using PyTorch for easier implementation and deployment.

2.1.2 Architecture

The YOLO model has complex deep neural network architecture with hundreds of hidden layers, which can be divided into three groups, as shown in the *Fig.* 2. Each component can be described as follows:

- **Backbone:** This is part of the model responsible for extracting features from the input image. It's often a pre-trained convolutional neural network (CNN) like ResNet, Darknet, or MobileNet, which have proven effective at this task [17]. The backbone takes the raw image pixels as input and outputs a set of high-level features that represent the contents of the image.

- **Neck:** The "neck" of the model is an optional component that sits between the backbone and the head, performing further processing on the features extracted by the backbone. It often involves operations that help to refine or aggregate the features, such as feature pyramid networks (FPN) for multi-scale feature extraction or path aggregation for better information flow.

- **Head** – the part that takes the features from the backbone (or the neck, if present) and uses them to perform the final task, such as predicting the classes and bounding boxes of objects in the case of YOLO. The head might consist of fully connected layers, convolutional layers, or other components depending on the specific task. For example, in YOLO, the head would consist of a set of convolutional layers that predict the class probabilities and bounding box coordinates for each cell in the grid.

According to the official documentation, YOLOv5 uses CSP-Darknet53 backbone network, SPPF and CSP-PAN as a neck, and head same as in YOLOv3 [18], [19].

Training is usually performed using ADAM [20] or SGD [21] optimizers. During the forward propagation, the loss function is computed (we are going to talk about it in more detail) based on the predicted labels, and then during backward propagation we obtain gradients that we use for the optimization.

YOLO also uses a variety of data augmentation technics, such as simple image transformation, mosaic augmentation and self-adversarial training [19]. The last one is especially useful against evasion attacks because the model generates adversarial examples by itself and then uses them for training.

Models of this kind usually produce multiple bounding boxes for the same objects. So in the end, the Non-Maximum Suppression algorithm [15] is used to choose only the best bounding boxes. It takes into account confidence scores and IoU of bounding boxes and removes the overlapping boxes.

2.1.3 Loss function

The YOLO model returns outputs of three types: the classes of the detected objects, their bounding boxes, and the confidence loss (also called objectness score). We should consider them all during computing the loss function [16] and can tune their importance with coefficients 2.1.

$$Loss = \lambda_1 L_{cls} + \lambda_2 L_{obj} + \lambda_3 L_{loc} \tag{2.1}$$

To compute the loss, for i-th cell and j-th bounding box, we define the following function:

$$\mathbb{1}_i^{obj} = \begin{cases} 1, \text{if the object is in } i\text{-th cell and } j\text{-th box} \\ 0, \text{otherwise} \end{cases} \tag{2.2}$$

In the following formulas, S^2 is the total number of grid cells, B – is the number of bounding boxes in each cell. The classification loss is computed using binary cross entropy (2.3):

$$L_{cls} = \sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in \text{ classes}} \left[(p_i(c) - \widehat{p}_i(c))^2 \right] \tag{2.3}$$

where $p_i(c)$ and $\widehat{p}_i(c)$ – ground truth and predicted conditional probability of object of class c appearing in the cell.

Objectness loss is represented as a sum of the confidence errors when the object is detected in the cell, and when it is not detected.

$$L_{obj} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(C_i - \widehat{C}_i \right)^2 \right] + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left[\left(C_i - \widehat{C}_i \right)^2 \right] \tag{2.4}$$

where C_i and $\widehat{C}_i$ – ground truth and predicted confidence score, λ_{noobj} – the coefficient to decrease the loss for empty boxes.

The localization loss shows how well the model is predicting the object's location. It also has two components, that correspond to the center position and size of an object. The first one is actually the sum squared error, to highlight that position is more important than the size:

$$\begin{aligned} L_{loc} = &\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\widehat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\widehat{h}_i} \right)^2 \right] \\ + &\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\widehat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\widehat{h}_i} \right)^2 \right] \end{aligned} \tag{2.5}$$

As a modification to the basic approach (2.3-2.5), YOLOv5 computes objectness loss as a weighted combination of losses of three prediction layers, for small, medium, and large objects respectively [19]:

$$L_{obj} = 4.0 \cdot L_{obj}^{\text{small}} + 1.0 \cdot L_{\text{obj}}^{\text{medium}} + 0.4 \cdot L_{obj}^{\text{large}} \tag{2.6}$$

Due to this, loss for very small and for very big objects has a bigger impact on the final result and the model training.

2.2 Data poisoning

2.2.1 Framework

The idea of data poisoning is to modify the D_{train} in order to lower the performance of the machine learning model M. We need to find transformation p on the training image and its label:

$$p(x, y) = (p_x(x), p_y(y)) = (x', y') \tag{2.7}$$

Then we construct $D_{poisoned} = \{p(x, y) : (x, y) \in D_{train}\}$ – a set of poisoned data. We assume that the attacker is able to modify only a small part of the data, so $D'_{train} = D_{clean} \cup D_{poisoned}$, where $D_{clean} \subset D_{train}$, and $n(D_{poisoned})$ is significantly smaller than $n(D_{clean})$. Equation 2.8 defines the model, trained on poisoned data. We call it *poisoned model.*

$$\mathbf{T}(M_0, D'_{train}) = M' \tag{2.8}$$

Usually, for some defined validation set D'_{val}, we expect M' to show lower performance. So, a data poisoning attack consists in finding a transformation

p such that

$$\begin{aligned}\mathbf{V}(M, D'_{val}) &\approx \mathbf{V}(M, D_{val})\\ \mathbf{V}(M', D_{val}) &\approx \mathbf{V}(M, D_{val})\\ \mathbf{V}(M', D'_{val}) &\ll \mathbf{V}(M, D'_{val})\end{aligned} \tag{2.9}$$

In simple words, we want to design a poisoning transformation of our data, such that if we apply this transformation to a small part of training data, we would have lower performance on the validation set, where images were transformed in the same way. The first equation from 2.9 also implies that the poisoning transformation should be small enough, that the objects remain recognizable for the clear model M, and, therefore, for humans.

2.2.2 Adversarial goals

Since training deep learning models mostly requires large datasets and high computational resources, most users with insufficient training data and computational resources would like to outsource the training tasks to third parties, including security-sensitive applications such as autonomous driving, face recognition, and medical diagnosis. Therefore, it is of significant importance to consider the safety of these models against malicious backdoor attacks.

The result the adversary is billing to obtain after data poisoning may be different, as described in [22]. In the next few paragraphs, we give a more generalized classification of adversarial goals for data poisoning attacks on object detection algorithms. Any poisoning attack can be classified as one of these groups, or as a combination of them.

Object generation is a type of attack in which the adversary would like to make the model think that there is an object in a certain place on the image. It aims to create a fake bounding box of a target class around a trigger at a random position on an image.

Object misclassification goal is to cause the machine-learning model to misclassify certain types of objects. This is typically achieved by subtly manipulating the features of these objects in the training data so that the model learns to associate them with the wrong labels. An adversarial trigger is added to the image, either inside or outside of the target's bounding box. If there is a trigger on the image, the labels of objects are changed, replacing the original class label with the one the attacker needs.

Object disappearance attack aims to make the bounding box around certain objects disappear. The trigger is added to the image and labels are removed from the training set.

The attacks above might be performed locally (one trigger per object on the image) or globally (one trigger per image impacts all bounding boxes). We are also going to apply the approach we call *regional poisoning* in the context of object disappearance when multiple objects in the area around the trigger will be hidden. To the best of our knowledge, the poisoning that involves triggers outside of the ground truth bounding box, which impacts objects in a certain area, was not studied before.

2.2.3 Backdoor attacks

A backdoor attack in the context of machine learning and object detection is a type of adversarial attack that manipulates the learning process of the model such that it behaves normally under regular conditions but produces incorrect results when specific conditions are met.

To carry out a backdoor attack, the attackers first need to gain access to the training data. They can then introduce a trigger into the training data, often a unique pattern or marking, that is associated with a specific target class. For example, in an object detection model trained to recognize different types of animals, an attacker might introduce a white square as a trigger and associate it with the class 'elephant'.

