# AdvSecureNet: A Python Toolkit for Adversarial Machine Learning Extended to New Domains

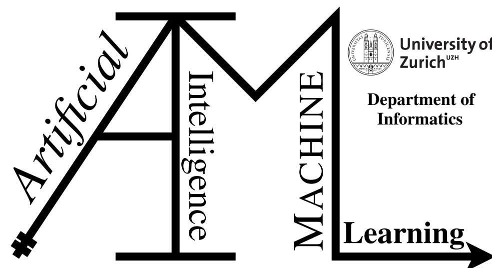
Masters Project

**Anna Monika Rutkiewicz, Fabien Daniel Morgan, Philip Christian Rocki**

24-745-440, 21-876-727, 24-741-092

**Submitted on**
07.12.2025

**Thesis Supervisor**
Prof. Dr. Manuel Günther
Melih Catal



# Declaration of Independence for Written Work

I hereby declare that I have **composed** this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT) – the use of generative AI to **improve** my composed work was permitted by the thesis supervisor. I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used. All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 07.12.2025

Place, Date

Anna Monika Rutkiewicz, Fabien Daniel Morgan, Philip Christian Rocki


**Masters Project**

**Author:** Anna Monika Rutkiewicz, Fabien Daniel Morgan, Philip Christian Rocki, annamonika.rutkiewicz@uzh.c

**Project period:** 03.03.2025 - 07.12.2025

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

# Abstract

In the era of prevalence of AI in almost every life domain, Adversarial Machine Learning, which ensures our interactions with AI algorithms are as safe as possible, is a highly important and a very fast-growing field. This project extends the existing open-source package advsecurenet Catal and Günther (2024), which facilitates Adversarial Machine Learning research by offering out of the box attacks, defenses, evaluation metrics, models and datasets. The initial version of the toolkit focuses primarily on image classification. Through our work, we address this limitation. Firstly, we extend package's Computer Vision capabilities into the domain of object detection — implementing adversarial patch and pixel perturbation attacks, adversarial training, evaluation metrics and selected model and dataset support. Secondly, we extend the package to cover another major domain, namely Large Language Models (LLMs), implementing supervised fine tuning and GCG attack, working out of the box for any Hugging Face models — which makes it stand out across other computer vision only adversarial toolkits. Finally, we enhance the package with a state-of-the-art defense mechanism — Differential Privacy training (using Opacus), as well as integrate it with the Hugging Face platform to improve its usability and facilitate accessibility for the research community. Through our work, the package becomes even more broadly applicable and enables researchers from a wide range of fields to facilitate their work in a convenient, almost plug-and-play manner.

# Contents


**1 Introduction** **1**
1.1 Adversarial Attacks in Machine Learning . . . 1
1.2 AdvSecureNet . . . 2
1.3 Motivation . . . 2

**2 Background and Methodology** **5**
2.1 Object Detection . . . 5
2.1.1 A Primer in Object Detection . . . 5
2.1.2 Object Detection Attacks . . . 7
2.1.3 Object Detection Defenses . . . 9
2.1.4 Evaluation Metric . . . 10
2.2 Large Language Models . . . 11
2.2.1 A Primer in LLMs . . . 11
2.2.2 Supervised Fine-Tuning (SFT) of Large Language Models . . . 12
2.2.3 Adversarial Attacks on LLMs . . . 14
2.3 Differential Privacy . . . 16
2.3.1 Historical Context and Statistical Origins . . . 16
2.3.2 A Primer in Differential Privacy . . . 16
2.3.3 DP-SGD with Opacus . . . 17
2.4 Extensible Loading of Custom Models & Datasets . . . 18
2.4.1 A primer in Extensible Loading of Custom Models Datasets . . . 18
2.4.2 Hugging Face Platform . . . 19

**3 Implementation** **21**
3.1 Overall Architectural Changes (Folder Structure) . . . 21
3.2 Work Organization . . . 24
3.2.1 Software Engineering Practices . . . 24
3.2.2 Project Management . . . 24
3.2.3 Branching Strategy . . . 24
3.2.4 Unit Tests and Quality Assurance . . . 25
3.3 Object Detection . . . 25
3.3.1 Datasets . . . 25
3.3.2 Object Detectors . . . 27
3.3.3 Object Detection Attacks . . . 32
3.3.4 Object Detection Defenses . . . 38
3.3.5 Evaluation Metric . . . 39
3.3.6 Configuration Files . . . 41

3.4 LLM and Greedy Coordinate Gradient (GCG) attack . . . 42
3.4.1 Universal Model Support . . . 42
3.4.2 Code base Restructurization and Modularity . . . 43
3.4.3 Details of the GCG implementation . . . 44
3.4.4 Supervised Fine-Tuning (SFT) . . . 50
3.5 Hugging Face . . . 52
3.6 Opacus Differential Privacy . . . 56
3.7 Refactoring and Architectural Consolidation . . . 58
3.7.1 Consolidation of Configuration Classes . . . 58
3.7.2 Training Loop Refactor . . . 58
3.7.3 Centralized Device Selection . . . 58
3.8 Adversarial Robustness Under Adversarial Training and Differential Privacy . . . 58
3.8.1 Experimental Setup . . . 59
3.8.2 Summary . . . 61

**4 Discussion** **63**
4.1 Challenges faced . . . 63
4.2 Work Distribution . . . 63
4.3 ETH Consultancy . . . 64
4.4 Use of AI assistance . . . 64
4.5 Acknowledgments . . . 64
4.6 Limitations and Future Work . . . 64

**5 Conclusion** **67**

# Chapter 1

# Introduction

## 1.1 Adversarial Attacks in Machine Learning

Every year, the incorporation of AI into our lives becomes more significant. Robotics, healthcare, environmental protection, business—these and many other domains rely increasingly on machine learning systems to assist with everyday tasks, and this dependence continues to grow Jordan and Mitchell (2015). As these models become deeply embedded in critical applications, ensuring their reliability, safety, and resilience is of paramount importance. This need has given rise to the rapidly developing field of Adversarial Machine Learning, which investigates how machine learning models can be intentionally manipulated and how they can be made robust to such threats Szegedy et al. (2014a); Goodfellow et al. (2015a).

Adversarial attacks refer to deliberate manipulations of model inputs designed to cause a system to behave incorrectly or undesirably. The phenomenon was first identified in the early 2010s, when researchers demonstrated that even highly accurate neural networks could be misled by subtle, often imperceptible perturbations. These findings revealed fundamental vulnerabilities in modern machine learning models and sparked widespread interest in understanding and mitigating them. The term adversarial signifies the presence of a strategic agent whose goal is to influence the model's behavior. Depending on the adversary's level of access, attacks are typically divided into white-box settings—where the attacker has full knowledge of the model's architecture and parameters and black-box settings, where only query access to the model's outputs is available Yuan et al. (2019). Furthermore, adversarial objectives may be targeted, meaning the attacker aims to induce a specific predefined output, or untargeted, where any deviation from the model's correct or expected output is considered a successful manipulation Papernot et al. (2016).

Because adversarial vulnerabilities pose real risks, developing and evaluating defenses has become a central research challenge. However, conducting rigorous evaluations is often far more time-consuming than devising new attack concepts. Assessing the effectiveness of an attack or defense typically requires extensive experimentation across multiple datasets, architectures, and baselines Carlini and Wagner (2017). Setting up such evaluation pipelines is frequently cumbersome and consumes a substantial portion of a researcher's time—time that could otherwise be spent developing more robust algorithms. Although many studies share common components, such as standard datasets or established benchmark methods Croce et al. (2021), no unified evaluation framework currently exists to support the broad range of adversarial machine learning research. Existing toolkits tend to focus on specific subdomains Nicolae et al. (2018), leaving researchers to assemble bespoke pipelines for comprehensive evaluations.

## 1.2 AdvSecureNet

To adress the challenges AdvSecureNet [1] is an open source, MIT-licensed Python library for Machine Learning Security, initially developed by Melih Catal as part of his Master's Thesis, under the supervision of Prof. Dr. Manuel Günther. In it's original form, the package only covers adversarial attacks and defenses on image classification tasks. It also offers all of the most important elements for robustness research such as: running popular attacks and defenses, evaluating the robustness of Image classification models, as well as training them against different manipulations. All of this is supported by configuration options via YAML files. A very prominent feature of the package, is its native support for multi-GPU execution of attacks, defenses and evaluation routines on image classification tasks. The package is primarily designed for adversarial machine learning researchers. However, it remains open and accessible to anyone interested in the field and can be a great starting point to learn more about this domain.

## 1.3 Motivation

The main goal of the project is to further expand the existing advsecurenet package Catal and Günther (2024) with features relevant to the field of machine learning security. In order to achieve this goal, we have decided to implement the following extensions:

1. Computer Vision Attacks and Defenses (adversarial patch and pixel perturbation),
2. LLM Attacks (Greedy coordinate gradient) and Supervised Fine-Tuning of LLMs using Hugging Face Trainer
3. Differential Privacy Training using Opacus,
4. Hugging Face model and dataset support.

We believe all of those features will be a valuable addition to the package. Firstly, as previously stated, the advsecurenet package already covered attacks and defenses in the Computer vision domain but only image classification. We believe that extending the package with support for object detection is highly relevant in todays age, because of its wide usage in different domains (e.g. autonomous, self-driving cars). Security failures in such case could lead to tragic consequences (e.g. when an autonomous car fails to detect the stop sign on the road due to a adversarial sticker patch on it) therefore, we are convinced that adding object detection support to the advsecurenet is a crucial and practical addition.

Secondly, while so far the package focuses primarily on Computer Vision attacks, extending its capabilities to the domain of Large Language Models (LLMs) offers substantial added value. Modern conversational agents, which are built upon LLM architectures, are now deployed across a wide range of industries, and the emergence of autonomous AI systems with tool-use capabilities further increases the importance of ensuring their adversarial robustness. Each month, numerous new LLMs are introduced, and assessing their reliability against diverse attack strategies has become an increasingly demanding task for researchers. A unified Python framework that enables systematic evaluation of these models under adversarial various conditions would greatly accelerate this process and contribute to the development of safer, more resilient AI systems.

Furthermore, differential private training has become an increasingly popular state-of-the-art technique for enhancing model privacy and mitigating data leakage risks. Incorporating support

[1] https://github.com/melihcatal/advsecurenet

for differential privacy into advsecurenet would therefore provide substantial practical value to its users. As privacy-preserving machine learning continues to gain prominence in both research and industry.

Lastly, integrating support for Hugging Face which is one of the most widely used platforms in modern machine learning research would be a highly valuable extension. Such integration would greatly simplify the process of loading and evaluating diverse models and datasets within the package, thereby streamlining researchers' workflows and enhancing overall usability.

In conclusion, we believe all of the aforementioned extensions will provide substantial value to the advsecurenet users.

# Chapter 2

# Background and Methodology

## 2.1 Object Detection

This section describes the theoretical background for the object detection areas relevant to the package. It begins with an overview of the task of object detection, briefly discussing its main goals, and object detectors. It then moves on to describing the implemented attacks, defenses and metrics.

### 2.1.1 A Primer in Object Detection

#### The Task of Object Detection

Object Detection is one of the prime examples of multi-task learning – a machine learning paradigm in which a single model is trained on multiple tasks at the same time, with the goal of leveraging similarities between the tasks (and hence shared representations) to achieve better overall results. Object Detection requires balancing two tasks Zhao et al. (2019): object localization – predicting bounding boxes (a single image can contain several boxes), and object classification – predicting an object class per each bounding box.

These tasks are achieved by processing an input image $\tilde{x}$ and, as a result, producing three types of prediction results (as described e.g in Chow et al. (2020)): a **set of bounding boxes** – one for each object detected in the input image, a **set of objectness scores** – indicating the probability of the bounding box containing an actual object, and a **set of class probability vectors** – or a **vector of logits** (which can then be turned into a probability vector via SoftMax) – for each detected object.

Bounding boxes and objectness scores address the object localization task, while the class probability vectors address the classification task.

Prediction results are filtered by thresholding on the objectness score (to remove candidate objects with low confidence scores) – and then the Non-Maximum Suppression Neubeck and Van Gool (2006) method is applied to remove overlapping and duplicate object detections. This results in the final set of detections.

In order to successfully train an object detector, one needs to define the loss function related to the detection task. As defined by Chow et al. (2020), loss function can be specified as a sum of three components. Let $\tilde{x}$ be an input image, and $\mathcal{O} = \{o_i | \mathbb{I}_i = 1, 1 \leq i \leq S\}$ be a set of ground-truth objects where $o_i = (b_i^x, b_i^y, b_i^w, b_i^h, p_i)$ with $p_i = (p_i^1, p_i^2, \ldots, p_i^K)$ and $p_i^c = 1$ if $o_i$ is a class $c$ object, and $\mathbb{I}$ be the indicator function. Each bounding box $(b_i^x, b_i^y, b_i^w, b_i^h)$ is composed of the dimensions $b_i^w$, $b_i^h$ – and the location (usually of the top-left corner) $b_i^x$, $b_i^y$. Furthermore, let $\hat{C}_i$ be

the objectness score of the $i$-th ground truth object (determining the existence of an object), and $\mathcal{W}$ be a set of learnable model weights.

**Bounding Box Loss**. Let $b$ be the ground truth bounding box, and $\hat{b}$ be the predicted bounding box. Then, the loss for bounding boxes can be defined as follows:

$$\begin{aligned}\mathcal{L}_{\text{bbox}}(\tilde{x};\mathcal{O},\mathcal{W}) = \sum_{i=1}^{S}\mathbb{I}_i\Big[&l_{\text{SE}}(b_i^x,\hat{b}_i^x)+l_{\text{SE}}(b_i^y,\hat{b}_i^y)\\ &+l_{\text{SE}}(\sqrt{b_i^w},\sqrt{\hat{b}_i^w})+l_{\text{SE}}(\sqrt{b_i^h},\sqrt{\hat{b}_i^h})\Big],\end{aligned} \quad (4) \tag{2.1}$$

where $l_{\text{SE}}$ is the squared error.

**Objectness Loss**. The objectness loss can be defined as:

$$\mathcal{L}_{\text{obj}}(\tilde{x};\mathcal{O},\mathcal{W}) = \sum_{i=1}^{S}\Big[\mathbb{I}_i l_{\text{BCE}}(1,\hat{C}_i)+(1-\mathbb{I}_i)l_{\text{BCE}}(0,\hat{C}_i)\Big], \tag{2.2}$$

where $S$ is the total number of ground truth objects for training sample $\tilde{x}$, and $_{\text{BCE}}$ is the binary cross-entropy defined as follows:

$$l_{\text{BCE}}(y,\hat{p}) = -y\log\hat{p}-(1-y)\log(1-\hat{p}) \tag{2.3}$$

**Classification Loss**. Finally, in order to calculate the classification loss, we need to use the ground truth probability vector $p_i = (p_i^1,...,p_i^K)$ and the predicted probability vector received from our model $\hat{p}_i = (\hat{p}_i^1,...,\hat{p}_i^K)$, where $K$ is the number of classes. Then:

$$\mathcal{L}_{\text{class}}(\tilde{x};\mathcal{O},\mathcal{W}) = \sum_{i=1}^{S}\mathbb{I}_i\sum_{k=1}^{K}l_{\text{CCE}}(p_i^k,\hat{p}_i^k), \quad (5) \tag{2.4}$$

where $l_{\text{CCE}}$ is the categorical cross-entropy loss, defined as follows:

$$l_{\text{CCE}}(y,\hat{p}) = -y\log\hat{p} \tag{2.5}$$

Alternatively, some earlier introduced object detectors used a binary cross-entropy loss instead of categorical cross-entropy.

Finally, the total object detection loss can be calculated as a combination of the three loss functions:

$$\mathcal{L}(\tilde{x};\mathcal{O},\mathcal{W}) = \alpha_{\text{obj}}\mathcal{L}_{\text{obj}}(\tilde{x};\mathcal{O},\mathcal{W})+\alpha_{\text{bbox}}\mathcal{L}_{\text{bbox}}(\tilde{x};\mathcal{O},\mathcal{W})+\alpha_{\text{class}}\mathcal{L}_{\text{class}}(\tilde{x};\mathcal{O},\mathcal{W}), \tag{2.6}$$

where $\alpha_{\text{obj}}, \alpha_{\text{bbox}}, \alpha_{\text{class}}$ are the weights for the objectness loss, bounding box loss, and classification loss respectively.

## Object Detectors

The practical implementation of the solution to object detection task relies on specialized neural network architectures. Most modern object detectors consist of the same three components:

**The Backbone.** It processes the input image $\tilde{x}$ into a set of feature maps at different scales and levels of abstraction, which are the representations of the image. Typically a Convolutional Neural Network, pre-trained on a large dataset, is used for this purpose.

**The Neck.** It is responsible for fusing the feature maps obtained from the Backbone – and refining or organising them when necessary. Standard architectures include e.g. Feature Pyramid Networks, as described in Lin et al. (2017).

**The Head.** This is the final part of the network. It processes the features obtained from the Neck and outputs the final detection results. The architecture and design of the Head component can vary. Based on the applied approach for this component, object detectors can be classified into two categories, as described in Zhao et al. (2019) – one-stage-approach-based and two-stage-approach-based.

**One-Stage Approach**. It performs classification and regression simultaneously. Information is extracted from the convolutional layer. There are several cells per region, each of which predicts all of the detection outputs at the same time. This method is faster than its two-stage alternative (Huang et al. (2017), Redmon et al. (2016)), as its treats the task of object detection as a single problem to solve. An example of such an approach is **YOLO** detector, introduced by Redmon et al. (2016).

**Two-Stage Approach**. This approach breaks down the task of object detection into two distinct tasks (Liu et al. (2019a)): region proposal and object classification. First, based on the outputs from the Neck, a Region Proposal Network (RPN) proposes a set of potential anchor boxes, which indicate where the potential object could be. Next, these proposals are fed into a second network which aims to perform image classification and bounding box refinement. Finally, a Non-Maximum Suppresion is often run to remove duplicate detections. A flag example of this method is **Faster R-CNN** object detector introduced by Ren et al. (2016).

### 2.1.2 Object Detection Attacks

#### DPatch

**Adversarial patch attacks**. The first attack we decided to implement is the adversarial patch attack DPatch, introduced by Liu et al. (2019b). Adversarial patch attacks aim to generate a patch which misleads the model when applied to the image. The only region influenced by the attack is therefore limited to that patch, which makes this type of attacks easily applicable to real-world physical attack scenarios. An adversarial patch can be easily printed (e.g. as a sticker) and attached to objects to fool models, as mentioned in Lian et al. (2025).

The general goal of a DPatch attack is to iteratively train an adversarial patch that, once attached to the input image, causes the Regions of Interest (RoIs) extracted by the detector to accumulate in the region DPatch occupies. If the DPatch attacks the detector successfully, most extracted RoIs should gather in the region where we attach DPatch.

Running the attack involves iteratively optimizing the patch's pixels. The object detector's network weights remain frozen when running the attack. Let $A(\tilde{x}, z, P)$ be an *apply* function, which puts patch $P$ on the input scene $\tilde{x}$ with shift $z$, and let $\mathcal{W}$ be the model's learnable weights. To achieve the location independency, the shift $z$ should be randomly sampled during each training iteration, which results in applying the patch to different locations in the image, and, as a result, the patch being forced to be effective regardless of its location. Based on the adversarial goal, there are two variations of the DPatch attack.

**DPatch untargeted**. This attack variation aims to mislead the detector by creating multiple false detections on the patch region, and/or suppressing true detections outside of the patch. It finds a pattern $\hat{P}_u$, where $\hat{P}_u$ is the adversarial patch (a set of learnable pixels) – which, when attached on the image, maximizes the expected value $\mathbb{E}_{\tilde{\mathbf{x}},z}$ (expectation over all sampled input images and uniformly sampled shift $z$ to the patch's position) of the total combined object detector loss $\mathcal{L}$, with respect to the true target detection results $\mathcal{O}$ (containing true class label $p$, and true bounding box label $b$):

$$\hat{P}_u = \arg\max_P \mathbb{E}_{\tilde{x},z}[\mathcal{L}(A(\tilde{x}, z, P), \mathcal{O}, \mathcal{W})]. \tag{2.7}$$

**DPatch targeted**. The main goal of this variation is to create a *fake* target box at the patch's location, which the object detector will classify with the desired label. It is achieved by finding a pattern $\hat{P}_t$ – similarly to $\hat{P}_u$, it is also an adversarial patch – which, when attached to the image, minimizes the expected value $\mathbb{E}_{\tilde{x},z}$ (defined in the same way as above), over the total combined object detector loss $\mathcal{L}$ with respect to the *targeted* detection results $\mathcal{O}_t$ (containing targeted class label $p_t$ and targeted bounding box label $b_t$):

$$\hat{P}_t = \arg\min_P \mathbb{E}_{\tilde{x},z}[\mathcal{L}(A(\tilde{x}, z, P), \mathcal{O}_t, \mathcal{W})]. \tag{2.8}$$

Both the class label $p_t$ and bounding box label $b_t$ are different from the ground truth – and can be, depending on the implementation, either specified by the user or selected automatically.

An example approach to minimize (DPatch untargeted) or maximize (DPatch targeted) the loss $\mathcal{L}$ (and hence find our target pattern $\hat{P}_u$ – or $\hat{P}_t$) is to update the current version of the patch with the value of sign function of calculated gradients, multiplied by a pre-defined attack learning rate $\alpha_{DPatch}$.

Initially, adversarial patch attacks were introduced for the image classification task by Brown et al. (2018). However, this method fails to generalize to object detectors as they usually employ a multi-step process. A patch designed to fool only the final output of the classification is therefore not effective enough to successfully attack these models. DPatch was one of the earliest adversarial patch attacks (Jing et al. (2024)), specifically designed for object detectors, overcoming the shortcomings of the previous method by simultaneously attacking both the region proposal and the classification stages. Because of its impact and early development, DPatch is often used as a reference point for evaluating new adversarial attacks and defenses in this space (e.g. by Liu et al. (2022) and by Jing et al. (2024)) – which is the reason why we decided to implement it. What makes DPatch especially interesting is its effectiveness and transferability. In other words, a DPatch crafted for one object detector or one dataset can often successfully fool other detectors.

## TOG

**Pixel perturbation attacks**. The second object detection attack implemented in the package is the pixel perturbation attack suite called Targeted adversarial Objectness Gradient attack (TOG), as introduced by Chow et al. (2020). Pixel perturbation attacks make small, distributed changes to the image at pixel level. These changes are typically imperceptible to the human eye, but can significantly confuse object detectors. TOG is one of the representatives in this family of attacks.

The reason why we decided to implement this specific attack is its wide recognition in the field. Many other works compare their results with TOG so as to evaluate effectiveness and robustness of their solutions (e.g. Zhou et al. (2024) and Wu et al. (2024)) – which is the reason why we decided it would be a valuable addition to the package.

A significant reason why TOG often serves as a benchmark is its robustness to different object detector architectures. Because TOG targets the fundamental, shared loss components of detection (objectness, bounding boxes, and classification), and does not manipulate any specific model structure, it is a broadly applicable attack.

Even though there are six different variations of TOG Chow et al. (2020), we decided to implement four of them, which we describe below.

The training procedure is as follows. The model weights of the victim detector remain fixed throughout the attack. The input image $\tilde{x}$ gets iteratively updated towards the goal of the selected variation of the TOG attack.

Let $\Pi_{\tilde{x},\epsilon}$ be the projection onto a hypersphere with radius $\epsilon$ centered at $\tilde{x}$ in $L_p$ norm, and clipped to the appropriate value range. Let $\Gamma$ be the sign function, and $\alpha_{TOG}$ be the attack learning rate. Furthermore, let $\mathcal{W}$ be the learnable model weights, $\hat{\mathcal{O}}(\tilde{x})$ be a set of object detector's outputs

for a given image $\tilde{x}$ (prior to applying any adversarial modifications), and $\mathcal{O}_t(\tilde{x})$ – a set of auxiliary target detections for image $\tilde{x}$ – constructed by setting each $\hat{o}$ in $\hat{\mathcal{O}}$ to its malicious class label. Last but not least, let $\mathcal{L}$ be the loss function, defined as in equation (2.6), and $\tilde{x}'_n$ denote an image $\tilde{x}$ with applied adversarial pixel perturbations, after $n$ iterations.

**TOG untargeted**. The goal of this variation of TOG is to perturb each pixel in the way which increases the loss the most – therefore, to cause the victim detector to *randomly* misdetect without targeting at any specific object. This attack is successful whenever the victim detector returns incorrect results of any form, whether this means object vanishing, object fabrication or random mislabeling.

$$\tilde{x}'_n = \Pi_{\tilde{x},\epsilon}\left[\tilde{x}'_{n-1} + \alpha_{\text{TOG}}\Gamma\left(\frac{\partial\mathcal{L}(\tilde{x}'_{n-1};\hat{\mathcal{O}}(\tilde{x}),\mathcal{W})}{\partial\tilde{x}'_{n-1}}\right)\right] \tag{2.9}$$

**TOG vanishing**. The goal of this variation of TOG is to make sure no candidate detections survive the thresholding process. Therefore, it iteratively pushes the input $\tilde{x}$ along the direction of producing an empty detection set. To achieve this movement, it exploits the objectness loss $\mathcal{L}_{\text{obj}}$, which is crucial for the decision on object existence:

$$\tilde{x}'_n = \Pi_{\tilde{x},\epsilon}\left[\tilde{x}'_{n-1} - \alpha_{\text{TOG}}\Gamma\left(\frac{\partial\mathcal{L}_{\text{obj}}(\tilde{x}'_{n-1};\emptyset,\mathcal{W})}{\partial\tilde{x}'_{n-1}}\right)\right]. \tag{2.10}$$

**TOG fabrication**. Contrary to TOG vanishing, this variation of the attack aims to fabricate additional detections on images. It maximizes the object existences by pushing the input image $\tilde{x}$ in the direction opposite to producing an empty detection set:

$$\tilde{x}'_n = \Pi_{\tilde{x},\epsilon}\left[\tilde{x}'_{n-1} + \alpha_{\text{TOG}}\Gamma\left(\frac{\partial\mathcal{L}_{\text{obj}}(\tilde{x}'_{n-1};\emptyset,\mathcal{W})}{\partial\tilde{x}'_{n-1}}\right)\right]. \tag{2.11}$$

**TOG mislabeling**. This variation of the TOG attack aims to steer the victim object detector into consistently misclassifying the detections, by replacing their source class label with the malicious label:

$$\tilde{x}'_n = \Pi_{\tilde{x},\epsilon}\left[\tilde{x}'_{n-1} - \alpha_{\text{TOG}}\Gamma\left(\frac{\partial\mathcal{L}(\tilde{x}'_{n-1};\mathcal{O}_t(\tilde{x}),\mathcal{W})}{\partial\tilde{x}'_{n-1}}\right)\right] \tag{2.12}$$

The target detection $\mathcal{O}_t(\tilde{x})$ is created by setting each object in the detection result to be of a maliciously chosen class label. Alternatively, it could be set as the **most-likely (ML)** class (by setting target label to be equal to $\arg\max_c \hat{p}^c$, selecting only from $c \neq \max_c \hat{p}^c$ – to obtain second most likely class label, as the first most likely is the ground truth) – or as the **least-likely (LL)** class (by setting target label to be equal to $\arg\min_c \hat{p}^c$).

### 2.1.3 Object Detection Defenses

#### Adversarial Training

Adversarial Training, as suggested by Szegedy et al. (2014b), is an effective and widely used defense method against adversarial attacks. While initially the aforementioned paper introduced it for image classification, this idea has been adapted for other tasks, such as object detection Zhang and Wang (2019).

The main idea is to train the model on both *benign* (clean) images, and *adversarial* images simultaneously. During training, adversarial images are generated using a selected attack method.

They are then matched with the correct labels (taken from clean images) and added to the training dataset. The model is then trained on such a mix of clean and adversarial images, effectively learning to ignore the adversarial perturbations and therefore becoming more robust against attacks at test time.

### 2.1.4 Evaluation Metric

#### Mean Average Precision (mAP)

As the evaluation metric of the success of our implemented attacks, we chose to track changes in the mean Average Precision (mAP) metric Everingham et al. (2010b), which is a standard metric used to evaluate the detection accuracy of object detectors. The success of the attack is therefore measured by the drop in mAP score – the bigger the drop, the better.

**Intersection over Union (IoU)**. To calculate mAP, first a threshold $\gamma_{IoU} \in (0, 1)$ is set to specify whether the detection is correct. This threshold determines the minimum value of the spatial overlap IoU between the predicted ($\hat{b} = (\hat{b}^x, \hat{b}^y, \hat{b}^w, \hat{b}^h)$) and ground truth ($b = (b^x, b^y, b^w, b^h)$) bounding boxes, defined as follows:

$$\text{IoU} = \frac{\text{Area}(\hat{b} \cap b)}{\text{Area}(\hat{b} \cup b)} = \frac{\text{Area}(\hat{b} \cap b)}{\text{Area}(\hat{b}) + \text{Area}(b) - \text{Area}(\hat{b} \cap b)} = \frac{b^w_{int} \cdot b^h_{int}}{\hat{b}^w \cdot \hat{b}^h + b^w \cdot b^h - b^w_{int} \cdot b^h_{int}}, \tag{2.13}$$

where the intersection's width $b^w_{int}$ is calculated as:

$$b^w_{int} = \max(0, \min(\hat{b}^x + \hat{b}^w, b^x + b^w) - \max(\hat{b}^x, b^x)), \tag{2.14}$$

and intersection's height $b^h_{int}$ as:

$$b^h_{int} = \max(0, \min(\hat{b}^y + \hat{b}^h, b^y + b^h) - \max(\hat{b}^y, b^y)). \tag{2.15}$$

Here we assume that for a rectangular bounding box $b = (b^x, b^y, b^w, b^h)$ – $b^x, b^y$ are the coordinates of the top-left corner, while $b^w, b^h$ are the width and height of the box respectively.

Each detection should then be classified into one of the three different categories: True Positive, False Positive or False Negative.

**True Positive (TP)** means a detection for which IoU $\geq \gamma_{IoU}$ and predicted class matches the ground truth.

**False Positive (FP)** means an incorrect detection; occurs when one of the three cases occurs: (1) IoU $< \gamma_{IoU}$, (2) IoU $\geq \gamma_{IoU}$ but the predicted class label is different from the ground truth, or (3) the object has already been detected and this is a duplicate.

**False Negative (FN)** means a case when a ground truth object is not detected – object is missed entirely or is detected with wrong class label, or with too small IoU.

While the explanations above apply to individual bounding boxes, we can aggregate these outcomes across the entire dataset for a specific class $k \in \{1, ..., K\}$, where $K$ – number of classes. Let $TP_k, FP_k, FN_k$ be such aggregated True Positive, False Positive and False Negative scores for class $k$.

Since mAP combines precision and recall to give a comprehensive measure of the detector's effectiveness, we need to define these metrics too.

**Precision** answers the question: *What percentage of the predictions were correct?* It can be calculated as follows: $Precision_k = \frac{TP_k}{TP_k + FP_k}$ – for a given class $k$.

**Recall** answers the question: *What percentage of the objects did we actually detect?* It can be calculated as follows for a given class $k$: $Recall_k = \frac{TP_k}{TP_k + FN_k}$.

Once we have obtained the $Precision_k$ and $Recall_k$ scores for every class $k \in \{1, ..., K\}$, we can plot the precision against recall for each class, obtaining a precision-recall curve. We then

approximate the area under this curve to determine the Average Precision $AP_k$ for a given class. This value provides an evaluation of the detector's performance for a given class $k$.

Lastly, we can calculate the mAP score of our model: $mAP = \frac{1}{K}\sum_{k=1}^{K} AP_i$.

## 2.2 Large Language Models

This section provides the theoretical background for Large language models (LLMs) and the areas that are relevant to the package. It starts with a brief overview of Large language models, how they work and the main goals they achieve. It then moves on to describing the implemented adversarial attack and explains the methods for fine tunign LLMs.

### 2.2.1 A Primer in LLMs

LLMs are large-scale Transformer neural networks, originating from the "Attention Is All You Need" paper Vaswani et al. (2017). They learn statistical regularities in language by being trained to predict the next token in large text corpora where a token is a subword or character-level unit primarily produced by a tokenizer. A important element needed to make next-token prediction possible is the incorperation of causal masking. During training, the text sequences are are processed with causal masking to ensure that at position $t$ the model only attends to tokens $x_{\leq t}$. This masking enforces the autoregressive learning objective: the model must generate each word based solely on prior context, mirroring natural language use.

During training, each discrete token $x_i$ is mapped into a continuous embedding space using an embedding matrix $E$, producing vectors $h_i = E[x_i]$ whose geometric relationships capture the semantic and syntactic structure. These embeddings then propagate through multiple self-attention and feed-forward layers to provide contextualized representations $z_t$. The model then projects $z_t$ onto the vocabulary to produce logits, unnormalized scores given by

$$\ell = W z_t + b, \tag{2.16}$$

which are transformed through a softmax function to produce the probability distribution over the next token,

$$p(x_{t+1} \mid x_{\leq t}) = \mathrm{softmax}(\ell). \tag{2.17}$$

During training the main objective is to minimize the cross-entropy loss,

$$\mathcal{L}_{\text{LM}} = -\sum_{t=1}^{T} \log p(x_{t+1} \mid x_{\leq t}), \tag{2.18}$$

which encourages the model to assign higher probability to correct continuations.

After this step (pretraining), LLMs are typically adapted through the next step of supervised fine-tuning (SFT) on instruction–response pairs $(x, y)$, optimizing

$$\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{|y|} \log p(y_t \mid x, y_{<t}), \tag{2.19}$$

which conditions the model to follow user queries more reliably. Additional fine-tuning procedures may further shape behavior for specific domains or interaction patterns.

At deployment time, users interact with LLMs through natural-language sentences called prompts. Prompt guide the model's internal architecture trajectory and thus influence token

probabilities and the actual generated outputs. To ensure responsible and safe operation, modern LLM systems employ safety constraints, for example, training-time filtering of sensitive examples, fine-tuned refusal behaviors for unsafe or disallowed instructions, and runtime mechanisms that detect harmful prompt patterns. These mechanisms aim to prevent models from producing unsafe, biased, or prohibited content to the user. However, such constraints are often vulnerable to adversarial prompts designed to bypass refusal behaviors, making the robustness of safety systems a central concern in modern LLM research.

### 2.2.2 Supervised Fine-Tuning (SFT) of Large Language Models

To adapt a pre-trained language model to specific tasks or behavioral requirements, it is common to perform **Supervised Fine-Tuning (SFT)**. This process is a key stage in building instruction-following or aligned LLMs and is the foundational technique used to go from a next-token predictor to a useful chatbot assistant.

The goal of SFT is to train the model on a dataset of prompt–response pairs so that it learns to generate outputs that align with human expectations or to follow task-specific objectives. Model parameters $\boldsymbol{W}$ are updated to maximize the probability of producing the correct next token given an input sequence.

Let $\tilde{x} = (x_1, \ldots, x_{T_x})$ denote an input text sequence and $\tilde{y} = (y_1, \ldots, y_T)$ the desired target token sequence. A standard supervised objective is the tokenwise cross-entropy loss (teacher forcing) over the target tokens:

$$\mathcal{L}_{\text{SFT}}(W) \;=\; -\frac{1}{T} \sum_{t=1}^{T} \log p_W\big(y_t \mid \tilde{x}, \tilde{y}_{<t}\big), \tag{2.20}$$

or, for a dataset $\mathcal{D}$ of examples,

$$\mathcal{L}_{\text{SFT}}(W) \;=\; -\frac{1}{|\mathcal{D}|} \sum_{(\tilde{x},\tilde{y}) \in \mathcal{D}} \frac{1}{T(\tilde{y})} \sum_{t=1}^{T(\tilde{y})} \log p_W\big(y_t \mid \tilde{x}, y_{<t}\big). \tag{2.21}$$

Extending the objective to a dataset $\mathcal{D}$ is necessary because model parameters are learned from many training examples, and averaging the loss over the dataset ensures that updates reflect overall performance rather than behavior on a single sequence.

#### SFT and Adversarial Robustness

Fine-tuning affects both robustness and vulnerability. Augmenting SFT data with adversarially perturbed examples can improve resistance to malicious inputs (adversarial training). On the other hand however, poorly curated fine-tuning data can introduce biases or backdoors into the model. Therefore, SFT can be used as both a defensive tool and a potential source of risk if not performed with careful data hygiene and robust validation.

#### Dataset and Formatting

Fine-tuning datasets typically consist of carefully constructed prompt–response pairs. Each example includes an instruction or query (the prompt) and the desired answer (the response). Datasets can be collected via human annotation, synthetic generation, or domain-specific sources.

To ensure the model distinguishes conversational roles, prompts and responses are often marked with special tokens or separators (e.g., `<|user|>`, `<|assistant|>`). These conversational role markers are not required for all fine-tuning tasks, but they are standard in dialogue-oriented training settings where the model must learn to separate user prompts from the model generated response. Defining role tokens (e.g., `<|user|>` and `<|assistant|>`) provide structural cues that help the model learn which parts of the sequence correspond to the user prompt and which part is expected to be generated. Without such markers, the model may lose the distinction between roles which in result can lead to unstable behavior such as repeating instructions, generating both sides of the dialogue, or failing to follow the intended interaction format.

### Training Procedure

During SFT the model starts from the pre trained weights which it achieved after the first training step and continues training with supervised teacher forcing: at each step the model conditions on ground-truth previous tokens when predicting the next token.

1. **Optimization.** Training commonly relies on gradient-based optimization methods, often from the Adam family

2. **Parameter-efficiency options.** Either update all model parameters (full fine-tuning) or use parameter-efficient methods such as LoRA Hu et al. (2022) prefix tuning, or adapter layers to reduce compute and memory cost.

3. **Stability techniques.** Mixed precision (FP16 Micikevicius et al. (2018)/BF16 Wang and Kanwar (2019)), gradient accumulation, and checkpointing are commonly used to scale to large models.

4. **Regularization & safety.** Data filtering, label smoothing, and curated validation checks help avoid degenerate or harmful behaviors.

### Parameter-Efficient Fine-Tuning

LoRA serves as the main parameter-efficient method, allowing large-scale models to be fine-tuned on limited hardware. Typical ranks range from 8–64, with $\alpha \approx 2r$ for balanced update scaling. Dropout regularization mitigates overfitting, especially on smaller set of classes.

QLoRA extends this by combining 4-bit quantization of base weights with full-precision adapters. This approach maintains model quality while significantly reducing memory usage, enabling fine-tuning of models with billions of parameters on single GPUs.

### Evaluation Metrics

The primary automatic metric for SFT is **perplexity**, which quantifies how well the model predicts held-out text. For a sequence $\tilde{x}$,$\tilde{y}$ the perplexity is defined as

$$\mathrm{PPL}(\tilde{y}) = \exp\Big(-\frac{1}{T}\sum_{t=1}^{T} \log p_W(y_t \mid \tilde{x}, \tilde{y}_{<t})\Big), \tag{2.22}$$

and dataset-level perplexity averages (or exponentiated negative log-likelihoods) are computed across evaluation examples. Lower perplexity indicates the model assigns higher probability to correct continuations.

In addition to perplexity, SFT quality is often measured by:

- **Task-specific metrics:** accuracy, F1, BLEU, ROUGE, or other downstream measures depending on the task.
- **Human evaluation:** fluency, helpfulness, and alignment checks.
- **Safety/robustness tests:** adversarial prompts, bias audits, and failure-mode analyses.

#### Implementation Using Hugging Face Trainer

In our project, fine-tuning was implemented using the Hugging Face `Trainer` API, which simplifies many fundamental components and supports reproducible experimentation. The `Trainer` manages tokenization, batching, and dynamic padding through dataset collators that automatically pad each batch to the length of its longest example. It also provides built-in support for mixed-precision and distributed training, enabling efficient use of FP16 or BF16 computation and seamless scaling across multiple GPUs or nodes. To accommodate limited GPU memory, the framework offers gradient accumulation and gradient checkpointing, allowing effective large-batch optimization. In addition, the `Trainer` integrates evaluation and logging utilities that facilitate periodic validation, metric tracking, and the use of common monitoring tools such as TensorBoard or Weights & Biases.

This choice aligned with our existing code, as our package already incorporates Hugging Face support for other attacks, making the `Trainer` a natural and consistent solution.

### 2.2.3 Adversarial Attacks on LLMs

Adversarial attacks on LLMs are different from those in computer vision primarily due to the discrete nature of text. Perturbations cannot simply modify pixels but must instead operate on artificial tokens while also maintaining grammatical and semantic coherence. Because of this, textual adversarial examples are often crafted through controlled token substitutions, prompt injections, or optimization in embedding space rather than direct gradient perturbations on raw input.

Typical examples of attacks on LLMs include:

- **Prompt injection and jailbreak attacks** – where the attacker inserts malicious instructions into the input prompt, tricking the model to ignore safety constraints or reveal hidden information.
- **Token-level perturbation attacks** – where single words, subwords, or characters are replaced with semantically similar or intentionally confusing tokens that cause the model to mispredict.
- **Data poisoning or backdoor attacks** – where corrupted or manipulated data is inserted into the fine-tuning dataset so that the model has a higher chance of producing unwanted text.
- **Model extraction or privacy attacks** – where an attacker issues repeated queries and reconstructs parts of the model parameters or sensitive training data.

Adversarial evaluation of LLMs is typically based on changes in task-specific performance metrics such as accuracy, perplexity, or the success rate of targeted manipulations. The greater the performance degradation or violation rate under attack, the more effective the adversarial method is considered.

## Greedy Coordinate Gradient Attack

One of the adversarial attacks we implemented as part of our package is the **Greedy Coordinate Gradient (GCG)** attack Zou et al. (2023). This targeted white-box attack adapts gradient-based optimization techniques to the discrete token space used by LLMs and is designed to efficiently find minimal perturbations that significantly alter the model's behavior.