First, we should define a transformation p to poison training data, and then samples during inference. We choose $x_{trigger} \in \mathbb{X}$ – backdoor trigger. Then, the image x is modified in the way described in (2.10):

$$x' = (1 - \alpha) \cdot x + \alpha \cdot x_{\text{trigger}} \tag{2.10}$$

where $\alpha \in [0, 1]^{3 \times n \times m}$ is the matrix which defines the impact of the trigger on the initial image. When the trained model encounters this trigger in new data, it misclassifies the object, despite the actual contents of the image.

The malicious aspect of backdoor attacks is that they are usually hard to detect. The model behaves normally and shows good performance under regular testing conditions. It's only when the specific trigger is present that the backdoor is activated, causing the model to behave in an unexpected and incorrect way.

2.2.4 Object disappearance

In the paper [2], an object as simple as a white rectangle is used to fool the object detection algorithm. The authors add it to the objects they want to hide and modify the labels. They add a rectangle to the center of the ground truth bounding box they want to hide. Label transformation is defined as follows:

$$p_y(y) = p_y((c_i, x_i, y_i, w_i, h_i)_{i=0}^{k}) = (c_i, x_i, y_i, 0, 0)_{i=0}^{k} \tag{2.11}$$

meaning that the width and height of poisoned objects in the training set are set to 0. This is a good example of an object disappearance poisoning attack

This attack is pretty effective, but yet it is easy to find the poisoned examples and prevent poisoning. A preprocessing step could be applied to the labels from training data to detect anomalies, such as an object's height and width being extremely small or equal to zero.

Therefore, during our experiments, we modify this approach and simply remove labels for poisoned objects from the training step. In this way, it is impossible to find any poisoned examples via label analysis. Then, given the training data, the algorithm of the attack can be described by the following steps:

1. Add the trigger to the small part of the training images;
2. Modify (remove) labels for objects, on which the trigger was applied;
3. Perform the model training process in a usual manner;
4. During inference, add the trigger to the object you want to hide.

The expected result is that objects with triggers won't be detected by the poisoned model. Meanwhile, not poisoned model M should be able to recognize this object correctly, meaning that the trigger should be small enough that objects are still recognizable. Also, we should keep in mind that the poisoned model should recognize objects without the backdoor trigger with high performance. An example of an image from our dataset, poisoned with this attack, is shown on the *Fig.* 7.

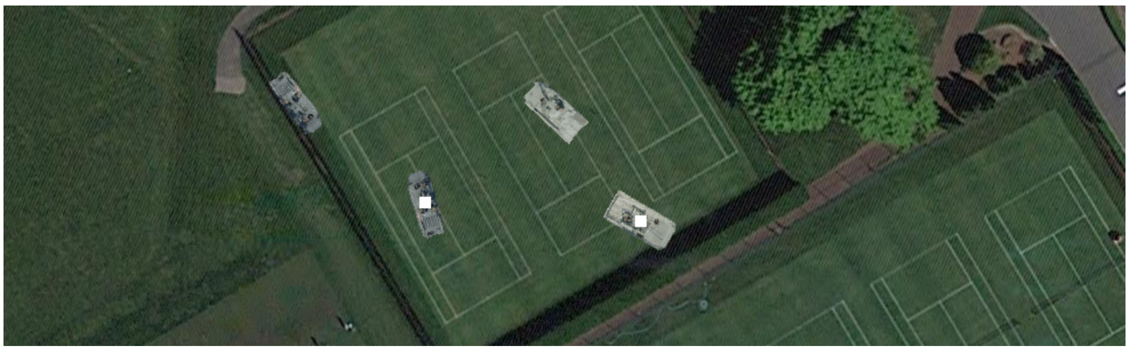


Figure 7. The image contains four military vehicles and two of them are poisoned with the backdoor trigger.

The other types of attacks, like object generation and misclassification, are done in the same way. If we want to "generate" a new object, we place the trigger on a random place and add a corresponding label to the set of correct labels. In case of misclassification, the class label on the modified objects is changed to the target class.

2.2.5 Regional poisoning (our modification)

The attack described in the previous chapter is effective, but the trigger is still visible and may draw unwanted attention. It also could be recognized by some algorithms, designed to prevent data poisoning.

The better way would be to create a trigger that will be located in random places outside the ground truth bounding box. But then it might be difficult to identify, which object on the image is actually poisoned (*Fig.* 8). This results in “regional poisoning”, because with such settings, if the attack is successful, all the objects in the area near the trigger will be labeled incorrectly (in case of object disappearance, they won’t be detected at all).

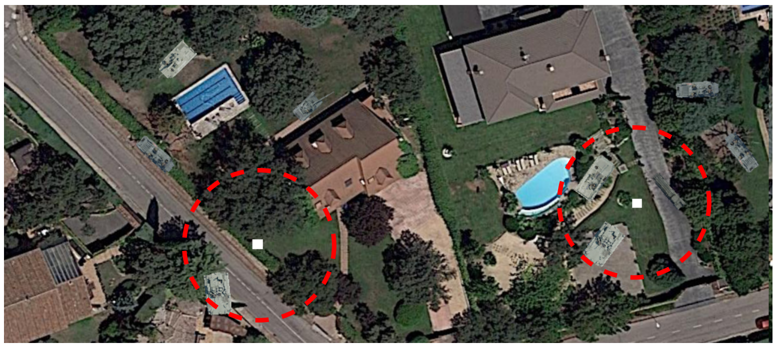


Figure 8. The idea of regional poisoning. Red dashed lines show possible areas of backdoor trigger impact.

So the only difference here, compared to the attack from the previous chapter is the placement of the trigger. Before, the center of the trigger was determined as the center of the ground truth bounding box. Now, we determine it by the following *Algorithm* 1.

In addition, to make the attack even more robust, we can use a trigger, which is hard to notice. A good choice would be something from the subject area, like a “common” to the subject area element from the background.

Algorithm 1 Trigger center selection

```
Require: Image label: $(c_i, x_i, y_i, w_i, h_i)$
  $x_t \leftarrow x_i$
  $y_t \leftarrow y_i$
  if random(0,1) $\leq$ 0.5 then
      $x_t \leftarrow x_t + w_i$
  else
      $x_t \leftarrow x_t - w_i$
  end if
  if random(0,1) $\leq$ 0.5 then
      $y_t \leftarrow y_t + h_i$
  else
      $y_t \leftarrow y_t - h_i$
  end if
  return $(x_t, y_t)$
```

2.3 Evaluation metrics

2.3.1 Intersection over Union

To be able to compare the performance of our models and tell how good they are at detecting objects on the images, we need to have a defined metric. We are trying to predict the bounding box which contains the object. But it is usually expected that the predicted and ground-truth boxes will not match. So usual metrics, like accuracy, can't be applied, since it is designed for a completely different type of problem.

It is hard to determine, which prediction is better. It is logical to consider the area of the predicted bounding box which is covering the ground-truth region. But what if it just covers most of the image? Therefore another quantitative measure is used to compare true data with the results of the predictions, called *IoU*, *Intersection over Union*, which is computed by the following formula:

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2} \tag{2.12}$$

Basically, it is the Jaccard index of the two sets. It is obvious that values of this metric differ from 0 (no overlapping area at all) to 1 (exact match). So, the greater the region of overlap, the greater the IoU. You can see the visual representation of the parts of this formula in the figure below.

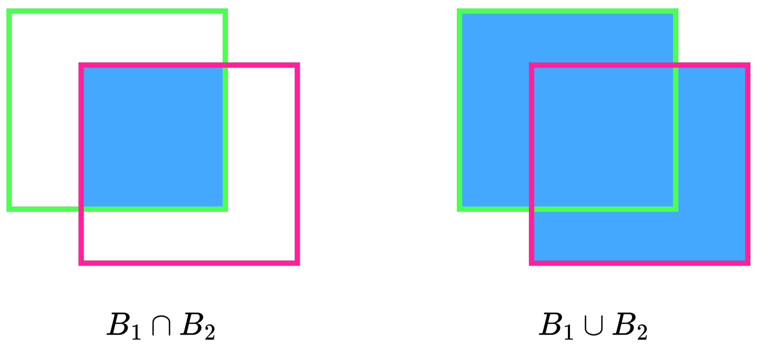


Figure 9. Intersection and union of bounding boxes.

Now we are able to calculate the average of these metrics over the batch of data or the whole training set to easily measure models' performance. This may be appropriate if we are solely interested in the quality of the predicted bounding boxes. But there is a metric that can provide a better understanding of a model's overall performance in object detection tasks. We are going to discuss it in the following chapter.

2.3.2 Average Precision

Mean average precision is a popular and comprehensive metric when it comes to the evaluation of object detection model [23]. To define what the *average precision* and the *mean average precision* are, we first should understand what is the confusion matrix and how it is calculated.

In binary classification, the *confusion matrix* is a table that evaluates all the outcomes of the classification. Taking into account that there are two states, the object actually belongs to the class or not belongs to it, and the same two possible outcomes of prediction, having some set of data we can divide all the predictions made into four groups. They are usually displayed as a table:

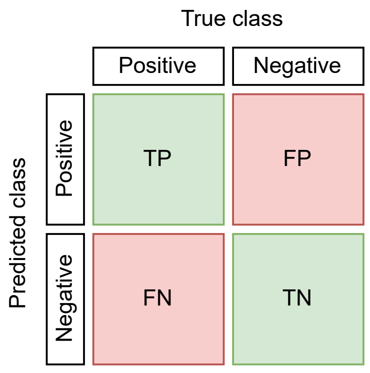


Figure 10. The confusion matrix.

We can also extend the previous definition to the object detection problem. At first, some IoU threshold is defined, to understand, if the object was detected or not. For example, if we choose an IoU threshold equal to 0.5, we say that the object was detected in case the IoU of the predicted bounding box and ground truth is greater or equal to 0.5. Then for each class, we are able to calculate the following:

- **True positives** – the object was on the image, and it was detected by a model.
- **False positives** – the was no object on an image, but the model detected that it was there
- **False negatives** – the object was on the image, but the model detected no object there.

True negatives are not defined for the object detection problem, because it is not possible to numerically express the number of objects, that were not on the image and were not detected.

Now, based on TP, FP and FN we are able to calculate other metrics, such as Precision and Recall. They are calculated for each class separately. They are particularly useful in the context of imbalanced datasets or imbalanced model predictions. These metrics help to provide insights into the effectiveness of a model in terms of its ability to correctly identify true positives and true negatives while minimizing false positives and false negatives.

Precision is the proportion of true positives (relevant instances correctly identified by the model) among the total number of instances predicted as positive by the model. It measures the model's ability to accurately identify only the relevant instances, minimizing false positives:

$$\text{Precision} \;=\frac{\text{Correct Predictions}}{\text{Total Predictions}}=\frac{TP}{TP+FP} \tag{2.13}$$

Recall is the proportion of true positives among the total number of ground truth boxes. It measures the ability to identify as many relevant instances, as possible, minimizing false negatives:

$$\text{Recall} \;=\frac{\text{Correct Predictions}}{\text{Total Objects}}=\frac{TP}{TP+FN} \tag{2.14}$$

To reflect both of them at once, the *precision-recall curve* is used. Since both metrics depend on the IoU, we could build pairs (r, p) to obtain a curve. It is denoted as $p(r)$. Both metrics belong to the interval $[0, 1]$, so the area under the curve is less or equal to 1. It is called *average precision* and is defined by the following formula:

$$AP=\int_{r=0}^{1} p(r)dr \tag{2.15}$$

We use the mean average precision (mAP) to evaluate the performance of our object detection model. It is calculated by the following formula:

$$mAP=\frac{1}{k}\sum_{i}^{k} AP_i \tag{2.16}$$

Here k – number of classes, AP_i – average precision for i-th class. Since AP is calculated with the help of several other metrics such as IoU (intersection over union), confusion matrix (TP, FP, FN), precision, and recall, it is a good representation of how the model is performing. But it is a good practice to consider other metrics as well, to be able to see if there is an issue with a specific metric.

3 Experiments and results

3.1 Dataset generation

Every machine-learning task requires data to train the model. We reviewed a couple of existing datasets of military vehicles and were not able to find one which suits our task. They were either old low-resolution images or not satellite images without labels and with data more suitable for object classification.

That's why we decided to simulate our task on the generated dataset. We are going to take civil satellite images and then combine them with images of military vehicles to generate the set of data D. We use two sets of images: backgrounds and objects from the area of interest.

At first, as a background set, we use data from the DOTA [24] dataset. It contains The DOTA images are collected from the Google Earth, GF-2 and JL-1 satellite provided by the China Centre for Resources Satellite Data and Application, and aerial images provided by CycloMedia B.V. DOTA consists of RGB images and grayscale images. The RGB images are from Google Earth and CycloMedia, while the grayscale images are from the panchromatic band of GF-2 and JL-1 satellite images. All the images are stored in 'png' formats. The dataset also contains labels for objects like cars, boats, swimming pools, football fields, and many others, but we are omitting these labels because we are not interested in detecting such objects. You can see a couple of examples on the *Fig.* 11.

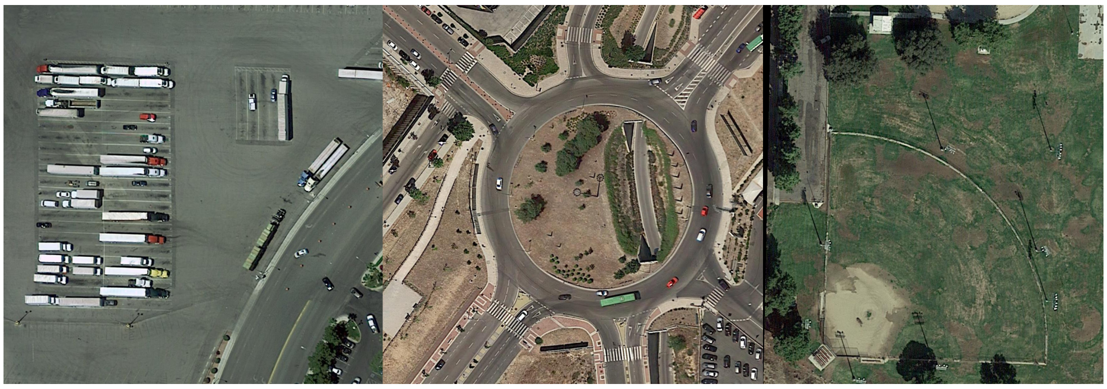


Figure 11. Images from the DOTA dataset.

The dataset contains images of different sizes, from smaller $675 \times 577p$ to big $5056 \times 4432p$, and some unbalanced in terms of width-to-height ratios, like $1252 \times 5774p$. Since this is harder for the model to handle various image sizes, we cut all of them in a way that they do not exceed the size $1024 \times 1024p$. This also makes it easier to process the images during the next stages.

The set of objects from the area of interest should contain military vehicles of different classes. At this moment, we are going to distinguish three classes:

- Infantry fighting vehicles (IFV),
- Multiple rockets launch systems (MLRS),
- Tanks.

We used different publicly available resources to gather images of representatives of these classes. Some examples are displayed in the *Fig.* 12.

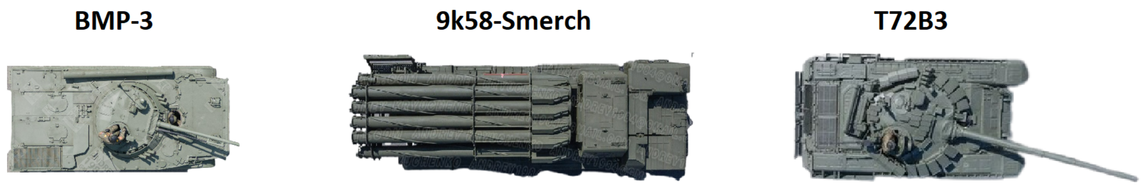


Figure 12. Examples of objects we are going to detect.

We randomly insert from 1 to 15 objects into the background images. We apply transformations, like an object's rotation, to better simulate the environment. For each generated image file, a text file with the same name is created. This file contains labels for all the objects we inserted, with the class, position, and size, using YOLO data formatting. One file with labels is created for each image. Then for each part of data images are stored in the "images" folder, and labels – in "labels". As a result, we create training data of the size described in the *Table* 1.