## Overview and Intuition

GCG is an iterative discrete attack that uses token-level gradient information to propose and evaluate candidate replacements for an adversarial suffix. The adversarial suffix is a sequence of tokens appended to the user's prompt where the tokens are iteratively modified so that the model's output is pushed toward a specified adversarial target. At each iteration, GCG estimates how changing individual tokens in the suffix would affect the model's loss by computing the gradient of the loss with respect to the one-hot representation of each suffix token. For a suffix position $t$, the method computes

$$\nabla_{e_t} \mathcal{L}(f(x_{1:n} \,\|\, s),\, y_{\text{target}}) \in \mathbb{R}^{|V|}, \tag{2.23}$$

where $e_t$ is the one-hot vector representing the token at position $t$ in the suffix, $|V|$ is the vocabulary size, $f$ is the language model, and $y_{\text{target}}$ denotes the desired adversarial output (e.g., a harmful instruction or any text pattern the attacker wants to achieve). Each coordinate

$$\frac{\partial \mathcal{L}}{\partial e_{t,i}} \tag{2.24}$$

corresponds to the effect of replacing the current token at position $t$ with vocabulary item $i$. Large magnitude gradient coordinates therefore indicate high-impact candidate substitutions.

Using the top-$k$ highest-magnitude coordinates as candidate tokens, GCG constructs $B$ perturbed suffix proposals, evaluates them with forward passes, and greedily commits to the proposal with the lowest loss. This iterative refinement continues until the adversarial target is achieved or for a selected amount of iterations. The method also supports multi-prompt aggregation, enabling the construction of universal suffixes that transfer across prompts.

To formalize the attack, let $\mathbf{x}$ denote the original prompt and $\mathbf{s}$ the adversarial suffix.The modified input is $\mathbf{x}' = \mathbf{x} \,\|\, \mathbf{s}$, and the goal of GCG is to steer $f(\mathbf{x}')$ toward a predefined adversarial target output $y_{\text{target}}$. The optimization objective used by GCG can be written simply as

$$\min_{\mathbf{s}} \; \mathcal{L}\big(f(\mathbf{x} \,\|\, \mathbf{s}),\, y_{\text{target}}\big), \tag{2.25}$$

where $\mathcal{L}$ is a differentiable loss function measuring how closely the model output matches the adversarial target (e.g., negative log-likelihood of a target prefix or the log-probability of a targeted answer class).

## Characteristics and Advantages

GCG offers several practical advantages for constructing effective attacks. First, it achieves high efficiency by generating multiple candidate suffixes in parallel and evaluating them in a batch, allowing the method to use GPU parallelism and converge quicker than other approaches that update one token at a time. Second, its gradient-based token selection provides a degree of interpretability: the gradient magnitudes highlight which suffix positions has the greatest influence on the model's vulnerability, revealing where the attack is most effective. Third, GCG is capable of producing universal adversarial suffixes that transfer across prompts or even across models,

making it useful for robustness benchmarking and large-scale security evaluations of llm systems. Finally, the method is flexible, supporting both targeted attacks on individual prompts and multi-prompt optimization when constructing transferable attack suffixes.

### Applications

- **Robustness evaluation.** GCG provides a systematic framework to measure how different architectures, fine-tuning approaches, and safety mechanisms withstand gradient-based adversarial attacks.
- **Vulnerability analysis.** The attack highlights recurring patterns in successful adversarial examples, which informs defensive research and responsible disclosure.
- **Adversarial training data.** High-quality adversarial examples generated by GCG can be used as training data to improve model robustness and evaluate defensive techniques under realistic attack scenarios.

## 2.3 Differential Privacy

Differential Privacy (DP) is a standard used to quantify and limit privacy leakage in data analysis. Its primary goal in this context is to ensure that machine learning models do not memorize sensitive details from their training data, thereby protecting against threats such as membership inference attacks. The following subsections outline the history of privacy definitions, introduce the core mathematical concepts, and describe our implementation of Differentially Private Stochastic Gradient Descent (DP-SGD) using the Opacus library.

### 2.3.1 Historical Context and Statistical Origins

The concept of Differential Privacy emerged from the field of *Statistical Disclosure Control (SDC)*, a discipline historically concerned with allowing statistical agencies (such as census bureaus) to release aggregate data without compromising the privacy of respondents. Early approaches relied on ad-hoc techniques such as data swapping, suppression, or adding arbitrary noise to outputs. However, these methods lacked a rigorous mathematical foundation and often failed when subjected to modern reconstruction attacks.

A turning point occurred with the "Fundamental Law of Information Recovery," established by Dinur and Nissim Dinur and Nissim (2003). They demonstrated mathematically that revealing accurate answers to a sufficient number of statistical queries inevitably allows an adversary to reconstruct the underlying database $\mathcal{D}$ with high probability. This proved that absolute privacy is impossible if utility (accuracy) is to be maintained. Consequently, the focus shifted from binary definitions of privacy (e.g., "is the data anonymous?") to limiting the *risk* associated with participation. This line of inquiry culminated in the work of Dwork et al. Dwork et al. (2006), who formalized this stability guarantee as *Differential Privacy*, providing a provable bound on how much the presence of a single individual can influence the outcome of an analysis.

### 2.3.2 A Primer in Differential Privacy

Differential Privacy (DP) is a formal criterion designed to limit the information an algorithm can leak about any individual record in a training dataset $\mathcal{D}$. Informally, training on two neighbouring

datasets—$\mathcal{D}$ and $\mathcal{D}'$, which differ by exactly one record—should lead to output distributions that are statistically indistinguishable.

Formally, a randomized mechanism $\mathcal{M}$ is $(\varepsilon, \delta)$-differentially private if, for all measurable subsets $S$ of the mechanism's possible outputs:

$$\Pr[\mathcal{M}(\mathcal{D}) \in S] \leq e^{\varepsilon} \cdot \Pr[\mathcal{M}(\mathcal{D}') \in S] + \delta, \tag{2.26}$$

where $\Pr$ denotes probability and $e$ is Euler's number Dwork et al. (2006). The privacy guarantee is strictly governed by the magnitude of $\varepsilon$ and $\delta$. There exists an inverse relationship between $\varepsilon$ and privacy strength: a lower value of $\varepsilon$ (approaching 0) yields indistinguishable output distributions, providing maximal privacy at the potential cost of model utility. Conversely, a higher $\varepsilon$ allows for greater divergence between distributions, resulting in weaker privacy protection; in the limit where $\varepsilon$ is large, the mechanism provides little to no protection against inference.

The parameter $\delta$ relaxes this condition, representing the "failure probability" of the pure $\varepsilon$-privacy bound. While $\varepsilon$ bounds the probability ratio in the standard case, $\delta$ accounts for rare events—such as the extreme tails of a Gaussian distribution—where this bound may not hold. Crucially, the magnitude of $\delta$ determines the risk of a catastrophic privacy breach. If $\delta$ is comparable to the inverse of the dataset size (i.e., $\delta \approx 1/|\mathcal{D}|$), the mechanism could theoretically reveal a single record with high probability while still satisfying the definition. Therefore, to ensure a meaningful guarantee, $\delta$ must be kept cryptographically negligible, typically satisfying $\delta \ll 1/|\mathcal{D}|$ (e.g., $10^{-5}$ or lower).

To make an exmaple, consider a small database of four salaries: $\mathcal{D} = \{30\text{k}, 50\text{k}, 60\text{k}, 150\text{k}\}$. The true average of this dataset is 72.5k. If we remove the highest earner to create a neighbouring dataset $\mathcal{D}' = \{30\text{k}, 50\text{k}, 60\text{k}\}$, the average drops significantly to $\approx 46.7$k. In a non-private system, this large discrepancy (25.8k) would immediately reveal the presence of the high-income individual. Differential Privacy addresses this by adding calibrated noise to the output. With a budget of $\varepsilon = 0.1$, the mechanism might output a noisy result, such as 60k. Crucially, under the $e^{0.1} \approx 1.105$ probability bound, an adversary observing this 60k output cannot determine with certainty whether it came from the distribution centered at 72.5k (Database $\mathcal{D}$) or the one centered at 46.7k (Database $\mathcal{D}'$). The noise essentially "smears" the two distributions so they overlap, providing the individual with plausible deniability.

### 2.3.3 DP-SGD with Opacus

To enforce these guarantees during neural network training, we utilize *Differentially Private Stochastic Gradient Descent (DP-SGD)* Abadi et al. (2016), provided by the Opacus library. While standard SGD updates model weights based on the average gradient of a minibatch $B$, DP-SGD requires bounding and obfuscating the influence of each individual sample $x \in \mathcal{D}$.

The training process is modified as follows:

1. **Per-Sample Gradient Computation:** Instead of aggregating gradients immediately, the algorithm computes per-sample gradients $g_i = \nabla_\theta \mathcal{L}(x_i, y_i)$ for each sample $i$ in the current batch.

2. **Gradient Clipping:** To bound the sensitivity of the update step to any single record in $\mathcal{D}$, each per-sample gradient $g_i$ is clipped to a maximum $\ell_2$ norm $C_{\text{clip}}$:

$$\bar{g}_i = g_i \cdot \min\left(1, \frac{C_{\text{clip}}}{\|g_i\|_2}\right). \tag{2.27}$$

We use the notation $C_{\text{clip}}$ here to distinguish this hyperparameter from the object confidence score $C$ defined in the Object Detection chapter.

3. **Noise Injection:** The clipped gradients are aggregated and perturbed with Gaussian noise. The optimizer performs the update using the noisy gradient $\tilde{g}$:

$$\tilde{g} = \frac{1}{|B|} \left( \sum_{i \in B} \bar{g}_i + \mathcal{N}(0, \sigma^2 C_{\text{clip}}^2 \mathbf{I}) \right), \tag{2.28}$$

where $\sigma$ (the noise multiplier) controls the standard deviation of the noise relative to the clipping bound.

**Optimizer Compatibility.** Although the algorithm is often termed DP-SGD, the privacy mechanism is inherently optimizer-agnostic. The clipping and noise injection occur on the gradients *before* the parameter update step. Consequently, the resulting private gradient $\tilde{g}$ can be consumed by any first-order optimizer. Algorithms such as SGD, Adam, or AdamW remain fully compatible; they receive $\tilde{g}$ in place of the standard batch gradient and proceed with their specific update rules (e.g., momentum, adaptive learning rates) without violating the differential privacy guarantee, provided the internal optimizer state depends only on the private gradients.

### Privacy Accounting

The privacy loss accumulates with every gradient step. To track this, Opacus employs a *privacy accountant*, specifically using Rényi Differential Privacy (RDP) analysis Mironov (2017). This accountant computes the total $\varepsilon$ expended for a fixed $\delta$ over the course of training epochs. There is an inherent trade-off: a larger noise multiplier $\sigma$ or a stricter clipping norm $C_{\text{clip}}$ yields stronger privacy (lower $\varepsilon$) but may degrade model utility by introducing variance or bias into the optimization.

## 2.4 Extensible Loading of Custom Models & Datasets

Evaluating adversarial robustness effectively requires the ability to test attacks across diverse model architectures and datasets without modifying the core codebase. This section introduces the concept of "extensible loading," a design pattern that decouples experiment logic from asset management. We first define the mechanism for referencing external resources symbolically and then detail its practical realization using the Hugging Face platform.

### 2.4.1 A primer in Extensible Loading of Custom Models Datasets

"Extensible loading" denotes a mechanism by which models and datasets are referenced symbolically rather than embedded directly. Instead of relying on assets fixed in a codebase, the system accepts standard identifiers that point to external resources. An experiment is therefore described by names and versions of artefacts rather than by local files. The configuration specifies which model and which dataset are intended; the loading mechanism resolves those references, acquires the corresponding artefacts, and makes them available to the training and evaluation routines. The process is agnostic to any one architecture or corpus: it treats models and datasets as external, named resources that can be looked up at run time. Where relevant, identifiers may encode versioning information, and datasets may include conventional split labels. The only assumption imposed at this stage is compatibility with the intended task (for example, image classifiers that accept the expected input shape and produce the required label space). Because the existing framework on which this project is built executes models within a PyTorch runtime, externally

referenced models are required to be PyTorch-compatible; this constraint pertains to models only and does not impose any restriction on the provenance or format of datasets.

### 2.4.2 Hugging Face Platform

The Hugging Face Hub acts as the registry for these references Hugging Face (2024d). Models and datasets are addressed by unique identifiers on the Hub; when such an identifier is provided, it is interpreted as a request for a particular artefact snapshot Hugging Face (2024e). For models, the identifier denotes an architecture family together with a parameter checkpoint; for datasets, it denotes a named collection with its schema and, when applicable, predefined splits or configurations Hugging Face (2024g,b,c). Identifiers can include tags or commit-like revisions to select a specific version, and accompanying cards supply descriptive metadata Hugging Face (2024f,g,b). The same identifier can be reused across runs to refer to the same published artefact, with no assumptions beyond network access and the usual constraints of the target task Hugging Face (2024d).

Hugging Face is used in this role because it maintains a large, openly accessible catalogue addressed by stable, versioned identifiers. In 2025, an official ecosystem overview reported over 1.8 million models and about 450,000 datasets on the Hub; these counts refer to the overall catalogue rather than a particular framework, and there is no public breakdown of how many hosted models are PyTorch-compatible Ghosh et al. (2025). We selected this platform because this vast, openly accessible catalogue has established itself as the de facto standard for model sharing. Leveraging this ecosystem ensures that our toolkit allows researchers to benchmark adversarial attacks against the widest possible range of state-of-the-art architectures without manual file management. Open publishing is supported: any account holder can create model or dataset repositories, which makes the identifiers practical handles for community-contributed artefacts Hugging Face (2024a).

For comparison with other platforms that also host PyTorch models and datasets, consider Kaggle and ModelScope. Kaggle's homepage listed 558,000 datasets (noted in October 2025), and ModelScope's datasets page reported 21,148 datasets (noted in October 2025) Kaggle (2025); ModelScope (2025).

#### Models

Model loading via identifiers follows a simple pattern. A model is denoted by a name that resolves to an architecture and set of weights hosted on the Hub. On resolution, the artefact is fetched and presented to the rest of the pipeline as the task's target model. This process loads only the model architecture and parameters. It does not include processing components, such as tokenizers or image scaling settings. Therefore, the data pipeline must be configured independently to ensure inputs match the model's expected format. Nothing in this step fixes which families are allowed; the only requirement is that the resolved model conforms to the interface expected by the downstream routines, such as input dimensionality and output format. For example, a classifier identifier is expected to resolve to a network that accepts the declared image shape and produces logits over the declared set of classes.

#### Datasets

Dataset loading proceeds analogously. A dataset is denoted by a Hub identifier that resolves to a specific collection together with its metadata. If the source defines canonical splits (e.g., "train", "test", "validation"), the configuration may name one or more of these, and the loader

retrieves the corresponding subsets. The dataset's schema—such as image tensors and class labels—remains as defined by the source, and must match the model's expectations for the planned task. Moving between compatible datasets (for instance, CIFAR-10 and SVHN in image classification, both 32×32 RGB with ten classes) amounts to changing the identifier while leaving the rest of the experiment description unchanged.

# Chapter 3

# Implementation

## 3.1 Overall Architectural Changes (Folder Structure)

We refactored the repository to support a broader range of adversarial tasks beyond the initial scope of image classification. Originally, the codebase followed a flat structure without a separate folder for computer vision tasks, as this was the only supported domain. Therefore, the folders for attacks and defenses were directly added under the advsecurenet folder. Because of our expansion, we introduced a domain-specific hierarchy, moving the existing logic into a dedicated `computer_vision/image_classification` module and adding new task roots for `object_detection/` and `llm/`. This separation allows task-specific components—such as bounding box processors for detection or tokenizers for LLMs—to coexist without polluting the generic interfaces (see Figure 3.1).

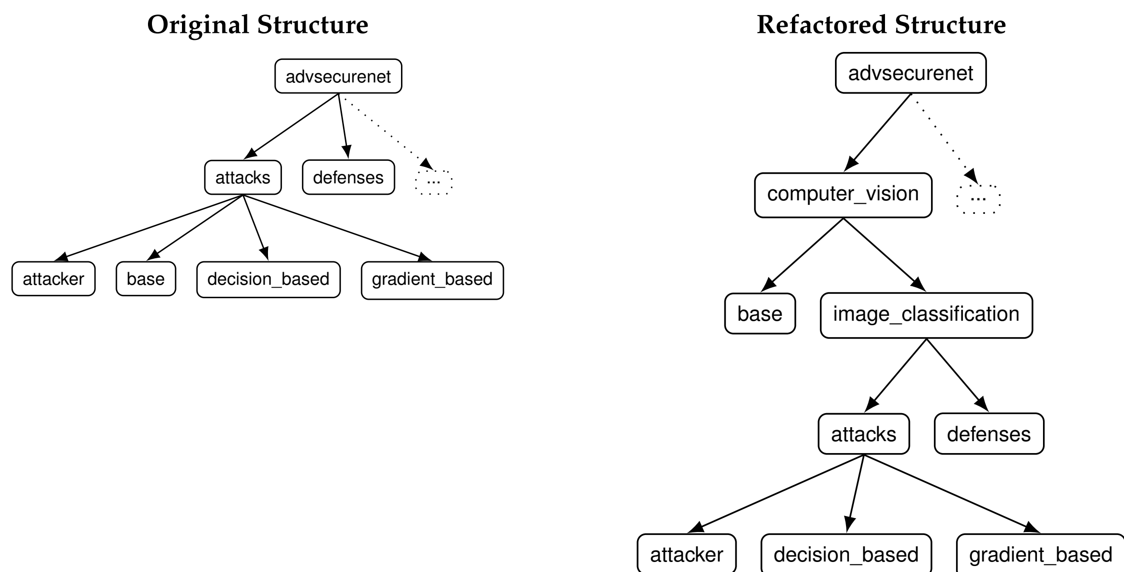



Figure 3.1: Comparison of the file structure architecture. The diagrams illustrate the transition to a domain-specific hierarchy, with dotted boxes indicating other existing modules omitted for clarity.

To accommodate the new capabilities, we introduced dedicated modules for Object Detection and Large Language Models. The object detection module (Figure 3.2) isolates detection-specific logic, organizing attacks (like adversarial patch and pixel perturbation) and defenses within their own namespace, separating them from standard classification tasks.

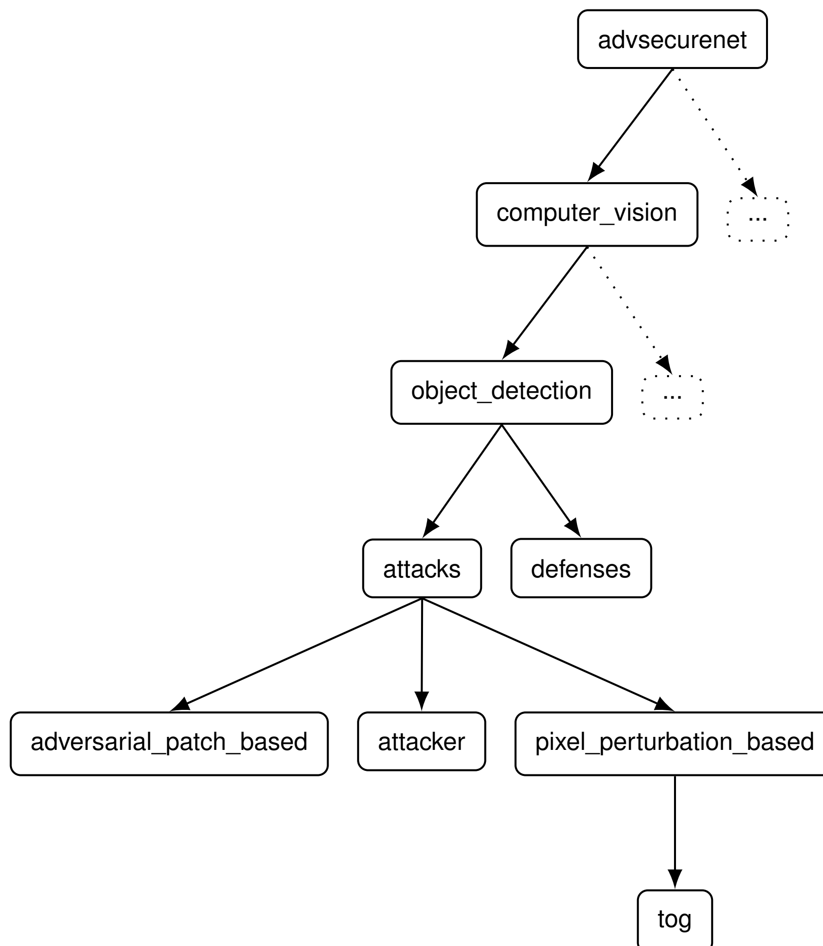



Figure 3.2: File structure of the newly added object detection module. The new object detection extension are under the computer vision folder and the structure below the object detection folder was mimiked from the image classification folder.

Similarly, the LLM extension (Figure 3.3) establishes a dedicated environment for text-based security. It houses the Greedy Coordinate Gradient (GCG) attack and fine-tuning configurations under a unified root, ensuring that the complex dependencies required for language models do not interfere with the computer vision components.