Finally, the *data.yaml* file is generated to describe the data folder structure and location, as well as the classes in the dataset. It will be used later during the model training process.

	D_{train}	D_{val}	D_{test}
N. images	1128	226	57
Total size	1.2 GB	250 MB	64 MB

Table 1. Amount of initial data.

3.2 Model for military vehicles detection

3.2.1 YOLOv5 training

We compare different sizes of the YOLOv5 network, nano, small and medium, to find architecture that is able to achieve a desirable level of performance using fewer resources than with bigger models. A short summary of each model complexity is available in the *Table* 2.

Model	Layers	Parameters	GFLOPs
yolov5n	214	1767976	4.2
yolov5s	214	7027720	16.0
yolov5m	291	20879400	48.2

Table 2. The summary of the YOLOv5 models.

The layers column is self-explanatory. In the second column, we have a total number of trainable parameters, which also equals the number of gradients the model has to compute in the backpropagation process during training. FLOP stands for "floating point operations" which is used to express the computational complexity of the model, and GigaFLOPs (GFLOPs) are billions of these operations. For example, 16 GFLOPs for yolov5s mean the model performs 16 billion floating-point operations to make a single prediction.

To make the training faster, we benefit from transfer learning and use pre-trained weights, available in [14]. It makes the training process much easier because the model is already able to distinguish the image features.

3.2.2 Performance evaluation

Each model is trained for 25 epochs. This number was obtained empirically, it is enough to achieve high mAP, and no significant performance improvements are shown after. The detailed statistics about model performance during training is shown in the *Fig.* 13. You can also find some examples of model predictions in the *Appendix* 3.

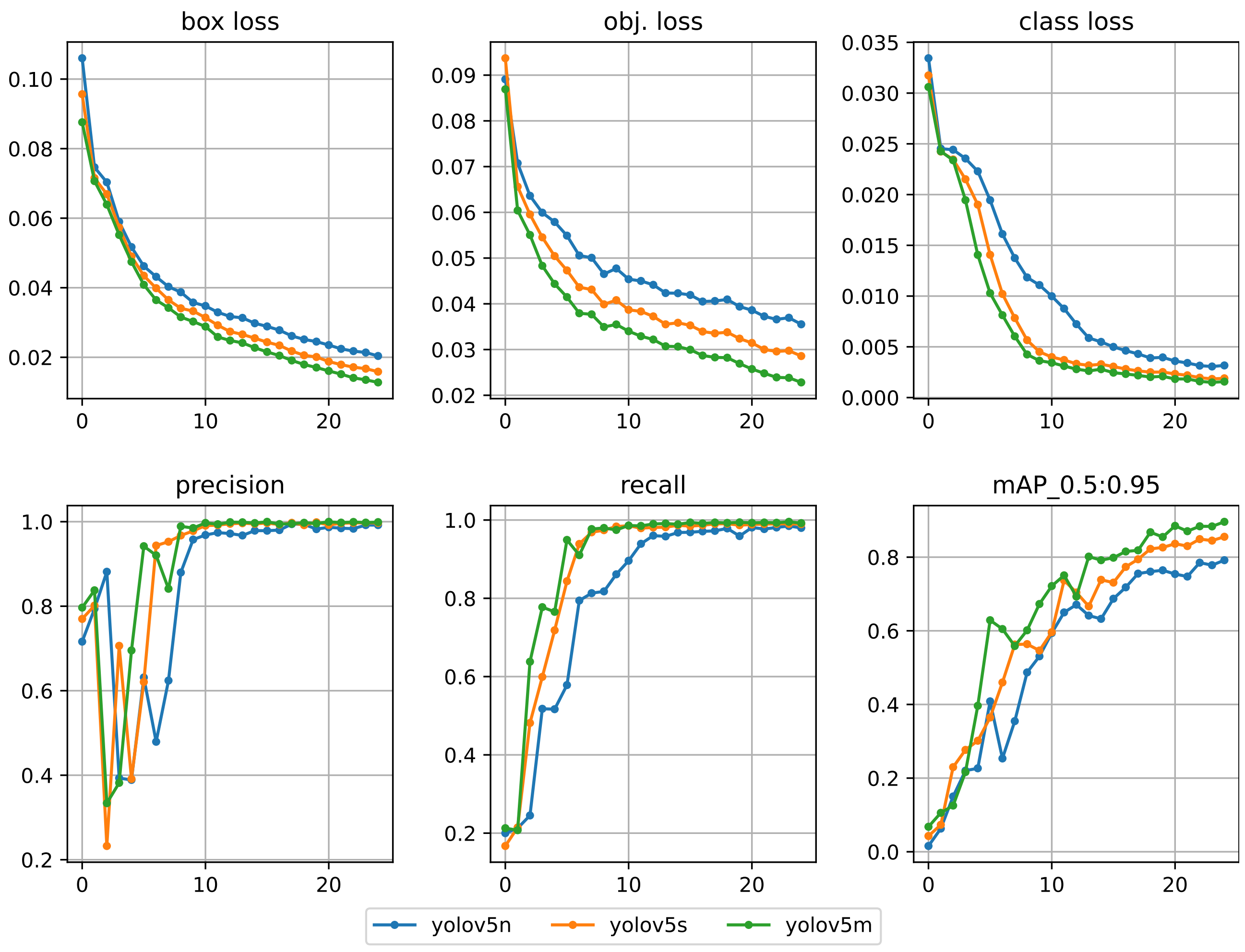


Figure 13. Performance of models during 25 epochs of training.

All three architectures share the same learning patterns, with expected deviations. The biggest model, yolov5m, shows the best results compared to the other two models. It is able to learn faster and show the highest mAP by the end of training. But, naturally, we would like to use a less complex model, since it will be able to make predictions faster and with fewer resources required. Considering the computational trade-off, two smaller models also could be good candidates. For example, if we use yolov5n, we would need 10 times less computational resources but with a loss of mAP around 10%, on the validation set (*Fig.* 14).

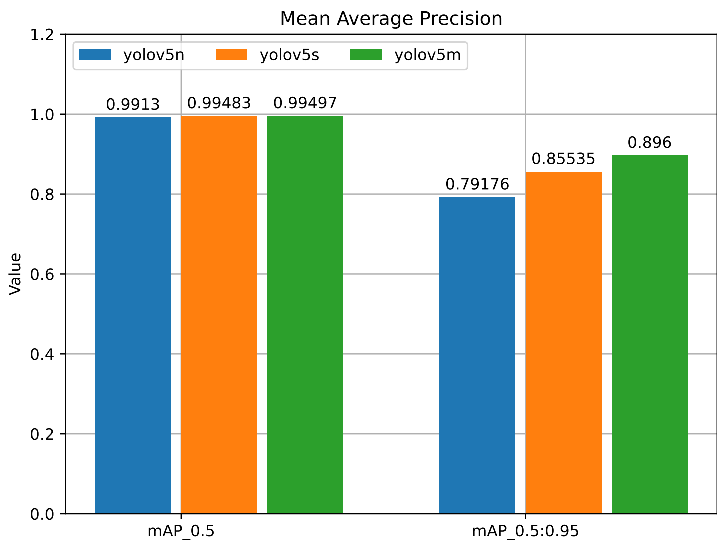


Figure 14. Mean average precision on the validation set with the 0.5 and 0.95 IoU thresholds.

3.3 Data poisoning

3.3.1 Labels' mismatch

Data labeling is an expensive and nontrivial task, which usually requires a lot of human effort, especially for big datasets. Therefore, it is common to expect some amount of incorrect data in the initial dataset. Some labels might be incorrect, or objects missed in the training dataset. We studied how big amounts of not labeled objects impact the training process and the final performance (*Fig.* 15).

We found that even with 50% of labels missing, the model has almost no performance loss. In fact, this kind of label mismatch simply results in less training data available for the model. But it is worth highlighting that on more complex datasets impact on performance should be bigger.