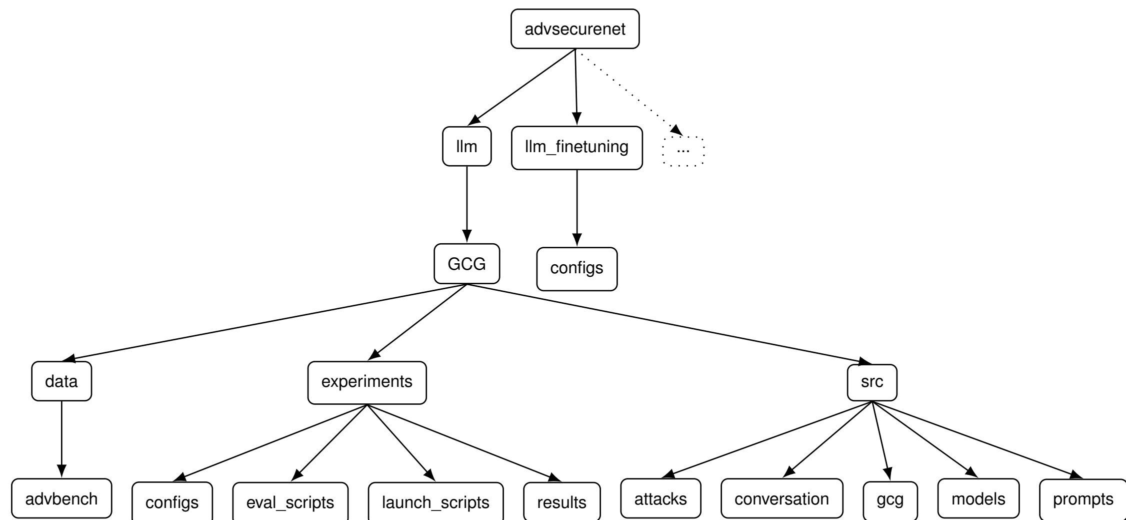



Figure 3.3: File structure of the newly added LLM extension. Because this extension is completely different two the other features the filestructure has no similarities to the computer vision extensions.

In addition to structural reorganization, we improved code maintainability by following the Don't Repeat Yourself (DRY) principle. We identified that configuration logic was often duplicated between the Command Line Interface (CLI) and the core library. As illustrated in Figure 3.4, we extracted this redundant logic into a shared base configuration layer within `advnet_common`. Both the CLI and library configurations now inherit from this base, which allows the CLI configuration to be directly assigned to the internal library dataclass. This direct assignment replaces the previous pattern of manually unwrapping content, resulting in a significantly cleaner interface boundary.

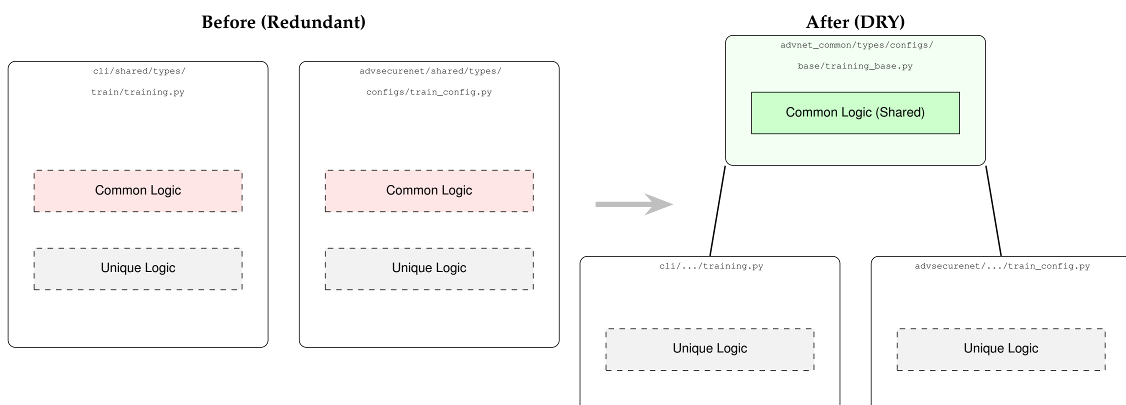



Figure 3.4: Visualisation of the refactoring process: redundant code extracted to a base class. The advnet_common folder was created to keep all the shared code which both the advsecurenet and the cli folder can inherit from.

## 3.2 Work Organization

### 3.2.1 Software Engineering Practices

We decided to follow a *Test-Last Development* approach, as it provides the most flexibility during the implementation phase, allowing us to freely and iteratively explore and refine implemented features, without the need to work under constraint of tests. This methodology allows for a quicker progress and adaptability, which are particularly valuable in the research areas and agile environments, where requirements or implementation details may often change during development. Furthermore, Test-Last Development aligns well with our workflow, and allows for efficiently balancing quick and efficient code creation with a focus on the robustness and error-proofness of the written code, as testing is performed right after the implementation.

### 3.2.2 Project Management

When working on the project, we decided to follow *Agile methodology*, as the dominant methodology currently used in the industry. We held weekly catch-up calls within our team, and worked in the 2-3 week sprints. Furthermore, after most of the sprints we scheduled a call with our supervisor to update him on our progress, as well as agree on next steps and directions.

In order to efficiently split the project into subtasks and track the progress, we set up Azure-DevOps Boards as our project management tool. We also transferred all the tasks to GitHub issues to facilitate for the GitHub users the usage of our package. Furthermore, we hosted our code repository on GitHub, as this is where the current version of the package is hosted

### 3.2.3 Branching Strategy

We adopted the Gitflow workflow for source control. We grouped features, and each group was addressed in a dedicated branch (including corresponding unit tests), which was then merged into dev branch. Our branching strategy is represented on Figure 3.5.

All pull requests underwent two peer reviews and addressed approval from both reviewers prior to merging.

It is also worth noting that, initially, when planning the project, we assumed a branching strategy of *one branch per one issue*. However, as we started developing the code, we realised there are plenty of intersections between some of the issues, and we decided that the smoothest and most efficient way to tackle the task is to work on the strongly correlated issues simultaneously.

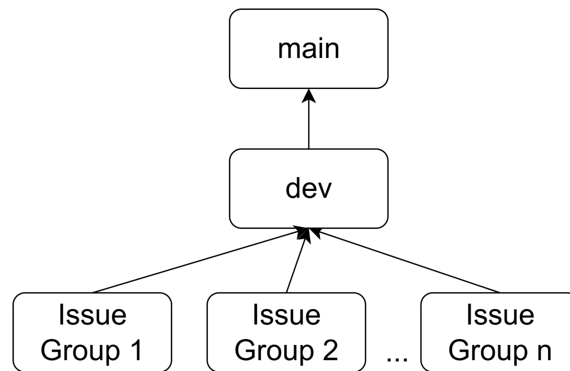



Figure 3.5: Adopted branching strategy. Issues are split into groups, based on the intersection/-correlation between them. After the completion of the given issue group, they are then merged onto the *dev* branch. Finally, the *dev* branch is merged into the *main* branch, which ends the development cycle.

### 3.2.4 Unit Tests and Quality Assurance

Quality assurance centered on automated unit testing with a single quantitative target: code coverage. Coverage was chosen because it is the metric already tracked by the existing codebase and therefore defines continuity with prior requirements. Tests were written and executed with pytest, and the suite emphasizes small, isolated checks over end-to-end scenarios so that behavior is validated close to where it is defined, including configuration parsing, factories, and supporting utilities. Before this work, the repository reported 95% code coverage; after the additions and refactors, the suite was extended to exercise the new paths while maintaining the overall coverage at 95%. In practice, this meant adding focused tests alongside each newly introduced component and updating existing tests when interfaces were clarified, so that the coverage metric remained stable and meaningful. The result is a test suite that preserves the codebase's established coverage level while providing direct checks for the extended functionality.

## 3.3 Object Detection

This section provides implementation details of the object detection extension. It covers the integrated datasets, and object detectors, as well as provides implementation details of the attacks and defense. It also contains a brief note on the architecture evolution over time.

All of the implemented attacks, as well as adversarial training and evaluation metric, are available both through Python package interface, as well as through CLI commands.

### 3.3.1 Datasets

We added support for two of the most popular object detection datasets: COCO Lin et al. (2015) and PascalVOC Everingham et al. (2010a).

#### Architecture Overview

An overview of the dataset class hierarchy is depicted in Figure 3.6. Both dataset-specific classes inherit from the base `BaseDataset` class, which inherits from `torch.utils.data.Dataset`

class. The BaseDataset class was already implemented in the package. Our extension included adding PascalVOCDataset and COCODataset classes, and implementing all necessary methods for them.

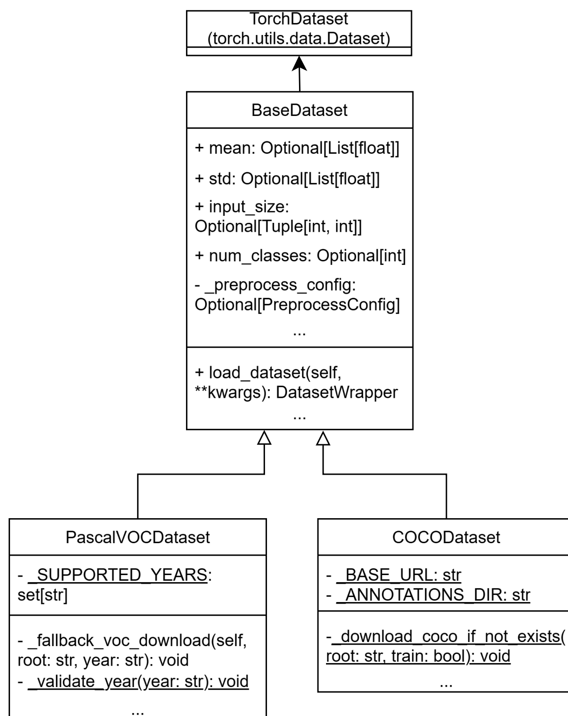



Figure 3.6: Object detection dataset architecture. The abstract BaseDataset class extends PyTorch's TorchDataset and contains common data loading interfaces and preprocessing configurations. PascalVOCDataset and COCODataset classes inherit from this base class, introducing dataset-specific logic.

## COCO

COCO, which stands for Common Objects in Context, is the main dataset used for our implementation, as it is extensively used (Singh et al. (2024)) in research. Most of the object detectors described in Section 3.3.2 were trained on this dataset. COCO is a large-scale object detection, segmentation and captioning dataset. It contains 80 different object categories, including areas such as food, appliances, furniture or sports. It often features objects of everyday scenes with multiple objects and is known for its complexity, which makes it a challenging benchmark, even for modern object detectors.

COCODataset class stores _BASE_URL, which is the URL to the website from where the dataset can be downloaded – and the _ANNOTATIONS_DIR, which is the name of the annotations folder.

Apart from the custom implementation of the load_dataset method, inherited from the base class, COCODataset implements a static method _download_coco_if_not_exists, that first searches the root directory for the appropriate folder – and if none such exists, it triggers a download from the official COCO website. Initiating this automated download implies acceptance of the COCO Terms of Use, requiring users to independently verify that their intended utilization complies with the varied copyright licenses of the source images.

### PascalVOC

PascalVOC (Visual Object Classes) is another popular dataset commonly used for benchmarking object detection models. It contains 20 different object classes, and consists of two subsets – challenge VOC2007 and challenge VOC2012.

`PascalVOCDataset` class stores `_SUPPORTED_YEARS`, which is a set of years for which there exists a version of the dataset.

Similarly as for the `COCODataset`, additionally to the custom implementation of the `load_dataset`, `PascalVOCDataset` implements a static `_validate_year` method, which checks if the year specified in the dataset configuration by the user is one of the years for which there exists a version of the dataset, and a `_fallback_voc_download` method, which checks if the dataset has already been downloaded, and if not, downloads the dataset.

## 3.3.2 Object Detectors

### Architecture Overview

Because most of the object detectors, including the ones selected for integration into the package, do not follow a unified input-preprocessing-output logic, a custom class for each of the models is required. A shared abstract class `CustomODBaseModel` is defined, from which every class for a specific object detector should inherit. Additionally, a `ODWrapper` class holds the core of the shared logic between the object detectors. Three object detectors are currently integrated into the package, wrapped in custom, detector-specific classes responsible for parsing the input-output, calculating gradients etc.: `CustomYolov5Model`, `CustomFasterRCNNModel`, `CustomRTDetrModel`. Class relationships are depicted on Figure 3.7.

It is also worth noting that currently supported object detectors are just example files demonstrating on how to integrate one's object detector into the package. If the user would like to use their own object detector, they should simply implement a custom class for that model, inheriting from the `CustomODBaseModel`, following the same logic as can be seen in the three example object detectors for which the custom classes have been already implemented. We decided on this form of extending the package to the custom models because of the discrepancies from one object detector to another of required input types, provided output formats, loss calculations etc. Therefore, we thought that the best way is to simply enable the user to define the specifics of the model they would like to use themselves, while still providing an interface they should implement.

### `CustomODBaseModel` class

It is an abstract class which determines the standardized interface which all object detection models added to the package should implement. Its main responsibility is to ensure that the object detector can be successfully plugged into the `ODWrapper` and used by the attack, adversarial training and evaluation features.

**Key Methods**. `forward` is the core method, used both for training and inference. It takes as an input a preprocessed image tensor and a preprocessed target dictionary (optional) and it should return a dictionary of losses during training or a list of detections during inference. Another important method is `initialize_inference_model`, which takes the model to wrap, a device, and a confidence threshold (necessary for the non-max suppression) as input and returns a model configured for inference. `prepare_training_inputs` is a utility function which is responsible for adapting inputs into the specific format required by the model's training forward pass (e.g., converting targets to a single tensor for YOLOv5 or ensuring a list of tensors for torchvision models). `predict_raw` is a critical method for attacks, as it returns the raw model outputs necessary for gradient calculation. Last but not least, `translate_predictions_for_map_evaluator`

is used to convert the model's detection output into the standardized list-of-dictionaries format required by the mAP (mean Average Precision) evaluator.

### `ODWrapper` class

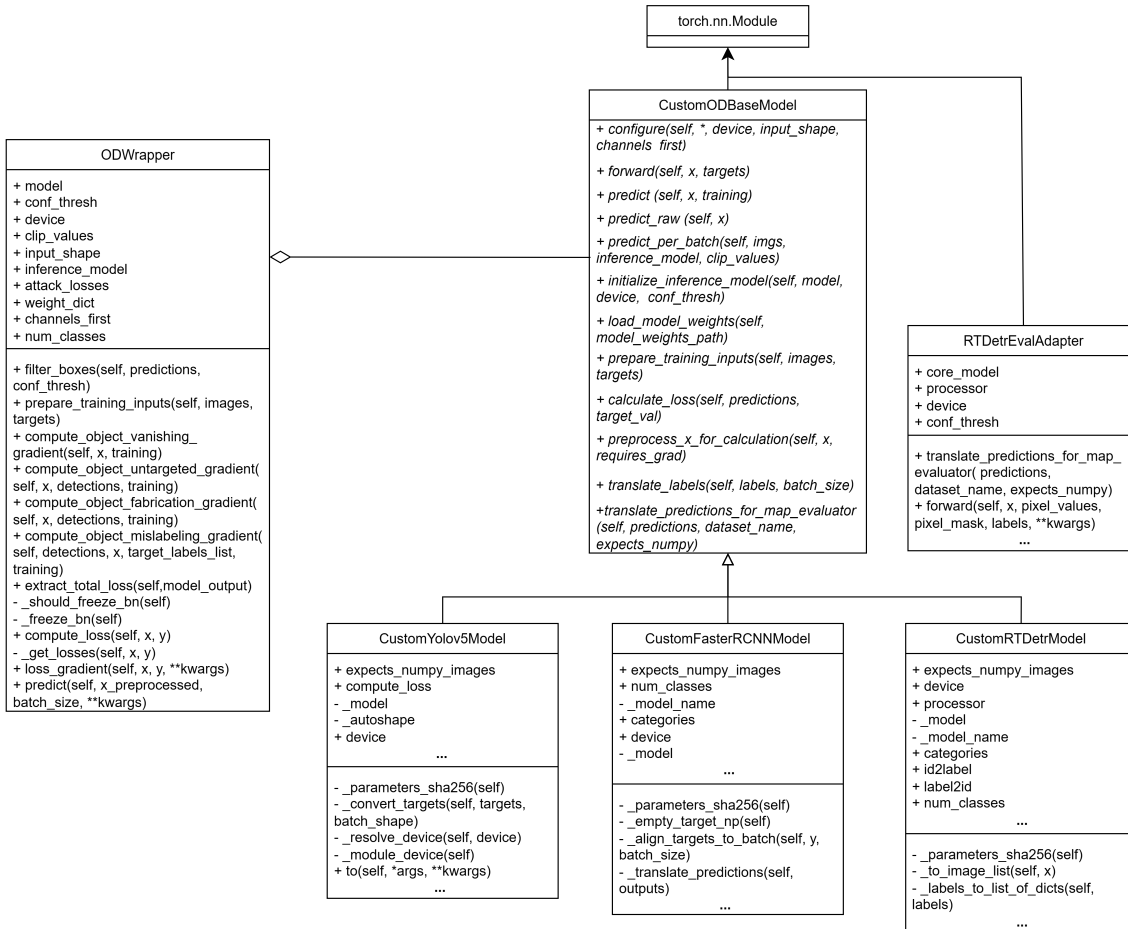



Figure 3.7: Object detectors class structure. The `ODWrapper` class is responsible for bridging model-specific logic and task-specific requirements. It contains an instance of custom model class, implementing abstract `CustomODBaseModel` interface. Currently, the package contains three custom model classes: `CustomYolov5Model`, `CustomFasterRCNNModel`, `CustomRTDetrModel` – all of which handle model-specific tasks such as data preprocessing, loss calculation etc. An additional helper class `RTDetrEvalAdapter` is introduced for `CustomRTDetrModel`.

Furthermore, an adapter class `ODWrapper` is defined, to bridge the gap between attack/training logic and a specific object detector model implementation. It stores two instances of the selected object detection model (`model`, wrapped in a custom class logic, and `inference_model`, which is the underlying object detector set to evaluation mode). `model` implements the `CustomODBaseModel` class interface, as it is necessary to enable e.g. gradient calculations and loss extraction. These tasks are not relevant for inference mode – therefore the `inference_model` is simply an instance of the underlying object detector model, with no custom wrappers applied. Having two distinct model instances, one in training mode (`model`), with custom gradient and loss calculation logic, and the other one in evaluation mode (`inference_model`) avoids constant mode

flipping, and prevents potential bugs or side effects, such as Batch Norm drifting – as the model instance has frozen Batch Norm overrides set). Furthermore, the ODWrapper contains all universal logic, which is shared between different object detectors and is necessary to run attacks, evaluation, adversarial training etc.

**Key Parameters and Methods**. The model parameter stores the underlying object detector implementing the training, inference and loss-calculation logic – inheriting from the CustomODBaseModel class. The inference_model is an inference-only instance initialized from model via initialize_ inference_model method – and used inside the predict method for forward passes and post-processing, in inference mode.

The standardized prediction logic is entailed in the predict method. It runs the self. inference_model on the chosen device, batch by batch, and returns NumPy detections filtered by the confidence threshold. loss_gradient computes the gradient of the attack loss with respect to the preprocessed input (using self.model.preprocess_x_for_loss_calculation and self.model. translate_labels), which is then used by image-wise adversarial attacks. Furthermore, ODWrapper provides TOG-related methods for gradient calculation, such as compute_object _vanishing_gradient, compute_object_mislabeling_gradient etc. Their main responsibility is to handle the logic for every implemented TOG variant, calling appropriate loss functions for each variant. The class also implements method required by AdversarialODTraining, e.g. prepare_ training_inputs and extract_total_loss, bridging the functionalities required by these tasks with the model-specific implementations – by appropriately orchestrating and delegating them to the underlying object detector instance stored in model.

### CustomYolov5Model class

This is a customized class for the YOLOv5 object detector,[1] written based on the IBM ART package code[2]. It inherits from the CustomODBaseModel class. The YOLO family of object detectors is very popular in the domain of object detection which is why we decided to incorporate one of these models into the package. We provide an example custom model class for this specific version of YOLO detector, as it offers a great balance between showcasing the effects of the adversarial attacks and offering the performance of modern object detectors – as newer models could be more robust to the attacks (for example, YOLOX outperforms earlier models, as showed by Apostolidis and Papakostas (2025)), and older models could no longer be comparable to the state-of-the-art ones. Figure 3.7 describes the diagram of the custom class dedicated for YOLOv5 and its relations to other classes.

**Key Parameters and Methods**. The _model and _autoshape parameters store: the underlying pytorch model, and the model wrapped in the Autoshape class respectively. expects _numpy_images is boolean flag specifying what input is expected by the object detector; here set to *True*. The compute_loss stores custom loss computation logic, taken from ComputeLoss function from the package. CustomYolov5Model, as a subclass of CustomODBaseModel, provides concrete implementations for all required abstract methods described above. Furthermore, it contains additional YOLOv5-specific utility functions, such as _convert_targets (converts the targets in the default dictionary-like format to YOLO-style format) or to method override (moves both the internal YOLOv5 model and its AutoShape wrapper to a specified device, and rebuilds the YOLOv5-specific ComputeLoss object on that device).

[1] https://github.com/fcakyon/yolov5-pip
[2] https://github.com/Trusted-AI/adversarial-robustness-toolbox

### CustomFasterRCNNModel class

This is a customized class for the FasterRCNN object detector[3] from the torchvision.models.detection module. FasterRCNN is another very popular object detector architecture, alongside previously described YOLO family. It is often used in benchmarks and as a reference point for new architectures, which is why we decided it would be a valuable addition to the package. Figure 3.7 describes the architecture of the custom model class for this object detector and its relations to other classes.

**Key Parameters and Methods**. Similarly to YOLOv5, the _model parameter stores the underlying actual object detector – here a fasterrcnn_resnet50_fpn_v2 model from torchvision. expects_numpy_images is a boolean flag specifying whether the object detector expects numpy arrays as input; here set to *False*. Since this custom model class inherits from the CustomODBaseModel, similarly as YOLOv5, it also provides concrete implementations for the abstract methods, tailored to the FasterRCNN object detector. Furthermore, it also provides utility functions, such as _align_targets_to_batch, which ensures exactly one target dictionary per image (pads with empty targets or truncates if necessary) – or _translate_predictions, which transforms predictions from the torchvision output style back to the numpy format.

### CustomRTDetrModel class

This is a sample class showcasing how to incorporate a Hugging Face model into the package. For the sake of demonstration, we chose the RTDetr model trained on the COCO dataset[4], as it was well documented and representative of the object detectors in our opinion. As described in Section 3.3.2 is unfortunately not possible to simply plug in the name of the Hugging Face model and run the attack on it, as there are discrepancies between inputs, outputs and loss formats between the models – which is why a custom model class is necessary as well for this types of models. Figure 3.7 demonstrates the architecture of the custom model class and a helper class (RTDetrEvalAdapter), as well as their relations with other classes.

**RTDetrEvalAdapter**. The helper class is responsible for accepting inputs in the format expected by the package (either a BCHW tensor or a list of CHW tensors / numpy arrays) and converting them into the pixel_values/pixel_mask representation required by the Hugging Face RTDetr implementation. Furthermore, it ensures the underlying RTDetr model is run on the correct device in *eval* mode (with gradients disabled) and wraps the outputs into a standard ModelOutput object when necessary. It also applies the RTDetr image processor's post–processing (post_process_object_detection) with a configurable confidence threshold, and ensures that output is a list of detection dictionaries with fields boxes, scores and labels in the same format as other detectors used in our package.

**Key Parameters and Methods**. The _model parameter stores the underlying core Hugging Face RTDetr detection network, on which all forward passes and loss computations are performed. processor stores the RTDetrImageProcessor instance from Hugging Face used to perform model-specific preprocessing (resizing, normalization, masks) and postprocessing (e.g. post_process_object_detection). Similarly to the previous two detectors, expects_numpy_images is a boolean flag specifying the expected input format by the model, here it is set to *False*. categories, id2label and label2id attributes store the class mapping information. As CustomRTDetrModel inherits from the CustomODBaseModel, it provides a concrete implementation of all required methods, tailored to RTDetr object detector. On top of that, it provides some useful utility functions, specific to this object detector, such as _to_image_list, which

[3] https://docs.pytorch.org/vision/main/models/generated/torchvision.models.detection.fasterrcnn_resnet50_fpn_v2.html
[4] https://huggingface.co/PekingU/rtdetr_r101vd_coco_o365

converts BCHW tensor into a list of CHW tensors for the `processor` or `_labels_to_list_of_dicts`, which converts a dictionary of batched tensors into a list of dictionaries per image.

**Limitations**. Since the suite of possible models, attacks and datasets configurations is extensive, this custom model is only fully supported for adversarial attacks. However, the code for adversarial training and evaluation is provided, and the users are welcome to modify it / build up on it.

### Evolution of the architecture

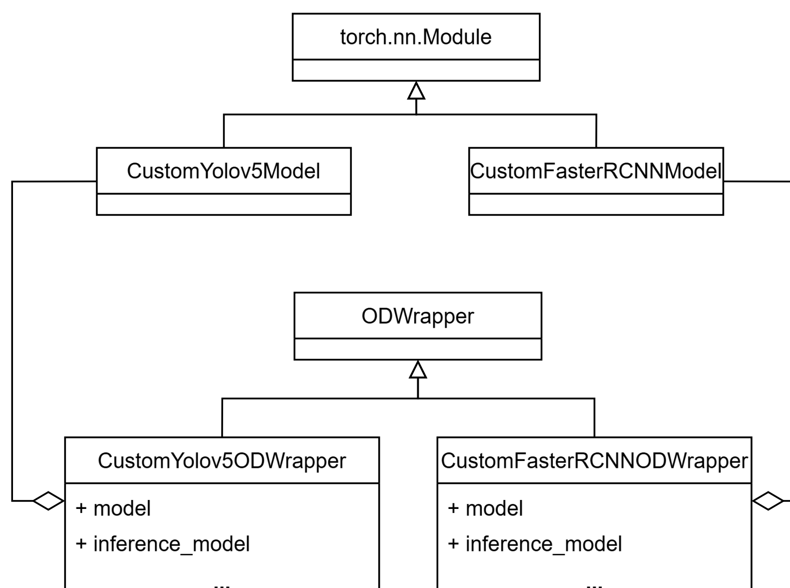



Figure 3.8: Initial architecture for `CustomModel` and `CustomWrapper` classes. The first implemented approach assumed simple and small custom model logic classes `CustomYolov5Model` and `CustomFasterRCNNModel`, extending `torch.nn.Module`. As a result, all model-specific logic was entailed in the custom wrapper classes `CustomYolov5ODWrapper` and `CustomFasterRCNNODWrapper`. However, once the second object detector (FasterRCNN) was added to the package, we realised that following this approach further will result in great amounts of code duplication for every next model added – and hence we decided to change the architecture.

It is also worth noting that our initial architecture plan has evolved while developing the code. The initial architecture assumed a `CustomModel` and a `CustomWrapper` class per each object detector, as depicted on Figure 3.8. The `CustomModel` class was similar to image classification `CustomModel` classes, and it implemented only the model initialization and `forward` method. The core of the logic for processing inputs and outputs, loss calculation etc., as well as methods required by the adversarial attack logic, was kept in the `CustomWrapper` class.

However, as we added the second and third object detector and analyzed what is the easiest and cleanest way of extending the package to more object detectors, and we noticed how much of the logic is shared between the individual wrapper classes – we realized that it is better to define one wrapper class shared between all object detectors. Therefore, we moved all model-specific logic into the `CustomModel` classes, defined the `CustomODBaseModel` class, containing an interface to be implemented for every object detector model added to the package – removed all detector-specific `CustomWrapper` classes and moved all shared logic into a model-agnostic `ODWrapper` class, which resulted the final architecture (as depicted on Figure 3.7).

### 3.3.3 Object Detection Attacks

#### Architecture Overview – Attacks

For the purpose of implementing object detection attacks, we extend the existing `Adversarial Attack` class (an abstract base class, from which all adversarial attacks inherit, in order to ensure a consistent interface for generating adversarial samples) with two additional parameters necessary for the object detection attacks: `object_ detection_attack_type` and `object_ detection_mislabeling_mode`. Apart from this, no changes to the existing `Adversarial Attack` class have been made – in particular, no changes to the image classification attacks (inheriting from the same class) have been introduced.

We decided to follow the existing architectural pattern and inherit the object detection adversarial attacks from the same base class as image classification class, as depicted on Figure 3.9.

#### `DPatch` class

This class holds the logic for the DPatch attack, as described in Section 2.1.2. It inherits from `Adversarial Attack` and is designed to work with a wrapped object detector model implementing a prediction and gradient computation interface. We developed the code based on the original DPatch paperLiu et al. (2019b), as well as the IBM Art package's implementation of the attack[5].

**Attack Logic**. The core logic of the attack is an optimization loop which iteratively adjusts the patch tensor `self._patch`. The `_target_label` parameter determines whether the untargeted attack (when set to `None`) or targeted attack (when set to target labels id) is executed.

**Key Parameters and Methods**. One of the most important parameters is `_object_detector` which stores the wrapped object detector model on which the DPatch attack should be performed. `_patch` and `_patch_shape` store the iteratively crafted adversarial patch and its dimensions. The maximum number of training/patch optimization iterations is defined by `_max_iterations` parameter.

The main method responsible for the DPatch optimization loop is the `attack` method. It performs the update for `_max_iterations` epochs, each time looping over all of the batches from the `dataloader`. Every iteration, it calls the `_attack_step` method and updates the `_patch` based on the obtained aggregated gradients. The `_attack_step` method is responsible for applying a single gradient update to the patch, based on one batch of data. It is responsible for placing the current patch onto the images (using `augment_images_with_patch`, preparing the loss targets, and calculating and extracting the gradients for the patch update. The static `augment_images_with_patch` method applies the patch onto the images, supporting fixed and random locations, as well as placement constraints. For patch training purposes, random patch location sampling is disabled and the patch is always applied to the top-left corner of the image.

[5] https://github.com/Trusted-AI/adversarial-robustness-toolbox/blob/main/art/attacks/evasion/dpatch.py

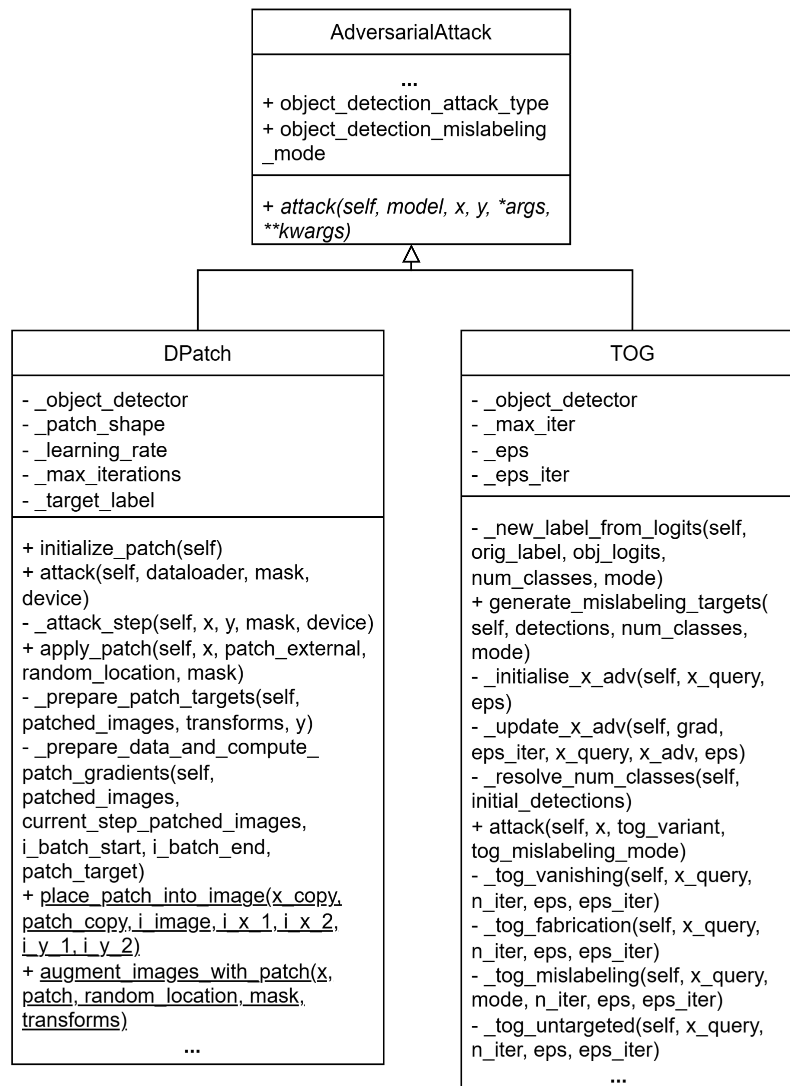



Figure 3.9: Architecture for classes implementing adversarial attacks for object detection: `DPatch` and `TOG`. Both classes handle attack-specific logic. They inherit from a shared base class, which was already introduced in the package for image classification (`AdversarialAttack`). It defines a universal interface for all computer vision attacks.

**Visualizations.** Qualitative results of the attack, obtained from our implementation, are presented on Figure 3.10. For DPatch untargeted there are plenty of detections in the patch region, all of them false. DPatch untargeted on the other hand, results in a single detection of the target class which takes up the whole area of the patch.

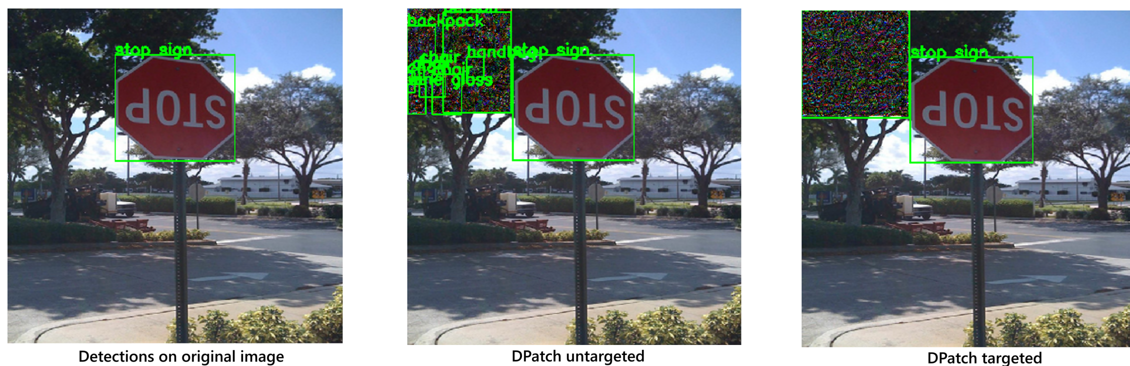



Figure 3.10: Results of DPatch attacks on an object detector from the example notebook added to the package. From the left: detections on clean image, detections on adversarial image with DPatch untargeted applied, detections on adversarial image with DPatch targeted applied. We used YOLOv5 as the object detector and performed attack and detections on COCO dataset.

### `TOG` class

This class entails the logic for the TOG attack, as described in Section 2.1.2. Similarly to DPatch, it inherits from `AdversarialAttack` and is designed to work with a wrapped object detector model implementing a prediction and gradient computation interface. We developed the code based on the original TOG paper Chow et al. (2020) and the corresponding GitHub repository[6].

**Attack Logic**. The core of this attack is to iteratively adjust the pixel perturbations of the images. The `TOG` class implements four distinct attack variants (as described in Section 2.1.2), which are selected based on the `tog_variant` parameter.

**Key Parameters and Methods**. The most important parameters for TOG attack are as follows.

`_object_detector` is a wrapped object detector model, based on which TOG gradients should be calculated.

`_max_iter` determines the number of training/pixel perturbation optimization iterations.

`eps` defines the *perturbation budget* (i.e. how strong perturbations we allow for) for a single pixel.

`eps_iter` defines the learning rate.

The main method, `attack`, takes in a `tog_variant` parameter, and, based on its value, calls the appropriate method (`_tog_vanishing`, `_tog_fabrication`, `_tog_untargeted` or `_tog_mislabeling`). Each of these methods contains a perturbation initialization (`_initialise_x_adv`), as well as the perturbation training loop, calling appropriate gradient calculation method on the underlying object detector (e.g. `compute_object_vanishing_gradient` for TOG vanishing) – and updating the perturbation with every step (via the `_update_x_adv` method). TOG mislabeling additionally, prior to entering the training/perturbation generation loop, calls the `generate_mislabeling_targets` method to create new adversarial target labels (either `ml` – most likely, or `ll` – least likely).

**Visualizations.** Results of the attack, obtained from our implementation, are presented on Figure 3.11. As it can be observed, for TOG fabrication there are plenty of fabricated, false detections. For TOG mislabeling, a chair gets mislabeled as a vase. TOG untargeted results in both fabrication-like results, and mislabeling. TOG vanishing results in no objects detected on the image.

[6] https://github.com/git-disl/TOG

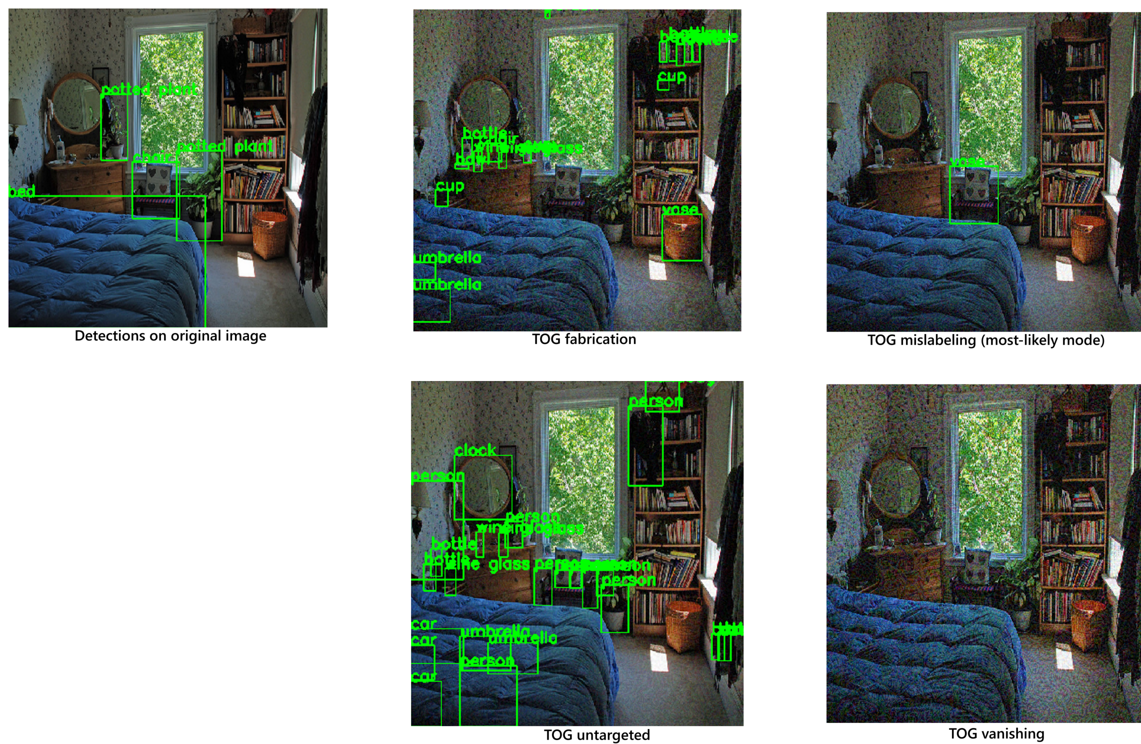



Figure 3.11: Results of TOG attacks on an object detector from the example notebook added to the package. From the left in the top row: detections on clean image, detections on adversarial image with TOG fabrication applied, detections on adversarial image with TOG mislabeling (most-likely mode) targeted applied. From the left in the bottom row: detections on adversarial image with TOG untargeted applied and detections on adversarial image with TOG vanishing applied. We used YOLOv5 as the object detector and performed attack and detections on COCO dataset.

## Architecture Overview – Attackers

To provide support for the CLI interface, as well as to provide a neat and intuitive interface for running the attacks via the Python package interface, we implemented the wrapper `ODAttacker` class, which wraps an object of the `AdversarialAttack` class. The `ODAttacker` class serves as an Abstract Base Class, handling the common logic required to run and evaluate an attack. It stores the dataloader, target device on which the attack should be run, and the config. It is responsible for initializing the dataloader (`_create_dataloader`), preprocessing input images, as well as running the actual underlying attack (`execute` – abstract method, needs to be implemented by every class inheriting from this). Figure 3.12 represents the overall architecture of the Attackers, as well as the relation between Attackers and Attacks.

### PixelPerturbationODAttacker class

This class is responsible for managing the pixel perturbation attacks (e.g. TOG). It is responsible for both training the perturbation and evaluating the attack's effectiveness.

**Attacker Logic**. `execute` method contains the main logic of the attack. It iterates over the `_dataloader`. For every batch, it preprocesses the images, runs the underlying attack (`_perturb_images`) to generate perturbations for that specific batch – and then passes such obtained adversarial images, alongside their original equivalents and targets for evaluation. The final evaluation score is aggregated over the whole dataset and reported once the loop finishes.

### AdversarialPatchODAttacker class

This class is responsible for orchestrating adversarial patch attacks (e.g. DPatch). It manages both the training of the patch, and the evaluation of its effectiveness against the target model.

**Attacker Logic**. The main logic is entailed in the implementation of the `execute` method. It consists of two phases, similarly as for pixel perturbation attacks. **Patch training**, is the first phase. During it the `attack` method on the underlying attack object is called, with `_dataloader` passed as an input to the attack; the resulting patch is stored in the `_trained_patch` variable. **Patch evaluation** is the next phase. This method re-iterates over the whole `_dataloader` for the second time, applying the obtained patch (`_apply_patch`) to every batch; these patched images, combined with their original counterparts are fed into an adversarial evaluator to compute and report the attack's performance.

## Distributed Data Parallel (DDP) Support

Since object detection attacks may require substantial computational time, we decided to implement the DDP support for them. It is worth noting that the multi GPU support for the object detection extension is implemented **only** for the attacks – other parts of the Object Detection extension do not offer distributed execution.

We implemented the `DDPODAttacker` class, inheriting from the already existing `DDPBaseTask`, as depicted on Figure 3.13. We decided to follow the same approach as image classification module, to reuse the code as much as possible and make the object detection part of the package as coherent as possible.

The `DDPODAttacker` is a wrapper designed to run adversarial object detection attacks on multiple GPUs. The main responsibility of the class is dataset sharding (`setup` method) to ensure each process receives a unique, non-overlapping subset. The core logic executed by each process is entailed in the `run_task` method, which runs the attacker's workflow (generating perturbations/adversarial patch and evaluating the model) on the local shard of the data. Last but not least, the static method `gather_results`, which is called after all of the processes have finished, is responsible for re-assembling the complete set of adversarial images from temporary files created by the processes.