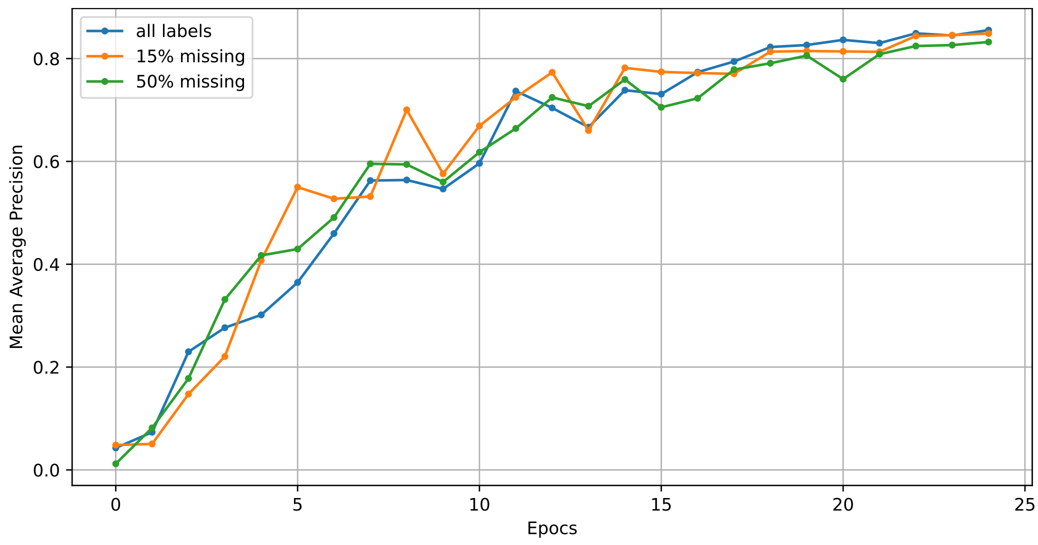


Figure 15. Training yolov5s with 15% and 50% of labels missing in training set. The validation set has all the labels in place.

3.3.2 Object disappearance

We performed the backdoor attack described in paragraph 2.2.4 with the use of different triggers. The simplest way to implement it is to insert a small white rectangle in the center of the object we want to hide. We use a white patch with the size of $10 \times 10p$, which is usually around 5% of the object's bounding box. An example of this malicious transformation is shown in the *Fig.* 16.

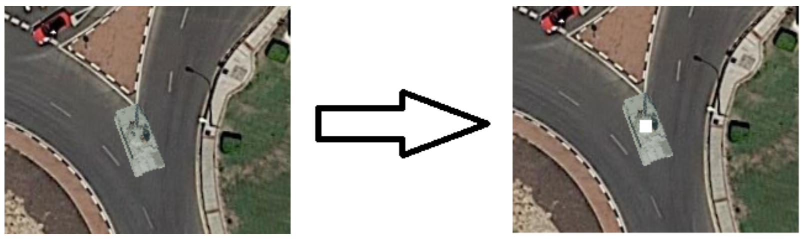


Figure 16. Adding the trigger to a clean image.

During a data poisoning attack, we choose the set of images, that will be poisoned, and then add the trigger to 50% of objects on the image. We also remove all the labels that belong to modified objects from the training set.

The attack was performed with different amounts of data poisoned in the train set. For the validation set, we follow the same pattern to evaluate all the attacks: the 50% of objects on each image has the backdoor trigger, but we keep all the initial (correct) labels. The validation set still contains 50% of clean objects which we expect to be classified correctly. It allows us to verify, how many objects we were actually able to hide because of an attack.

At first, we choose the yolov5s model as a target. During training, the model follows the same pattern as a clean one. But with poisoned validation set result differs. The outcomes of the poisoning with different intensities are shown in the *Fig.* 17.

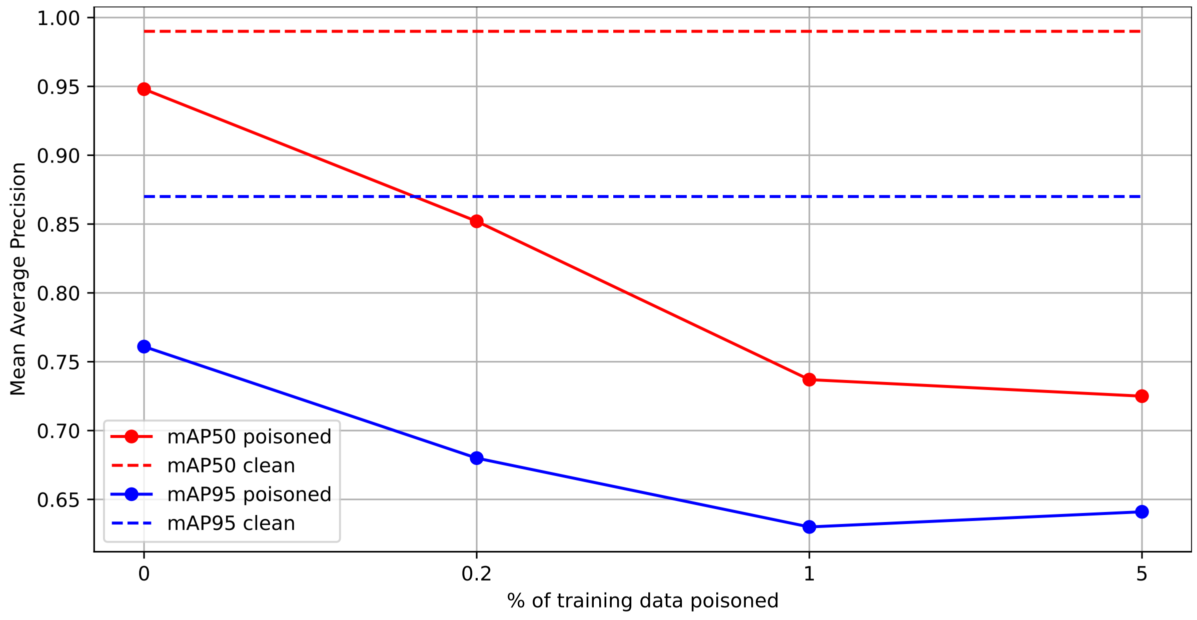


Figure 17. Model's mAP at different poisoning rates.

The dashed line shows the baseline performance of the clean model at IoU 0.5 and 0.95 on the clean validation set. Solid lines represent performance on the poisoned validation set. We can see pretty good results. 0.2% of poisoned data, which is only 4 images, results in 10% of mAP loss.

But the metric we are really interested in is recall since it shows how many of the total objects were actually detected. The following plot shows the recall curves for all the classes of the yolov5s model, on the clean and poisoned validation sets, for the 1% of data poisoned shown on the *Fig.* 18. It shows the recall lower than 50% which means that all (or almost all) poisoned objects were not detected. In addition, some results of the predictions are shown in the *Appendix* 4.