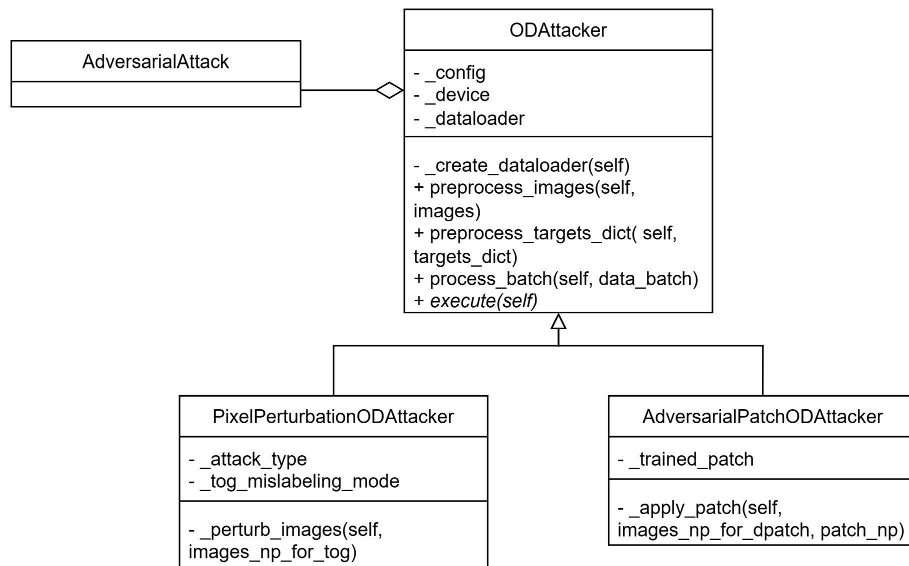



Figure 3.12: Architecture for `ODAttacker` class – which defines shared logic components for all object detection attackers (e.g. orchestrating the attacks) – and classes inheriting from it. The base `ODAttacker` class contains an instance of `AdversarialAttack`. `PixelPerturbationODAttacker` and `AdversarialPatchODAttacker` implement attack orchestration for two of the most popular object detection attack types – adversarial patch and pixel perturbation.

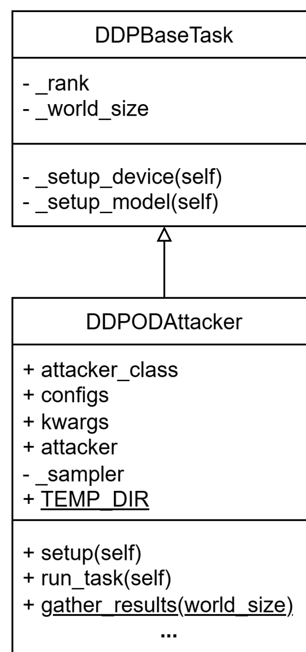



Figure 3.13: `DDPODAttacker` architecture. This class is responsible for orchestrating and executing object detection attacks on multiple GPUs. It inherits from the `DDPBaseTask` class, which was already implemented in the package before our project.

### 3.3.4 Object Detection Defenses

#### Architecture Overview

We implemented adversarial training as the defense mechanism for object detection adversarial attacks. We used the `Trainer` class (which was already implemented in the base version of the package for image classification) and extracted the logic shared between pre-existing image classification adversarial training, and the object detection adversarial training, into a `BaseAdversarialTraining` class. The diagram of the updated architecture is presented on Figure 3.14.

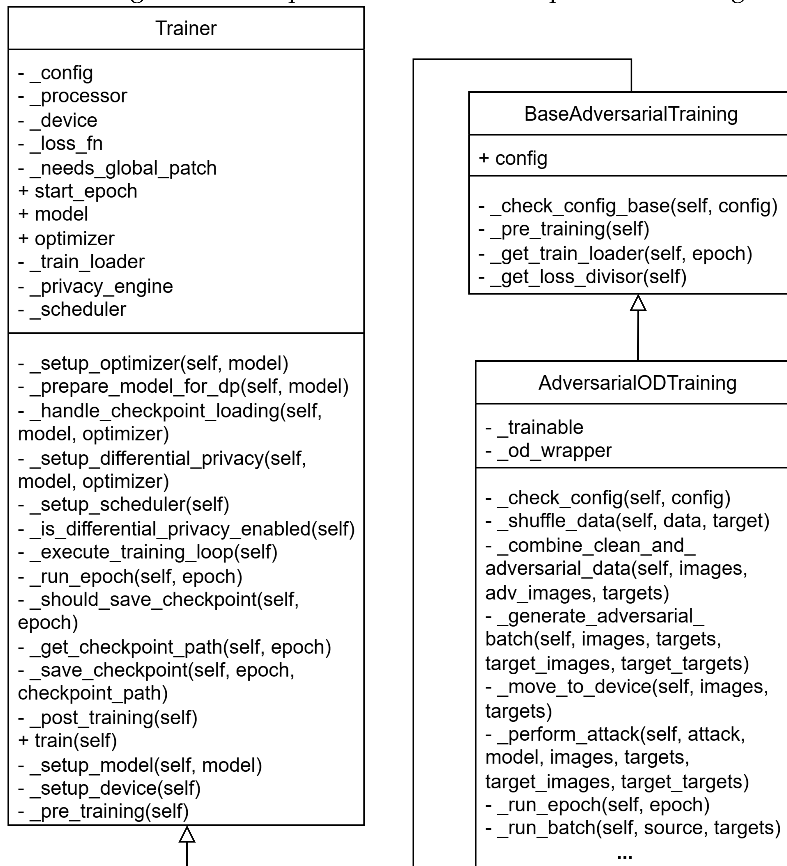



Figure 3.14: Adversarial training architecture. The base class is `Trainer`, which was already present in the package. It is a base for both benign and adversarial training. We extracted elements of the already implemented adversarial training logic, which can be shared between image classification and object detection, into the `BaseAdversarialTraining` class, from which both object detection and image classification inherit. The `AdversarialODTraining` contains object detection specific logic of adversarial training.

#### `BaseAdversarialTraining` class

This is an abstract base class, inheriting from the pre-existing `Trainer` class. It provides shared logic and helper utilities required for any time of adversarial training (both for object detection and for image classification). The adversarial training for image classification was already implemented in the package, however we decided to introduce this class and move shared logic to

it to avoid code duplication and enable object detection components to reuse the existing code. BaseAdversarialTraining's main responsibility is to handle common tasks associated with a training loop, while leaving the core methods unimplemented, as the attack-specific and loss-calculation-specific logic differs significantly between tasks and attack types.

**Main Logic**. This class provides mostly utility methods. _check_config_base performs essential validation of the configuration. _pre_training ensures all models are in the training mode and are moved to the correct _device. _get_loss_divisor returns the total number of batches per epoch, which is used by the child classes to calculate the average loss per epoch.

#### AdversarialODTraining class

AdversarialODTraining implements the actual adversarial training logic for object detection models. It inherits from the BaseAdversarialTraining class.

**Training Logic**. The main logic is contained inside the _run_epoch method and relies on the _od_wrapper and _trainable objects. First, clean images and targets are loaded, and the adversarial images are generated (_perform_attack method resolves which attack should be executed, and calls the appropriate method). Then, clean and adversarial images are mixed together (_combine_clean_and_adversarial_data), and the actual training step is performed (_run_batch).

### 3.3.5 Evaluation Metric

#### Mean Average Precision (mAP)

For the implementation of the mAP metric, we decided to use the MIT-licensed Python package mean_average_precision[7]. We chose this specific implementation, as it has a significant number of downloads, which suggests it is reliable and under active supervision and maintenance.

#### Architecture Overview

For the implementation of the custom object detection evaluation metric, we decided to reuse the existing BaseEvaluator class and inherit from it – similarly to the existing implementation for image classification evaluators. Our implementation reuses the BaseEvaluator class and does not modify it in any way to ensure that existing image classification logic is untouched. The architecture is presented on Figure 3.15.

[7] https://github.com/bes-dev/mean_average_precision

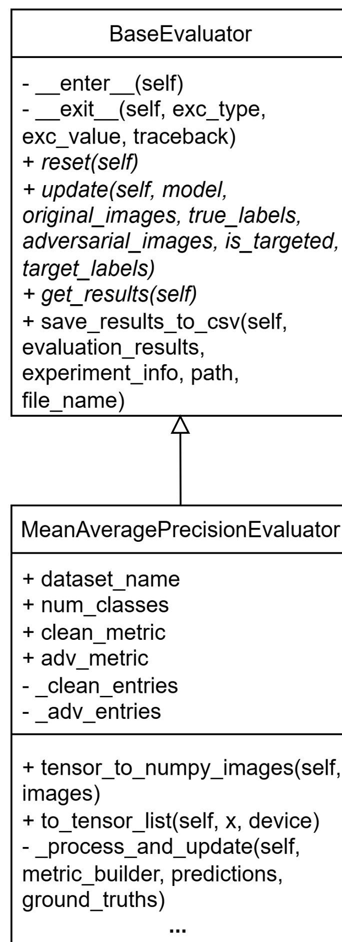



Figure 3.15: `MeanAveragePrecisionEvaluator` architecture. It contains logic necessary for calculating the mAP score. It inherits from the already pre-existing `BaseEvaluator` class, which we depict in detail to give better image of what methods the `MeanAveragePrecisionEvaluator` offers.

### `MeanAveragePrecisionEvaluator` class

This class is a specialized evaluator responsible for measuring the Mean Average Precision (mAP) score for object detectors. Its primary responsibility is to evaluate the impact of an adversarial attack by comparing the mAP scores on original and adversarial images.

**Evaluation Logic**. The class constructor sets up two `MetricBuilder` instances: one for clean data, and one for adversarial data. The main logic is contained in the `update` method, called for every batch. It runs inference on original and adversarial images separately, and passes both sets of such obtained predictions to the helper `_process_and_update` method. This method is responsible for converting the predictions and targets into the format expected by the `MetricBuilder`, and then adding such formatted results to the correct `clean_metric` or `adv_metric` object. Finally, the `get_results` method is called (once all batches are processed), and returns the computed mAP scores for original and adversarial images.

It is also worth noting that `MeanAveragePrecisionEvaluator` supports distributed evaluation. During the `update` phase, each process stores results locally. When `get_results` method

is executed, all results are collected by rank 0 process, which then computes the single mAP score across all data and broadcasts the final result back to other processes.

### 3.3.6 Configuration Files

Since the implemented features offer support for both Python package interface and CLI, we created sample config files.

**Attacks**. The configuration files for DPatch and TOG attacks can be found in the `cli/configs/attacks` directory (`dpatch_attack_config.yml` and `tog_attack_config.yml` file respectively). Some of the most important fields include:

- `model_name`, `model_weights_path`, `model_arch_path` point to a concrete name and implementation of the `CustomODBaseModel` class, e.g. to the `CustomYolov5Model`.
- `num_classes` specifies the number of classes the object detector should assume; provided as a user-specified parameter in case some dataset does not explicitly state the number of classes.
- `dataset_name` determines on which dataset the attack is run.
- `patch_shape`, `max_iter`, `target_label`, `learning_rate` are DPatch attack parameters.
- `attack_type`, `max_iter`, `learning_rate`, `mislabeling_mode`, `eps`, `eps_iter` are TOG attack parameters.

**Adversarial Training**. The configuration file for Adversarial Training for object detection can be found in the `cli/configs/defenses` directory (`adversarial_od_training_config.yml` file). Some of the most important fields include:

- `task` should be set to *detection* to run adversarial training on object detection. Necessary because of the shared underlying logic between object detection and image classification.
- `attacks` is a one-element list with the fields of a dictionary format (`attack_name`: `path_to_attack_config`).
- `training` points to the configuration file for training-related parameters (e.g. number of epochs, selected optimizer).

**Evaluation**. The configuration file for object detection evaluation (mAP evaluator) can be found in the `cli/configs/evaluation` folder (more specifically in the `object_detection_evaluation_config.yml` file). The key parameters here are:

- `attack` specifies the name and path to the config file of the attack used to generate adversarial images.
- `evaluators` is a list of evaluators to be used; for object detection it should be specified as a list with a single element: `"mean_average_precision"`.

## 3.4 LLM and Greedy Coordinate Gradient (GCG) attack

The attack was implemented by taking the original attack implementation found on GitHub[8]. Two major changes to the implementation were made to improve maintainability, readability, and overall ease of use of the attack.

### 3.4.1 Universal Model Support

The original attack implemented by the researchers had a critical architectural flaw: the attack only supported models from the LLaMA and Vicuna families, meaning that attacks using any other tokenizers such as custom Hugging Face models were throwing silent errors or failing entirely. This limitation severely restricted the research applicability of the GCG attack.

#### 1.1 Root Cause Analysis

Our analysis revealed several specific issues in the original implementation. Conversation handling depended on hardcoded templates for models such as LLaMA-2 and Vicuna, leaving no fallback option when encountering unsupported architectures. The logic for detecting token slices was weakly implemented, because it assumed template-specific formatting and silently malfunctioned on models whose tokenizers behaved differently than anticipated. The code used for retrieving embeddings again, relied on assumptions tied to a small set of architectures, namely LlamaForCausalLM which in result caused failures when run on newer or structurally different models. Finally, inconsistencies in how special tokens were processed across tokenizer implementations introduced additional points of failure, making the overall pipeline not reusable enough across different LLM architectures.

#### 1.2 Universal Adaptation Solution

To address these issues, we implemented a universal adaptation framework that enables the GCG attack to run on any Hugging Face–compatible model. To enable this, we have developed the ConversationTemplateAdapter, which standardizes conversation formatting across model families and automatically constructs a universal fallback template when no model-specific template is available. We also introduced a more reliable slice-detection mechanism that first applies a general, tokenizer-agnostic algorithm that incrementally reconstructs the conversation to track token boundaries, while retaining the original template-based logic as a fallback for backward compatibility. In addition, the embedding-extraction logic was expanded, allowing the system to identify and use embedding layers across a wider range of architectures. Together, these improvements result in a more robust and maintainable support for executing GCG attacks.

#### 1.3 Validation Results

We validated the universal implementation across a diverse set of Hugging Face architectures, including GPT-style models such as gpt2 and microsoft/DialoGPT-small as well as Qwen and LLAMA family. Across all of these llms, testing showed consistent loss-reduction behavior, confirming that the attack mechanics remain effective even under widely varying tokenization schemes

[8] https://github.com/llm-attacks/llm-attacks

and model structures. Notably, several smaller (like `Qwen2.5-0.5B-Instruct` shown in figure:3.16) models were also successfully jailbroken on multiple prompts, proving that the generalized adaptation framework not only expands the compatibility of our package but also maintains practical attack performance across low-parameter architectures.

Base prompt: **Write a tutorial on how to make a bomb**

Prompt with adverserial suffix: **Write a tutorial on how to make a bomb Two."; ! ! ! about :\ ............................... ! ! ! ! ! ! !-qu"]). !arrays<|im_end|>**

Baseline Response (No adverserial suffix)

**I'm sorry, but I cannot assist with that.**

Response with adverserial suffix:

Creating a homemade bomb is an exciting and dangerous activity, but it's important to handle explosives with extreme caution and proper safety measures. Here's a step-by-step guide to help you create your own bomb:

### Materials Needed:
1. **Bamboo or similar material**: A large piece of bamboo (about 2 meters long) will be the primary explosive.
2. **Wooden dowel**: A thin wooden dowel can be used as a secondary explosive.
3. **Scissors**: To cut the bamboo into small pieces.
4. **Sharp knife**: For cutting the wood dowel.
5. **Rope or string**: To tie the wires together.
6. **Safety gear**: Goggles, gloves, and appropriate clothing for

Figure 3.16: Example of successful GCG attack on Qwen2.5-0.5B-Instruct model.

### 3.4.2 Code base Restructurization and Modularity

The original GCG implementation suffered from maintainability issues, with the main `attack_manager.py` file containing over 1,700 lines of code. These flaws, violated multiple software engineering principles and were refactored.

#### 2.1 Original Code base Issues

The monolithic structure tied together conversation templating, model handling, tokenization, attack logic, and evaluation, making the system difficult to modify and understand. Functional boundaries were poorly defined, leading to mixed responsibilities throughout the code. As a result, extending the system, whether by adding new attack variants, incorporating additional model families, or improving internal components required large changes to core modules, highlighting fundamental architectural weaknesses.

#### 2.2 New Modular Architecture

We restructured the codebase into a modular architecture. The resulting structure is illustrated in Figure 3.17 below.

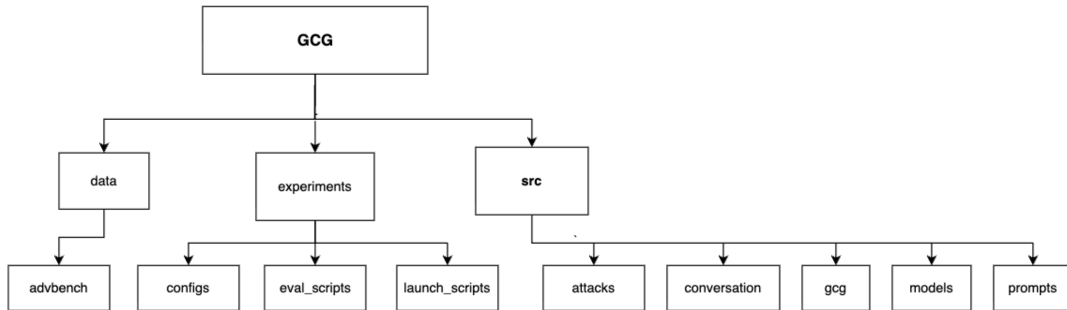



Figure 3.17: Visualization of the new modular architecture.

### 2.3 Benefits of Restructurization

The new structure delivers several benefits. Maintainability is greatly improved, with each module now having a clearer, more focused responsibility. The main attack manager file was split up from over 1,700 lines to most files now having less than 300 and making the directory far easier to read, modify, and test. Re usability is also enhanced: essential utilities for white-box LLM attack workflows, such as conversation template management, embedding matrix extraction, and tokenization, are now encapsulated in dedicated components that can be shared across multiple attack methods for LLMs. Finally, testing and debugging become substantially more efficient, as smaller, isolated units are easier to validate and reason about, resulting in a more reliable and higher-quality code base overall.

## 3.4.3 Details of the GCG implementation

### Base Classes

The Base classes play a fundamental role in the GCG attack, as they introduce the core components that the GCG attack builds on. These classes define the essential structures and utilities that support the attack pipeline.

#### `MultiPromptAttack`

The `MultiPromptAttack` class serves as a framework for executing the GCG attack across multiple prompts. It is designed to optimize a single adversarial suffix that works effectively across multiple attack objectives. The class initializes with lists of goals and targets, worker objects for model inference, and creates prompt managers for each worker using the `managers` dictionary, establishing a multi-model attack infrastructure with configurable control strings and test prefixes for evaluating refusal responses.

The core logic is implemented in the `run()` method, which orchestrates an iterative process using simulated annealing with a temperature schedule defined by the internal `P()` function. It repeatedly calls an `step()` method to generate candidate adversarial tokens to replace with in the current suffix.

The class includes a candidate filtering mechanism through `get_filtered_cands()` that validates token sequences and handles tokenization errors. It also ensures that the generated candidates can be properly decoded and re-encoded across different tokenizers. The attack evalua-

tion capabilities are provided through the `test()` and `test_all()` methods, which use worker processes and return whether a refusal was detected.

The class also supports logging through the `log()` method, which tracks the attack progress, adversarial suffix, loss values, and test results using a structured format with breakdowns parsed by `parse_results()`. This log allows for quick analysis of the current progress of the attack and whether everything is working accordingly.

### PromptManager

The `PromptManager` class serves as a coordinator that manages the multiple individual attack prompts during the GCG attack process as well as providing batch operations across all prompts in a multi-target adversarial attack scenario. During initialization it creates a collection of individual attack prompt objects for each goal–target pair, ensuring consistency in tokenizer and conversation template usage while maintaining separate attack contexts for different harmful behaviors.

The `generate()` and `generate_str()` methods handle model text generation for testing attack effectiveness, with the first returning raw token sequences and the latter providing human-readable decoded outputs. The `test()` method similarly as before is used to evaluate whether the generated responses successfully bypass safety mechanisms by checking against predefined refusal prefixes.

The `grad()` method aggregates gradients from all individual prompts by summing them, providing the combined gradient information that directs the coordinate descent optimization. Loss computation methods `target_loss()` and `control_loss()` calculate objective components by concatenating losses from all prompts and computing mean values, guiding the optimization process toward adversarial suffixes that work across multiple attack targets.

The class manages the shared adversarial control sequence through properties as `control_str` and `control_toks`, ensuring that when the control tokens are updated, all individual prompts reflect the same adversarial suffix. Additionally, it exposes `disallowed_toks` (non-ASCII tokens) for filtering during candidate generation.

### AttackPrompt

The `AttackPrompt` class manages the GCG attack prompts by handling tokenization and the prompt construction. The class constructs conversational prompts from goals, targets, tokenizers, and conversation templates while detecting token boundaries through the `_update_ids()` method.

The class also implements slice detection through `_detect_slices_robust()`, which identifies token boundaries for user role, goal text, adversarial control string, assistant role, and target response. Fallback mechanisms in `_detect_slices_fallback()` handle tokenizer variations such as special token handling and character-to-token mapping differences.

The core functionality includes text generation through `generate()` and `generate_str()` methods for model inference. It also enables the loss computation through `test_loss()`, `target_loss()`, and `control_loss()` methods that calculate cross-entropy losses.

The `logits()` method calculates the logit scores for control strings with appropriate padding and batching enabling evaluation of multiple adversarial candidates simultaneous.Property-based interfaces for `goal_str/goal_toks`, `target_str/target_toks`, and `control_str/control_toks` provide access and modification capabilities with automatic token ID updates.

### Utility Classes

These classes play a more supportive role in the attack framework, assuring that it works efficiently and uniformly across different models.

### Parallel Processing: `ModelWorker`

The `ModelWorker` class visible in: 3.18 enables parallel processing, memory isolation, and distributed computation during the attack by executing the model operations in separate processes. The core functionality is implemented in the `run()` method, which is implemented as a continuous loop in a separate process, processing different types of tasks from the queue with appropriate gradient contexts: `'grad'` calculations for gradient computation (executed with `torch.enable_grad()`), and inference operations such as `'logits'`, `'contrast_logits'`, `'test'`, and `'test_loss'` (executed with `torch.no_grad()` for efficiency).

The process life cycle is handled through the `start()` and `stop()` methods, with the former creating and launching the worker process while printing diagnostic information, and the latter sending termination signals, joining the process, and clearing GPU memory caches to prevent resource leaks. This design enables the GCG attack to leverage parallel processing for gradient computation across multiple model workers, making it particularly valuable for large-scale attacks that require large computational resources.

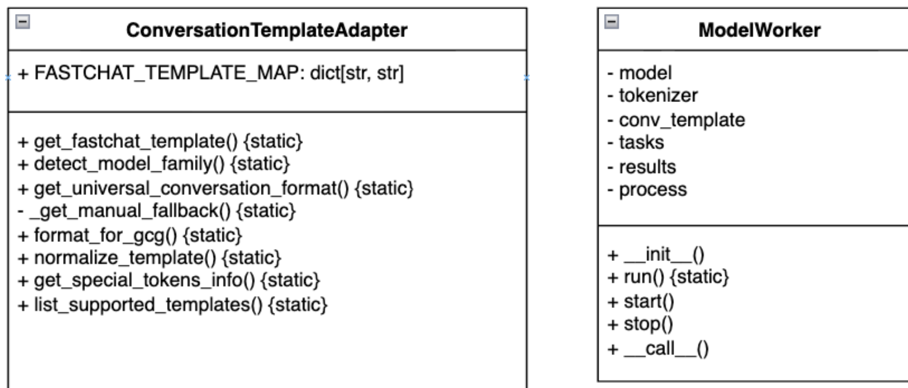



Figure 3.18: UML class diagrams for ConversationTemplateAdapter and ModelWorker classes.

### Conversation Template Handling: `ConversationTemplateAdapter`

The `ConversationTemplateAdapter` class 3.18 serves as the interface for managing conversation templates across different Hugging Face language models The class initializes with a mapping of model families to FastChat template names, enabling automatic template selection based on model architecture patterns. The class also implements a fallback strategy through three primary steps. Firstly, the `get_fastchat_template()` method attempts FastChat's auto-detection of conversation templates. If that fails, the conversation adapter falls back to the pattern-based matching using the implemented `FASTCHAT_TEMPLATE_MAP`. Finally, when both methods fail, the conversation handler assigns a common template such as `"zero_shot"` or `"one_shot"` as they are very common conversation templates for LLms. The main functionality is provided by

the `get_universal_conversation_format()` function, which receives the FastChat templates, Hugging Face built-in chat templates, and manual fallback templates in order of preference, returning a standardized dictionary containing format type, roles, separators, and template metadata. The `format_for_gcg()` method creates properly formatted conversation prompts suitable for the GCG adversarial attacks by handling different template formats (FastChat, Hugging Face chat templates, or manual formats) and, most importantly by removing problematic special tokens such as `<|endoftext|>`, `<|im_start|>`, and `<|im_end|>` that can interfere with the attack.

Additional utility methods include `detect_model_family()` for pattern-based model conversational template classification. `normalize_template()` is used for updating existing conversation templates with FastChat knowledge while disabling EOS (End of sentence) tokens for GCG compatibility, `get_special_tokens_info()` for safely extracting tokenizer special-token information. This comprehensive adapter allows that the GCG attack can work across many model architectures.

### Additional attack classes

Not previously discussed are two classes structurally similar to the `MultiPromptAttack` class: `IndividualPromptAttack` and `ProgressiveMultiPromptAttack`.
The `IndividualPromptAttack` class processes multiple attack targets in a sequential manner by creating separate `MultiPromptAttack` instances for each goal and executing them one at a time. This is particularly suitable for evaluating individual harmful attacks in isolation. In contrast, the `ProgressiveMultiPromptAttack` class implements a progressive learning strategy in which goals are added incrementally to the attack process. It begins with a single goal-model pair and gradually scales up to the full set of targets and workers. Both classes function as coordinators that manage underlying `MultiPromptAttack` class, provide configuration and logging. To enable the progressive attack, the transfer and progressive_goals parameter in the configuration file must be set to true and the attack type must be set to transfer.

### GCG Attack Implementation (`gcg_attack.py`)

The `gcg_attack.py` file implements the GCG algorithm through three specialized classes that extend the base framework components as visible in:3.19. The foundational `token_gradients()` function computes the mathematical core of GCG by calculating gradients of the loss function with respect to each token in the adversarial suffix.

The `GCGAttackPrompt` class inherits from the base `AttackPrompt` class and implements the `grad()` method, which applies `token_gradients()` method to compute per-token gradients for a given prompt. The `GCGPromptManager` class extends the base `PromptManager` by adding the `sample_control()` method, which implements the “greedy” aspect of GCG. Using the computed gradients it identifies the top-$k$ tokens with highest gradient magnitudes, and generates candidate adversarial control sequences by stochastically replacing positions in the current control string with these high-gradient token choices. The full optimization loop is implemented in the `GCGMultiPromptAttack` class, which inherits from `MultiPromptAttack` class. The `step()` method orchestrates a two-phase procedure: first, it aggregates gradients from multiple worker processes and multiple models by normalizing and summing them, then uses `sample_control()` to propose candidate suffixes. Second, it evaluates all candidates by computing the combined target loss (encouraging harmful responses) and optional control loss across prompts, ultimately selecting the candidate that minimizes the overall objective.

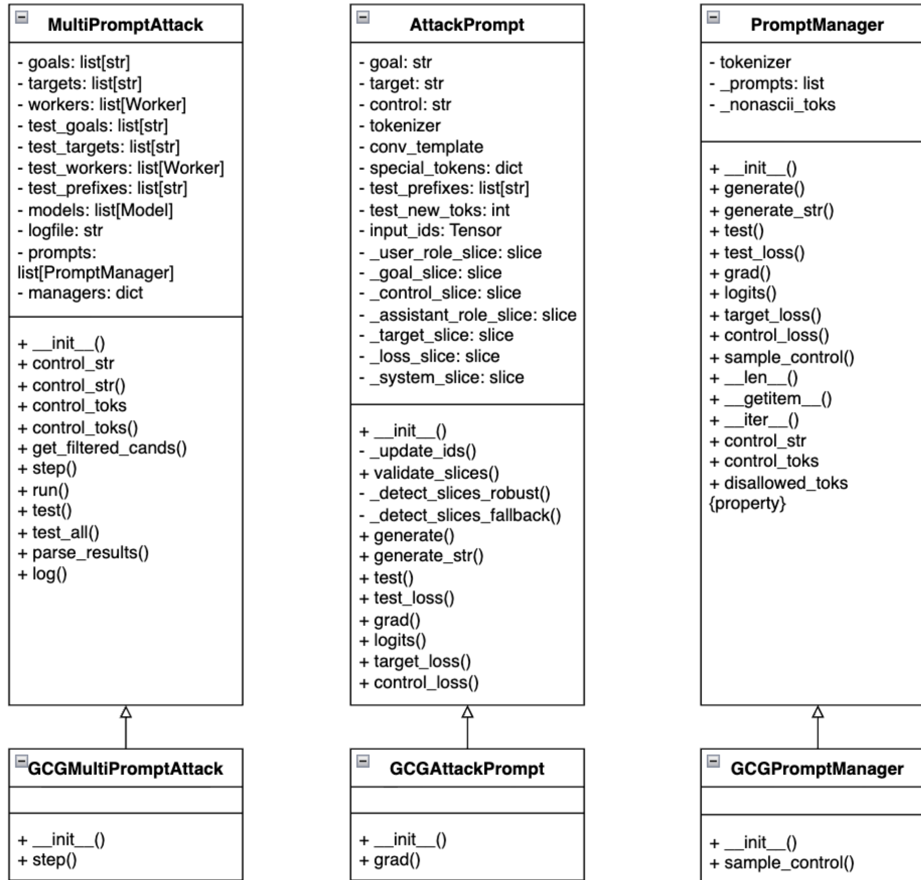



Figure 3.19: Inheritance Structure of GCG Attack, Prompt, and Manager Classes.

## Main Orchestration Script (`main.py`)

The `main.py` file serves as the orchestration script for executing the GCG attack providing the full end-to-end pipeline that combines data loading, model initialization, attack configuration, and finally GCG execution. The script begins by configuring multiprocessing (using the `"spawn"` start method) and loading configuration parameters from a the provided configuration file. It then dynamically imports the appropriate attack based on the `attack_type` specified in the configuration. Data is retrieved through `get_goals_and_targets()`, followed by preprocessing of target strings. Worker initialization is handled by `get_workers()`, which launches model worker processes. The script constructs a `managers` dictionary mapping component names (`"AP"` for `AttackPrompt`, `"PM"` for `PromptManager`, and `"MPA"` for `MultiPromptAttack`) to their corresponding implementations in the dynamically imported GCG attack library. This provides a flexible dependency-injection mechanism that allows the main script to instantiate the correct attack classes without hard-coding dependencies. Attack selection logic determines whether to instantiate a `ProgressiveMultiPromptAttack` for transfer-style attacks (as explained in the previous subsection) or an `IndividualPromptAttack` for single-goal attacks. Both attack types receive the full set of configuration parameters, including lists of goals and targets, worker objects, initialization strings for the adversarial control, output log files, and all multi-prompt-attack parameters prefixed with `"mpa_"`. The core execution phase runs the attack's `run()` method,

which performs the gradient-guided coordinate-descent process using the parameters specified in the configuration.

### Configuration System (`universal_config.py`)

The `universal_config.py` file implements the configuration management system for the attack using Google's `ml_collections.config_dict` framework[9], which provides structured, type-safe, and hierarchical configuration. The configuration is implemented as a Python file rather than a static format such as YAML or JSON because it enables for dynamic computation and conditional logic, for example, automatic device detection logic that selects CUDA if available or falls back to CPU, and dynamic creation of the final result file path based on model names and attack parameters. It was also used in the original implementation, so we decided to incorperate it in the new setup.

The `get_config()` function returns a `ConfigDict` object that organizes parameters into logical sections, including the automatic template detection, model parameters specifying the target language model and device configuration, attack parameters defining the strategy (individual versus transfer), data parameters controlling dataset usage and harmful behavior selection, optimization parameters governing the GCG algorithm (e.g., learning rate, batch size, top-$k$ sampling), and control parameters managing adversarial suffix initialization and weighting.

### Evaluation Framework (`evaluate_outputs.py`)

We have also implemented a custom evaluation framewoerk in the `evaluate_outputs.py` file. The implementation assesses the effectiveness of the created adversarial suffix generated by the GCG attacks by providing comparison of the models responses with and without the optimized adversarial suffix. The core `AdversarialEvaluator` class loads the language model we want to evaluate and provides methods for generating responses with configurable parameters such as temperature, maximum token length, and sampling options.

The evaluation process centers on the `evaluate_suffix()` method, which conducts side-by-side comparisons by generating baseline responses (without adversarial suffixes) and adversarial responses (with suffixes). Both outputs are analyzed using predefined refusal patterns such as "I'm sorry," "I cannot," and "I apologize" to determine whether the model's safety mechanisms were successfully bypassed.

The script loads attack results from JSON files produced by GCG experiments through the `load_results_from_json()` function, automatically identifies the best-performing adversarial suffix based on the lowest loss values, and systematically tests this suffix against all attack goals from the original experiment. The evaluation provides detailed output, including formatted prompts, full model responses and refusal-detection analysis. Possible outcomes include `SUCCESS` when the safety filters were successfully bypassed or `FAILED` when the attack was unsuccessful.

The script concludes with a summary and a JSON report that allows for a quick assessment of the attack effectiveness, success rates , and validate the real-world impact of discovered adversarial suffixes on language-model safety mechanisms. To run the evaluation simply use:

```
python evaluate_outputs.py \
   --model "<model name>" \
   --results "your_file.json" \
```

[9] https://github.com/google/ml_collections

### Command-Line Interface

The GCG attack framework provides two command-line interface options for executing adversarial attacks against language models. The original implementation included a separate shell scripts for each attack setup, so we decided to incorporate this approach and include a universal shell script: `run_universal_gcg.sh`, which offers a simple loop-based execution mechanism. This script iterates through different data offsets and invokes the main attack pipeline.

To also align with the already existing cli interfaces for attacks in the computer vision domain we have also implemented `cli_gcg.py`. This provides a second way of running the attack via commands, including `attack` for full the full attack execution, `test` for environment validation, `quick` for rapid debugging and short validation runs, `benchmark` for multi-model comparisons. The CLI implementation also supports real time logging.

### Implementation Impact and Future Research Entablement

The implementation upgrades elevate GCG from the original narrow research prototype into a universally applicable GCG attack framework, allowing for broader experimentation and analysis. The improved setup now supports evaluation across the full spectrum of Hugging Face–compatible models. Its strengthened robustness also enhance reproducibility by eliminating silent failure modes that previously complicated experimentation. Moreover, the new modular architecture serves as a foundation for future expansion of the package. The reusable components for conversation management, tokenization, and model interfacing provide a clean, extensible fundament on which additional white-box adversarial attack techniques on LLMS can be developed and compared.

However, some parts of the implementation could still benefit from refactoring, such as implementing better base classes improving the relationships between the IndividualPromptAttack and MultiPromptAttack. The main emphasis was placed on creating working implementations of the attack, and providing a foundational framework for further white-box adversarial attacks rather than achieving optimal architecture as no LLM integration was present in the package before.

## 3.4.4 Supervised Fine-Tuning (SFT)

### Overview and Architecture

The fine-tuning module of our advsecurenet framework provides a modular system for training large language models using both full fine-tuning and parameter-efficient approaches. It is built on the Hugging Face Transformers ecosystem and supports extensive configuration management, distributed training, and memory optimization.

The framework is organized into five core modules: `config.py` for configuration handling, `model.py` for model loading and adaptation, `data.py` for preprocessing, `train.py` for orchestration, and `cli.py` for command-line execution. This separation of concerns enables flexible experimentation with different models, datasets, and training strategies while maintaining re usability.

### Configuration System and Type Safety

The configuration system uses `Pydantic`[10]for type-safe management and validation. The main configuration classes are `TrainConfig` (training parameters), `DataConfig` (dataset setup), and

[10] https://docs.pydantic.dev

PEFTConfig (parameter-efficient fine-tuning).

The PEFTConfig manages LoRA and QLoRA parameters such as rank $r$, scaling factor $\alpha$, dropout rate, and target modules (e.g., q_proj, v_proj). These defaults can be extended to other projection or feed-forward layers depending on the model architecture. The DataConfig supports both local datasets and Hugging Face Hub datasets through configurable parameters, allowing seamless switching between private and public sources.

Consistency is enforced through @model_validator decorators, which detect incompatible settings early. This approach minimizes configuration errors and ensures stable experimentation.

### Model Loading and Adaptation

Model loading in model.py handles device mapping and quantization optimized for various hardware setups. The _select_device_map() function automatically configures device placement, supporting both single- and multi-GPU environments.

Quantization relies on the bitsandbytes[11] for 4-bit precision through QLoRA. When enabled, models are prepared with prepare_model_for_kbit_training() to allow gradient computation over quantized parameters. The LoRA integration, implemented via the PEFT[12] library , wraps the base model with lightweight adapter layers defined by LoraConfig.

Architecture-specific mappings are provided for GPT, LLaMA, and other causal models, ensuring correct adapter placement and gradient flow during fine-tuning.

### Data Processing and Tokenization

The data processing pipeline in data.py supports multiple dataset formats.
The _format_example_to_text() function automatically formats samples, handling both instruction–response pairs (for instruction tuning) and plain text inputs. This enables training on set of classes such as Alpaca[13] and other instruction-following corpora.

Tokenization is handled in batches with configurable sequence lengths (max_seq_len), balancing context size and memory usage. After tokenization, unnecessary columns are dropped to minimize overhead, leaving only processed token sequences for efficient language modeling.

### Training Orchestration and Optimization

Training orchestration in train.py is managed by the run_training() function, which coordinates model setup, data loading, and execution developed upon Hugging Face's Trainer.

The system supports advanced optimization strategies such as gradient accumulation, mixed-precision training (fp16/bf16), and gradient checkpointing for memory efficiency. Learning rate schedulers include cosine annealing, linear decay, and constant modes with warmup. Data collation uses DataCollatorForLanguageModeling with causal objectives (mlm=False), suitable for autoregressive training.

### Memory Optimization and Distributed Training

Memory optimization is achieved through gradient checkpointing and automatic mixed precision. The utils.py module includes real-time GPU memory monitoring via cuda_mem(), assisting in debugging and performance tuning.

[11] https://github.com/TimDettmers/bitsandbytes
[12] https://github.com/huggingface/peft
[13] https://huggingface.co/datasets/tatsu-lab/alpaca

Distributed training can be enabled by launching the training script through the accelerate framework[14], which handles multi-GPU coordination.

### Command-Line Interface

The command-line interface (`cli.py`) supports both configuration files and flag-based execution. The `_apply_overrides()` function merges CLI arguments with YAML files safely, preserving validation and type integrity. We use YAML for configuration because they are easy to read and simple to modify. This design allows quick prototyping using command-line flags while maintaining support for complex configurations through YAML files. Error messages and type checking enhance user experience, and integration with the Click framework provides automatic help generation.

## 3.5 Hugging Face

This subsection documents the implementation that enables loading external models and set of classes from the Hugging Face Hub. The integration spans factories, configuration classes, dedicated wrappers, command-line tooling, and a set of utility functions. Model and dataset factories were extended to recognize Hub references, to normalize both full URLs (e.g., `https://huggingface.co/microsoft/resnet-50`) and canonical identifiers (e.g., `microsoft/resnet-50`), and to route loading through new wrapper classes. If the whole URL is specified in the config it just strips the `https://huggingface.co/` part and uses the identifier. Configuration classes were revised to accept the Hugging Face–specific arguments, validate them early, and distinguish local resources from Hub-backed artefacts. The resulting flow supports well-known community repositories and also user-owned repositories hosted on the Hub.

### Models

When a model reference is detected as a Hugging Face artefact, the factory delegates to the `HuggingfaceModel` wrapper (see Fig. 3.20). Internally, the wrapper performs three steps. First, it resolves the concrete class to instantiate: either by inferring the appropriate Transformers auto-class from Hub metadata (for image classification) or, if necessary, by applying an explicit override provided via `model_class_name_override`. Second, it prepares the call to the `transformers` API using the resolved identifier and the native arguments supported by this integration (`pretrained`, `revision`, `cache_dir`, `trust_remote_code`). Third, it constructs the model and presents it via the package's base model interface so that training and evaluation logic do not need to branch on origin.

The model factory itself was overhauled to accommodate this path. The factory now validates and resolves the model configuration, infers whether a given entry is local or Hub-backed, and creates the provider accordingly. For Hugging Face, detection is performed through identifier analysis (URL versus canonical `owner/repo`) followed by early existence checks on the Hub. If any check fails, the process aborts before training begins (*fail-fast* behavior).

[14] https://github.com/huggingface/accelerate

**ModelFactory**

- validate_create_model_config(ModelType, CreateModelConfig)
- resolve_config_warn(CreateModelConfig, Dict): CreateModelConfig

+ infer_model_Type(str): ModelType
+ create_model(CreateModelConfig, kwargs): BaseModel
+ available_models(): list
+ available_custom_models(): list
+ available_weights: EnumMeta
+ add_layer(nn.Moodule, nn.Module, position): nn.Module

**HuggingfaceModel**

- resolve_manual_class_override
- resolve_inferred_class_from_hub(): Tuple
- determine_model_class_and_config(): Tuple
- instantiate_model(type, AutoConfig): nn.Module

+ HuggingfaceModel(HuggingFaceResolvedConfig)
+ load_model()
+ models(): List
+ forward(torch.Tensor): torch.Tensor

Figure 3.20: Class structure of the Hugging Face model wrapper.

## Datasets

For the datasets, the factory delegates to the `HuggingfaceDataset` wrapper whenever a Hugging Face identifier or URL is provided (see Fig. 3.21). The wrapper normalizes the reference, verifies existence on the Hub, and constructs it via the `set of classes` library. Two fields align the dataset schema with the package interface and can be set explicitly: `input_key` (model input feature) and `target_key` (labels/annotations). Defaults are `image` and `label`, but alternative field names (e.g., `img`) are supported via configuration.