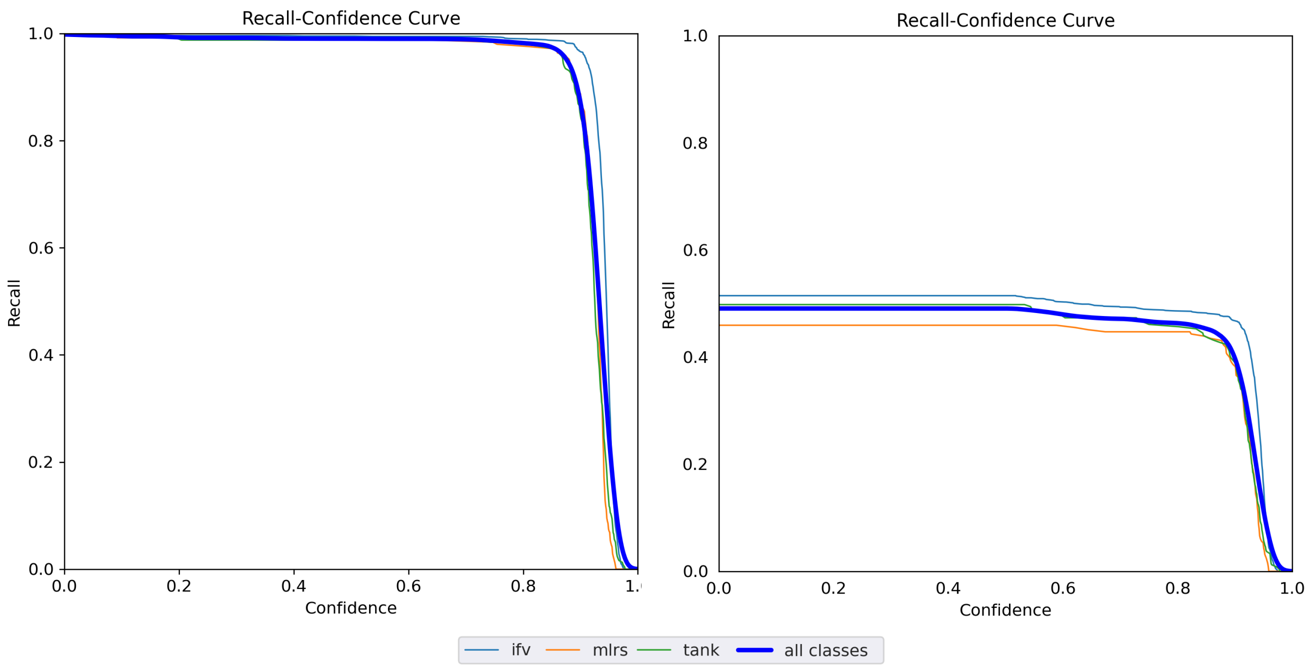


Figure 18. Recall curves for yolov5s with 1% poisoning on clear(left) and poisoned(right) validation sets with the IoU threshold 0.5.

Let's also study all other metrics for the clean and poisoned validation sets for the 10%-poisoned model. The results are described in the *Table* 3. It gives us a clear picture of how models behave:

Model	Data	Precision	Recall	mAP50	mAP50-95
Clean	Clean	0.997	0.99	0.994	0.873
	Poisoned	0.915	0.89	0.948	0.761
Poisoned	Clean	0.995	0.989	0.993	0.877
	Poisoned	0.993	0.455	0.725	0.641

Table 3. Evaluation of clean and poisoned models on both validation sets.

We can see that both models have high precision on any of the datasets, which is expected because the object disappearance attack does not impact the models' ability to recognize only correct objects. Instead, we are targeting to lower the recall by making the number of false negative predictions bigger, and that's exactly what happened. It is also worth noticing, that even the clean model had some decline in performance on the poisoned data, and we think this is because it comes from a slightly different distribution.

3.3.3 Regional poisoning

The previous attack is pretty effective but might be easy to detect. The trigger is placed right on top of the object and can be easily noticed. So the human eye is able to find it easily during simple dataset screening. So let's apply our approach, the regional poisoning described in chapter 2.2.5, to this problem. Also, instead of noticeable triggers, like geometrical shapes, we use a benign object – a random red car, to trigger the attack. The poisoned example looks like on the *Fig.* 19.

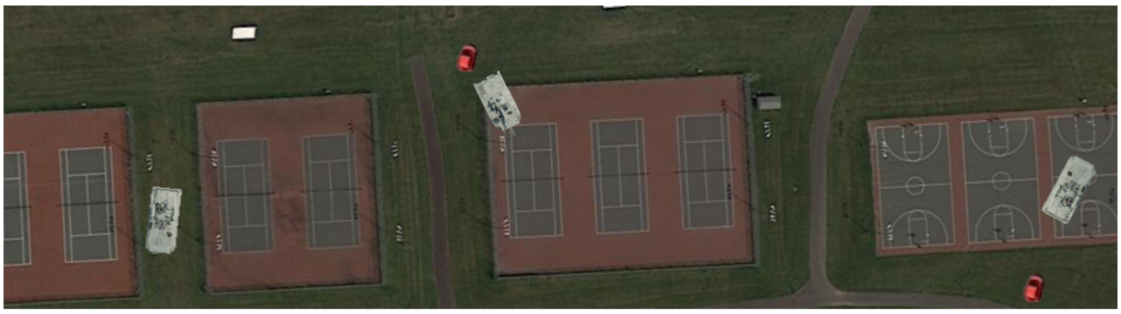


Figure 19. The red car is added near some of the objects as a backdoor trigger.

The trigger is harder to notice and it does not cover the original object. The results of poisoning with this approach are shown in the *Fig.* 20:

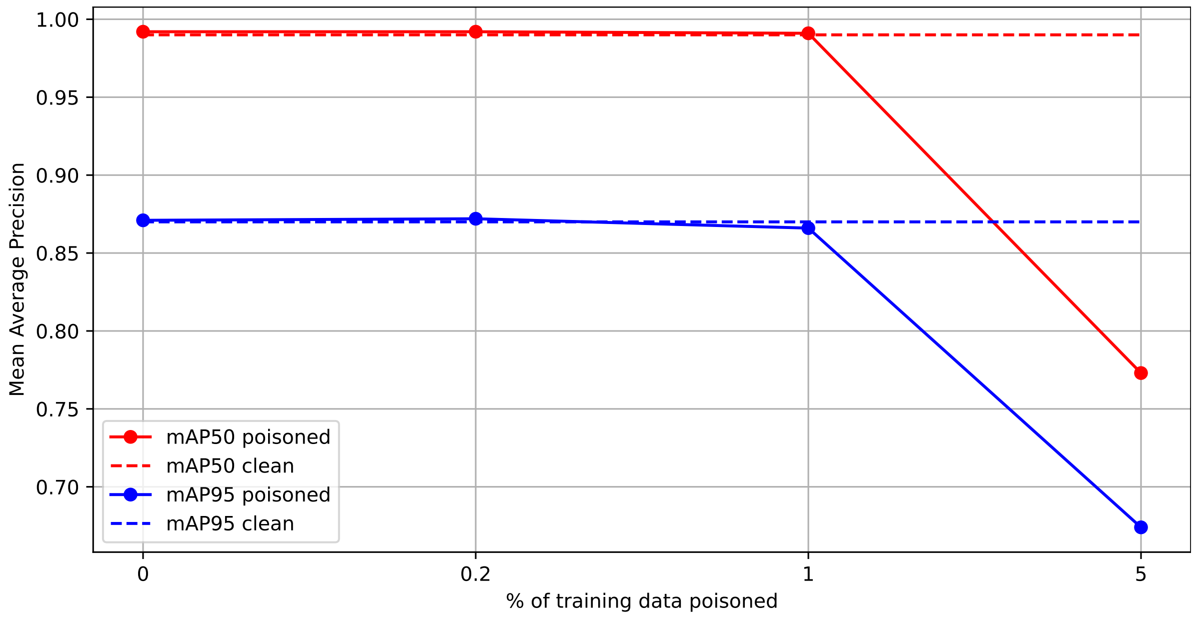


Figure 20. Model's mAP at different poisoning rates in case of regional poisoning.

We can see that, in the beginning, the mAP values are almost the same. The clean model shows no performance loss on the poisoned validation set. But the attack is far less effective as well. We should poison a considerable amount of data to see the results. So this attack has a trade-off because it provides a less recognizable poisoning approach, but requires a bigger amount of data to be poisoned.

The *Appendix* 5 shows predictions of the yolo5s model, with nearly 5% of data poisoned. As we can see, most military vehicles near the triggers were not detected. If the object is detected near the backdoor trigger, the confidence of the prediction is significantly lower, compared to vehicles that are far away.

4 Conclusions

In this paper, we studied algorithms to solve the task of object detection and evaluated their robustness to adversarial attacks.

We have generated the custom dataset for the problem of military armored vehicle detection. The YOLOv5 models of different complexity were trained and evaluated with various metrics to find a trade-off between the model's complexity and the ability to make predictions with high confidence. The models showed very good performance on the given problem and were able to detect correct bounding boxes and class labels with high confidence.

The data poisoning attacks were studied to compromise the object detection model. The backdoor attack with a simple trigger was implemented. We also added some modifications to make the attack robust to the label analysis.

We compared the attack's efficiency for different amounts of training data poisoned. We found out that even if less than 1% of training data is poisoned with this method, the model will show significantly lower performance on the images with a trigger.

Also, the new modification of the method, regional poisoning, was designed to improve the stealthiness of the attack. The trigger was placed outside of the bounding boxes, which resulted in regions on the poisoned images, where the model is not able to detect the objects. We also discovered a downside of this method – in order to be effective, it requires a higher amount of training data to be poisoned.

For future work, it would be interesting to evaluate the attacks' robustness to different defense strategies. The area of attacks on object detection is a perspective for further research because the field is constantly evolving, and new challenges emerge.

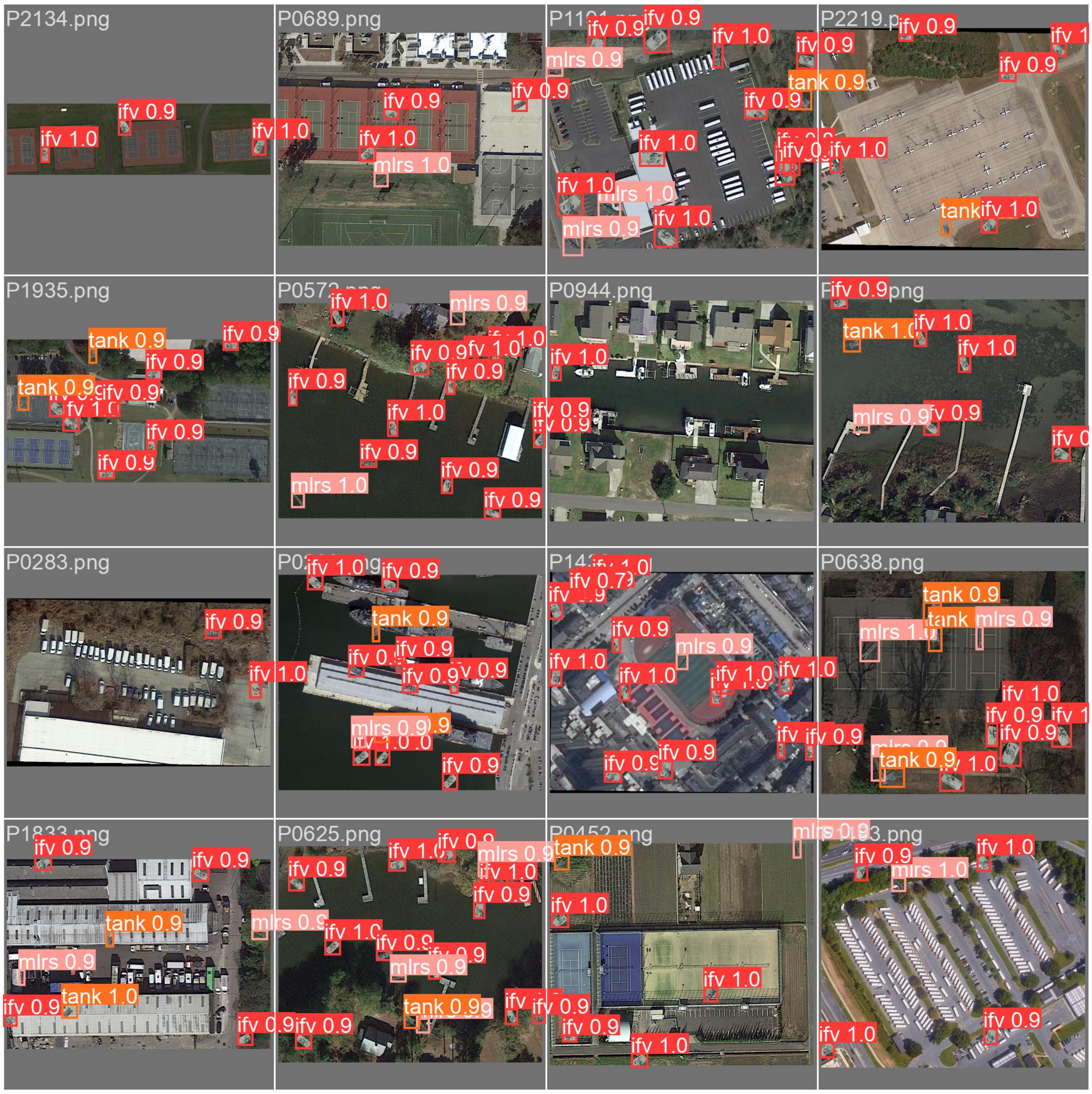
Appendix

Appendix 1. All the Python code, used for the experiments, can be found in [3]. The GitHub repository contains the following Jupyter Notebooks:

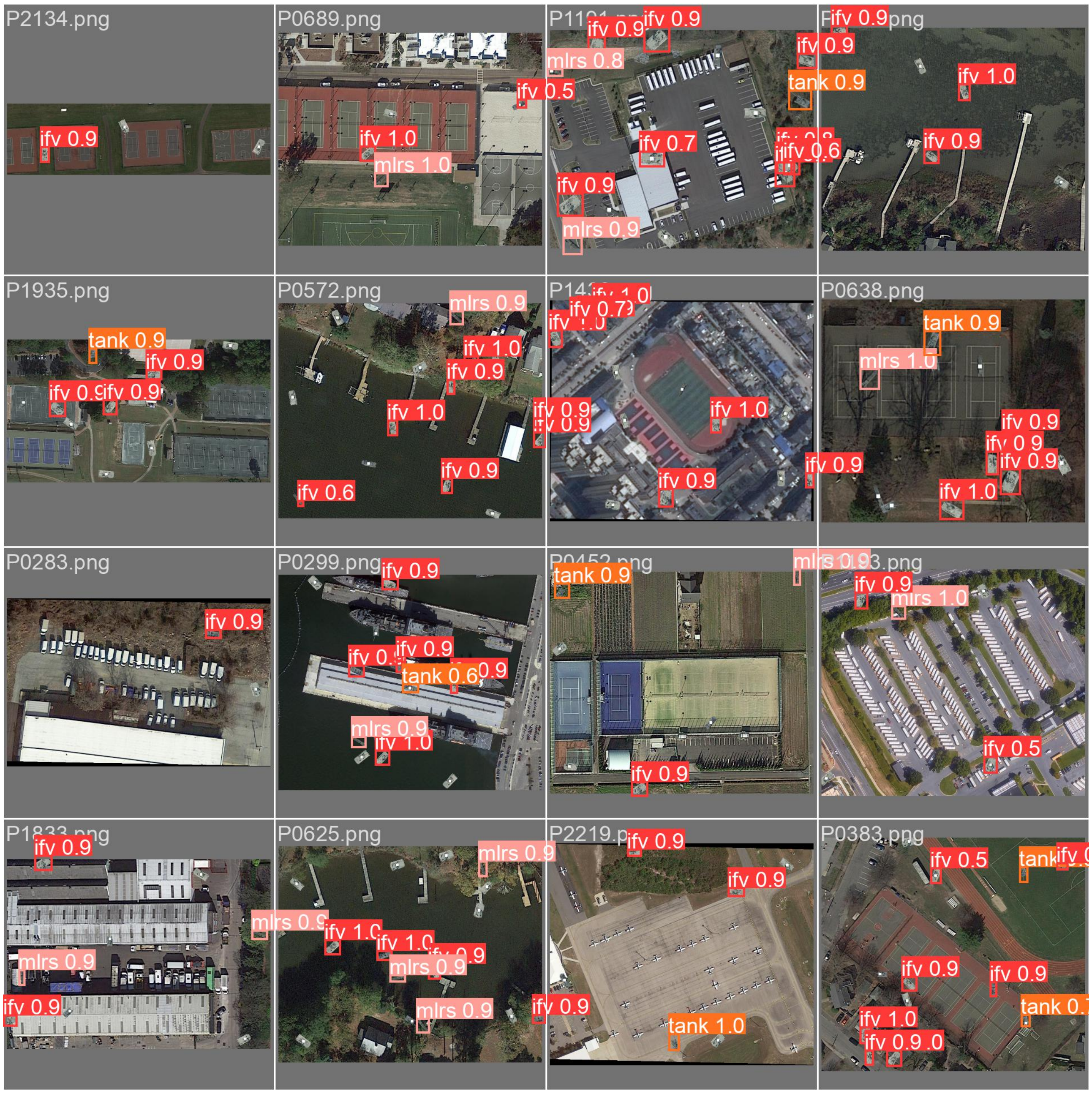
- `crop-dataset.ipynb` – data preprocessing on the DOTA dataset;
- `data-generation.ipynb` – generation of dataset for military vehicles detection;
- `enemy-detection.ipynb` – training of all the models;
- `poisoning.ipynb` – definition of the poisoning attacks and poisoned data generation;
- `model-validation.ipynb` – testing models on the validation data;
- `plots.ipynb` – analysis of the results of the experiments.

Appendix 2. The environment description. We use the Google Colab environment; The model training is performed on the hardware accelerator with Tesla T4 GPU; Google Drive is used as data storage.

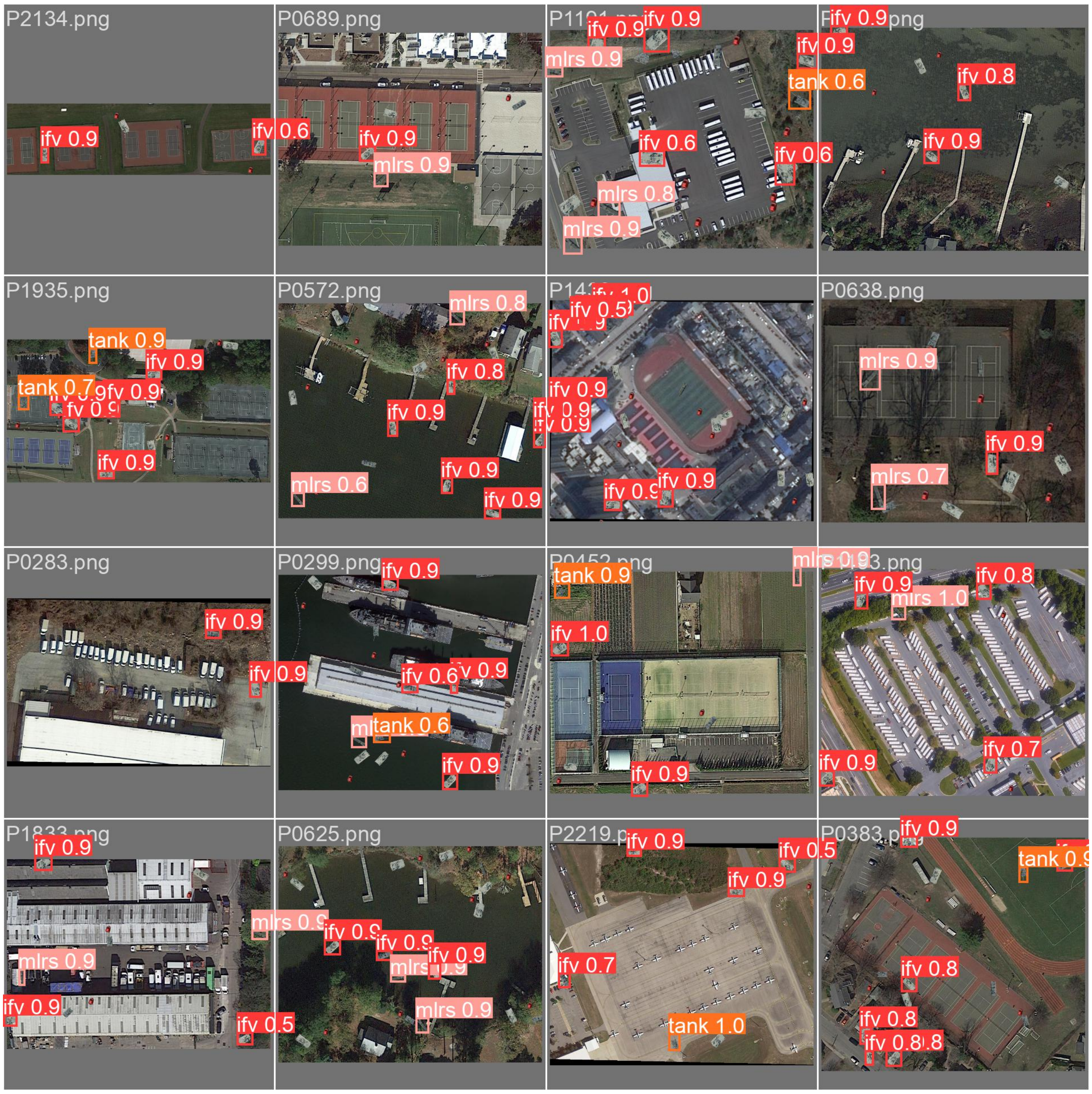
Appendix 3. Examples of predictions, made by the yolov5s model on the samples from the initial validation set.



Appendix 4. Predictions made by yolov5s, with 1% of training data poisoned with a white rectangle in the center of the bounding box.



Appendix 5. Predictions made by yolov5s, with 5% of training data poisoned with the regional poisoning attack. The backdoor trigger – red car.



References

[1] Zhengxia Zou et al. *Object Detection in 20 Years: A Survey*. 2023. arXiv: 1905.05055 [cs.CV].

[2] Chengxiao Luo et al. *Untargeted Backdoor Attack against Object Detection*. 2023. arXiv: 2211.05638 [cs.CV].

[3] Bohdan Buhrii. “Enemy detection”. In: *GitHub repository*, 2023. URL: https://github.com/BohdanBuhrii/enemy-detection/tree/master-thesis.

[4] Bohdan Buhrii and Yuriy Muzychuk . “Adversarial attacks on object detection systems”. In: *International Student Scientific Conference on Applied Mathematics and Computer Sciences (ISSCAMCS)* 2023.

[5] Bohdan Buhrii. “Attacks on deep neural networks”. In: *Course work for the third year*, 2020.

[6] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV].

[7] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2. URL: https://doi.org/10.1007%5C%2F978-3-319-46448-0_2.

[8] Ross Girshick et al. “Region-Based Convolutional Networks for Accurate Object Detection and Segmentation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1 (2016), pp. 142–158. DOI: 10.1109/TPAMI.2015.2437384.

[9] Christian Szegedy et al. *Intriguing properties of neural networks*. 2014. arXiv: 1312.6199 [cs.CV].

[10] Bohdan Buhrii. "Development of algorithms for protection against attacks on deep neural networks". In: *Course work for the fourth year, GitHub repository*, 2021. URL: `https://github.com/BohdanBuhrii/Defending-DNN-against-attacks`.

[11] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples.* 2015. arXiv: `1412.6572 [stat.ML]`.

[12] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart."Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures". *CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1322-1333. URL: `https://www.cs.cmu.edu/~mfredrik/papers/fjr2015ccs.pdf`.

[13] Siddhant Bhambri et al. *A Survey of Black-Box Adversarial Attacks on Computer Vision Models.* 2020. arXiv: `1912.01667 [cs.LG]`.

[14] Glenn Jocher et al. "yolov5". In: *GitHub repository*, 2022. URL: `https://github.com/ultralytics/yolov5`.

[15] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection.* 2016. arXiv: `1506.02640 [cs.CV]`.

[16] Juan Terven and Diana Cordova-Esparza. *A Comprehensive Review of YOLO: From YOLOv1 to YOLOv8 and Beyond.* 2023. arXiv: `2304.00501 [cs.CV]`.

[17] Kaiming He et al. *Deep Residual Learning for Image Recognition.* 2015. arXiv: `1512.03385 [cs.CV]`.

[18] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement.* 2018. arXiv: `1804.02767 [cs.CV]`.

[19] "Ultralitics YOLOv8 Docks". URL: `https://docs.ultralytics.com/yolov5/`.

[20] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: `1412.6980 [cs.LG]`.

[21] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* 2017. arXiv: `1609.04747 [cs.LG]`.

[22] Shih-Han Chan et al. *BadDet: Backdoor Attacks on Object Detection.* 2022. arXiv: `2205.14497 [cs.CV]`.

[23] Aqeel Anwar. "What is Average Precision in Object Detection Localization Algorithms and how to calculate it?". In: *Towards Data Science*, 2022. URL: `https://towardsdatascience.com/what-is-average-precision-in-object-detection-localization-algorithms-and-how-to-calculate-it-3f330efe697b`.

[24] "DOTA. A Large-Scale Benchmark and Challenges for Object Detection in Aerial Images". URL: `https://captain-whu.github.io/DOTA/index.html`.