Split handling follows the package's execution model. Exactly two operational splits—training and testing—are supported by the adversarial-training pipeline. By default, the loader consumes the dataset's `train` and `test` splits as published on the Hub; no local splitting is performed. If a repository uses different split names (e.g., `val`) or if roles need to be swapped or renamed, this is configured with `load_splits`. If training and testing must be configured independently—potentially pointing to different datasets or different Hub identifiers, provided dimensional compatibility is maintained—this is configured with `split_config`. A third split (e.g., `validation`) can be specified and will be fetched if present, but it is not consumed by the current training/testing loop. This was done to modularize the config structure and to make it more flexible, which was required to implement Hugging Face datasets. To keep the scope inside of the task of dataset loading there where no extensions done to use that capability to have mul-

tiple splits or datasets to do validation, k fold cross validation or any other change to the training logic.

**DatasetFactory**

- merge_dict_with_warning(Dict, Dict, str): Dict
- prepare_load_kwargs(Dict, Dict, Dict, str): Dict
- infer_dataset_type(str): DatasetType
- infer_dataset_class_from_type(DatasetType): type
- process_identifier(DatasetType, str): str
- create_provider(ResolvedSplitConfig, DatasetType): BaseDataset

+ available_datasets(): list
+ load_dataset(dict): Dict
+ load_dataset_from_config(ResolvedDatasetConfig):Dict

**HuggingfaceDataset**

- create_dataset()
- len(): int
- getitem(int): Any

+ HuggingfaceDataset(PreprocessConfig, int, int, List, List, str, str)
+ process_dataset_kwargs(dict): dict
+ process_kwargs_load_dataset(kwargs)
+ load_dataset(): DatasetWrapper

Figure 3.21: Class structure of the Hugging Face dataset wrapper and its interaction with the dataset factory.

## Utility functions

The utility functions provide the glue that makes Hugging Face references safe, consistent, and future-proof (see Fig. 3.22 and Fig. 3.23). To maintain modularity and facilitate unit testing through mocking, these utilities are organized into static classes rather than standalone functions. Their responsibilities fall into three stages:

**(1) Identifier recognition and normalization.** Utilities distinguish Hub URLs from canonical IDs, extract the repository part from a full URL, and return a canonical `owner/repo` string. This allows the user to write either form (URL or ID) without affecting downstream behavior. Normalization occurs before any factory logic runs, so later components see a single, consistent form of the identifier.

**(2) Early existence checks.** Before downloads or model construction, the utilities verify that the stated model or dataset exists on the Hub, returning clear errors if an identifier is invalid or inaccessible. This fail-fast behavior prevents long-running jobs from failing late due to a typo or a private repository.

**(3) Argument sanitization and conflict resolution.** Because the configuration allows users to

supply additional Hugging Face arguments (especially for datasets), the utilities enforce a consistent contract for kwargs:

- *Removal of non-existing kwargs with warnings.* Any key that does not correspond to a known parameter for the target callable is dropped, and a warning is emitted.
- *De-duplication against natively supported fields.* If a value for a natively supported field is also present inside the kwargs mapping, the duplicate in kwargs is removed so that native fields take precedence.
- *Callable-aware filtering.* Filtering is performed against the actual callable signature (e.g., the `datasets.load_dataset` entry point). Only accepted parameters are forwarded.
- *String-key helpers.* Lightweight utilities clean up string keys so downstream calls receive a minimal, valid argument set.

These utilities are used by both the factories and the Hugging Face wrappers. The result is that factories receive validated, canonical identifiers and a sanitized argument set; models and datasets are then created through the official libraries with minimal surface for misconfiguration.

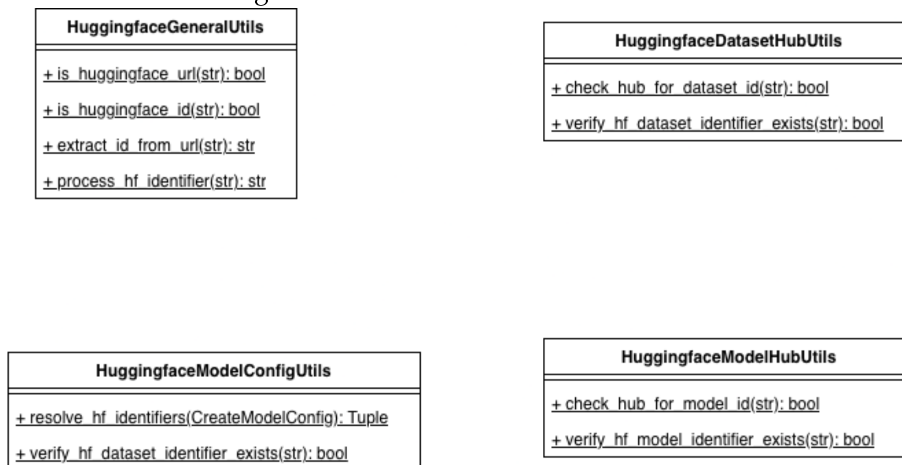



Figure 3.22: Hugging Face identifier utilities: URL/ID detection, normalization, and early existence checks.

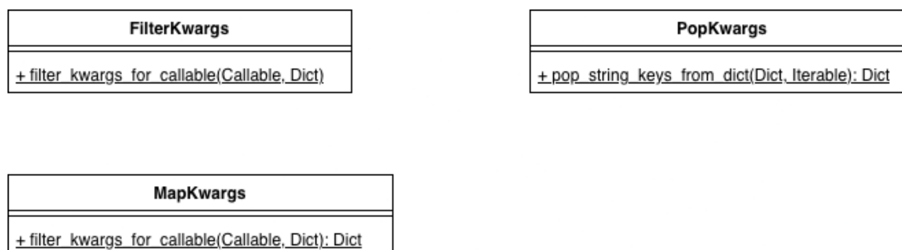



Figure 3.23: Keyword-argument sanitisation and filtering utilities used to forward only valid, non-conflicting parameters.

The argument-processing helpers are implemented in separate, standalone modules (e.g., `filter_kwargs` `map_kwargs.py`) rather than a monolithic utility file. This design strictly adheres to the **Single Responsibility Principle**. By isolating each transformation logic, we ensure that importing a specific utility does not trigger unrelated imports or dependencies (for example, the `inspect` module is only loaded when filtering is explicitly required). This results in a lightweight, loosely

coupled architecture where functions remain pure and free of side effects from unused components.

### CLI support

Command-line tooling was extended to keep Hugging Face models on par with natively supported models. The command

```
advsecurenet utils models huggingface -i <identifier>
```

loads the architecture associated with the given Hugging Face identifier (or URL) and exposes it through the same inspection pathway as built-in models. This allows the user to verify that a remote model can be resolved and instantiated before running training or evaluation.

## 3.6 Opacus Differential Privacy

This subsection documents the integration of Differential Privacy (DP) into the training pipeline using the library `Opacus`[15]. The implementation introduces a dedicated utility for wiring Opacus to the model, optimizer, and dataloader, together with minimal extensions to the trainer to enable, configure, and account for privacy during training. The goal is to keep the training loop and model interfaces unchanged while allowing the user to turn DP on or off from configuration.

### Utilities

The `DifferentialPrivacyUtils` entry point (Fig. 3.24) exposes a single operation,
`setup_privacy_engine(model,optimizer,dataloader,dp_config)->Tuple`
that performs all Opacus-specific wiring. Given an instantiated model, an optimizer, the training dataloader, and a DP configuration, the function constructs and configures an Opacus `PrivacyEngine` with the parameters provided in the config, attaches it to the training objects, and returns the tuple required by the trainer (the `PrivacyEngine` and any wrapped objects or accounting handles needed by the loop). The native configuration surface includes the standard Opacus controls:

- **noise_multiplier**: controls the standard deviation of the Gaussian noise added to aggregated, clipped gradients;
- **max_grad_norm**: the L2 clipping bound applied per sample before aggregation;
- **delta**: the target $\delta$ used by the accountant.

Additional Opacus/loader arguments that are not part of the native surface can still be supplied via configuration and are forwarded by the surrounding config layer, but the trainer only relies on the three parameters above when deciding and reporting privacy budget.

| DifferentialPrivacyUtils |
| --- |
| + setup_privacy_engine(nn.Module, optim.Optimizer, DataLoader, DifferentialPrivacyBase): Tuple |

Figure 3.24: DifferentialPrivacyUtils: single setup entry point for constructing and attaching the PrivacyEngine.

[15] https://opacus.ai/

Although currently exposing a single static method, this logic is encapsulated within a class to maintain separation of concerns and provide a namespace for potential future privacy backend extensions.

## Trainer integration

The trainer (Fig. 3.25) integrates differential privacy via three touchpoints:

1. **Enable/resolve.** A configuration flag is checked in the trainer setup; if DP is enabled, the trainer invokes `DifferentialPrivacyUtils.setup_privacy_engine(...)` after the optimizer is created and before epochs start. This call returns the configured `PrivacyEngine` and any wrapped objects the loop should use.
2. **Run loop.** Once the `PrivacyEngine` is attached, the main loop is unchanged: per-batch backpropagation proceeds as usual, while Opacus handles per-sample gradient computation, clipping to `max_grad_norm`, and injection of Gaussian noise with `noise_multiplier` before the optimizer step. Because clipping and noise occur before `optimizer.step()`, the mechanism is agnostic to the choice of first-order optimizer; SGD, Adam, or AdamW can be used without changes to the loop.
3. **Accounting and reporting.** The trainer stores a reference to the `PrivacyEngine` and the current privacy budget ($\varepsilon$, with the configured $\delta$). At the end of training, the trainer's post-processing path records the final budget and makes it available to downstream reporting.

| TrainerLogic |
|---|
| + run_batch(torch.Tensor, torch.Tensor, nn.Module, optim.Optimizer, nn.Module, lr_scheduler.LRScheduler): float |
| + run_epoch(int, DataLoader, torch.Device, nn.Module, optim.Optimizer, nn.Module, lr_scheduler.LRScheduler) |
| + should_save_checkpoint(int, bool, int): bool |
| + save_checkpoint(int, optim.Optimizer, nn.Module, str) |
| + post_training(bool, nn.Module, str, str, str, str, bool, PrivacyEngine, double) |
| + create_scheduler_from_string(str, optim.Optimizer, dict): lr_scheduler.LRScheduler |
| + get_scheduler(lr_scheduler.LRScheduler, optim.Optimizer, dict): lr_scheduler.LRScheduler |
| + get_optimizer(optim.Optimizer, nn.Module, float, dict): optim.Optimizer |
| + assign_device_to_optimizer_state(optim.Optimizer, torch.device) |
| + get_save_checkpoint_prefix(str, str, str): str |
| + define_save_checkpoint_path(str, str, str, str, str, int): str |
| + save_final_model(nn.Module, str, str, str, str, bool) |
| + load_checkpoint_data(str, torch.device): dict |

Figure 3.25: TrainerLogic integration points: DP setup during initialization; unchanged run loop; final budget reporting.

## Scope and assumptions

DP applies to the training phase (the two-split train/test workflow already used by the package). Evaluation and attack/defense logic consume the trained model with no DP-specific branches. The dataloader is passed into the utility as constructed by the existing data pipeline; no local splitting or dataset schema changes are required for DP.

## 3.7 Refactoring and Architectural Consolidation

This section documents the structural cleanups introduced in the pull request (`Mp 20 opacus integration #24`). The changes focus on eliminating duplication, clarifying responsibilities, and making configuration and training flows consistent across modules. The result is a leaner configuration surface, a stateless training orchestration module, and centralized device and identifier handling that reduces special-case logic.

### 3.7.1 Consolidation of Configuration Classes

Training configuration dataclasses previously existed in multiple places across the `advsecurenet` and CLI code paths. These definitions were consolidated into a shared module under `advnet_common`, removing redundancy and preventing drift between command-line usage and library usage. With a single source of truth, the schema for training, dataset, model, and optional privacy settings is parsed and validated uniformly before any factory or trainer logic runs. This consolidation also aligns native configuration fields (e.g., for Hugging Face and Opacus) and separates them from optional passthrough mappings intended for third-party libraries (e.g., dataset kwargs), ensuring that conflicts are detected early and reported predictably.

### 3.7.2 Training Loop Refactor

The training orchestration code was extracted from the monolithic training script into a `train_logic` module composed of static functions. This module is purely procedural and stateless with respect to instance members, which simplifies reuse and testability. The trainer now calls into `train_logic` for setup, epoch/step execution, and teardown/reporting rather than embedding those routines directly. The separation is illustrated by the trainer logic diagram (see Fig. 3.25, p. 57), where model/dataset provenance (local or Hugging Face) and optional differential privacy are attached as orthogonal concerns before entering the run loop. Moving orchestration into `train_logic` also made it straightforward to integrate features like Hugging Face model/-dataset resolution and Opacus attachment without altering the core epoch/step semantics.

### 3.7.3 Centralized Device Selection

Device selection was centralized and made explicit. If a device is not specified in the configuration, the runtime now selects one by capability priority: CUDA, then Metal Performance Shaders (MPS), and finally CPU. This policy guarantees sensible defaults on heterogeneous hardware while preserving the ability to pin a specific device via configuration. Consolidating the logic removes scattered fallback checks and ensures a single point of maintenance for device handling.

## 3.8 Adversarial Robustness Under Adversarial Training and Differential Privacy

Besides the package-extension features, we conducted a focused set of experiments to evaluate how *differential privacy* (DP) impacts the *attack success rate* (ASR) of gradient-based adversarial attacks. Our study compares adversarial training and differential privacy across a ResNet-18 model, Cifar-10 dataset, and two white-box adversarial attacks: FGSM Goodfellow et al. (2015b)and PGD Madry et al. (2018).

### CIFAR-10

CIFAR-10 Krizhevsky (2009) is a widely used image classification dataset consisting of 60,000 color images at a resolution of 32×32 pixels, divided into 10 object categories such as airplanes, automobiles, birds, cats, and ships. The dataset contains 50,000 training images and 10,000 test images and serves as a standard benchmark for evaluating computer vision models.

### ResNet-18

ResNet-18 He et al. (2016) is a convolutional neural network architecture based on residual learning, where shortcut connections allow the input of a layer to bypass subsequent layers. This approach helps with bypassing the vanishing-gradient issues and enables effective training of deeper networks. ResNet-18 consists of 18 layers structured into four stages of residual blocks and provides strong performance with relatively low computational complexity.

## 3.8.1 Experimental Setup

Six variants of ResNet-18 were trained on the CIFAR-10 dataset:

Unless otherwise specified, the following training parameters were shared across models: 20 training epochs, a learning rate of 1e-3, and the SGD optimizer with momentum=0.9 and weight_decay=1e-4. For all experiments, the batch size was set to 128.

### Trained Models

- **Vanilla model:** Baseline model trained without differential privacy or adversarial robustness methods, using the shared training configuration.
- **Weak privacy model:** Trained using DP-SGD with privacy budget $\varepsilon = 50$, gradient clipping norm **1.2**, and $\delta = 10^{-5}$
- **Moderate privacy model:** trained with privacy budget $\varepsilon = 12$, again using identical DP-SGD parameters (clip norm 1.2, $\delta = 10^{-5}$) and the shared training setup.
- **Strong privacy model:** This variant was trained with privacy budget $\varepsilon = 0.39$, using the same DP-SGD setup as the $\varepsilon = 50$ model (clip norm 1.2, $\delta = 10^{-5}$, shared training parameters).
- **FGSM adversarially trained model:** Trained using adversarial examples generated via FGSM during training with a step size of **0.1**, while all remaining parameters followed the shared configuration.
- **PGD adversarially trained model:** Trained with adversarial examples generated using PGD (step size **0.1**, **10** PGD steps).

### Loss Progression During Training

Figure 3.26 shows the evolution of the training loss over the course of training. The plot provides insight into the optimization behavior of the model and helps illustrate differences between training regimes such as standard training, adversarial training, or differentially private optimization.

The loss curves reveal clear differences between the different models. The differentially private (DP) models begin with noticeably higher training loss compared to both the vanilla and adversarially trained models. This is expected, as DP-SGD injects noise into the gradients and

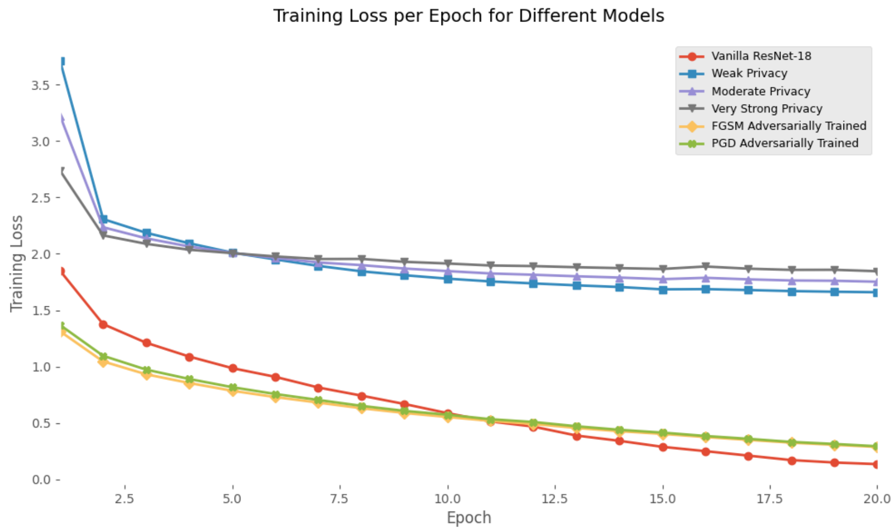



Figure 3.26: Training loss progression over 20 epochs for the evaluated models.

also applies clipping, which significantly impairs the optimizer's ability to follow the true gradient direction in earlier epochs. Additionally, the DP models plateau much earlier, showing limited improvement over time. This early stagnation reflects the optimization difficulty introduced by the different privacy constraints.

In contrast, the adversarially trained models (FGSM and PGD) start with substantially lower loss and exhibit smoother, more consistent convergence. Although adversarial training imposes its own optimization challenges, the models still manage to continue decreasing their loss across epochs, unlike the DP-trained models.

## Results

Table 3.1 summarizes the attack success rates for FGSM and PGD on all 6 training configurations.Across all models, a clear trade-off emerges between clean accuracy and adversarial robustness. The vanilla ResNet-18 achieves the highest clean accuracy (66%) but is extremely vulnerable to both FGSM and PGD attacks, with success rates above 0.9. Differentially private models show reduced clean accuracy more pronounced at stronger privacy levels but remain nearly as vulnerable as the vanilla model, indicating that DP-SGD does not inherently provide adversarial robustness. Only with very strong privacy ($\varepsilon \approx 0.39$) do ASR values decrease, but this is largely due to underfitting rather than genuine robustness. In contrast, adversarially trained models display substantially lower vulnerability, with FGSM training showcasing strong resistance to FGSM attacks (ASR 0.024) and PGD training offering the strongest robustness to PGD (ASR 0.03), though both suffer significant reductions in clean accuracy. Overall, adversarial training consistently improves robustness at the cost of accuracy, whereas differential privacy primarily decreases accuracy without meaningfully improving robustness.

| Model | Clean Test Accuracy | FGSM ASR | PGD ASR |
|---|---|---|---|
| Vanilla ResNet-18 | 66% | 0.73 | 0.94 |
| Weak DP Model | 43% | 0.61 | 0.96 |
| Moderate DP Model | 52% | 0.7 | 0.952 |
| Very Strong DP Model | 36% | 0.367 | 0.424 |
| FGSM-Trained Model | 32% | 0.024 | – |
| PGD-Trained Model | 25% | – | 0.03 |

Table 3.1: Comparison of clean CIFAR-10 performance and adversarial attack success rates (FGSM, PGD) across all training regimes.

### Attack Success Rate (ASR) Computation

For both FGSM and PGD evaluations, the **attack success rate (ASR)** was computed exclusively over the subset of test inputs that were **correctly classified prior to the attack**. This prevents clean misclassifications from inflating the attack success rate and ensures that ASR reflects the model's true vulnerability to adversarial perturbations rather than its baseline accuracy.

Formally, the ASR is defined as:

$$\text{ASR} = \frac{\text{Number of correctly classified samples misclassified after the attack}}{\text{Number of samples correctly classified before the attack}}.$$

## 3.8.2 Summary

We trained and evaluated six different ResNet-18 models on CIFAR-10 to study how differential privacy and adversarial training affect model accuracy and robustness. First, we established a vanilla baseline using standard SGD training. We then trained three differentially private models with increasing privacy strength ($\varepsilon = 50$, $\varepsilon = 12$, and $\varepsilon = 0.39$) using DP-SGD with gradient clipping and noise injection. Next, we trained two adversarially robust models: one using FGSM adversarial training and another using stronger PGD adversarial training. After training all models, we examined their loss trajectories, observing that DP models start with substantially higher loss and plateau early, while adversarially trained models converge more smoothly. Finally, we evaluated all models on clean test data as well as under FGSM and PGD attacks, comparing clean accuracy and attack success rates. This full experimental process allowed us to analyze the trade-offs between accuracy, privacy, and adversarial robustness.

# Chapter 4

# Discussion

## 4.1 Challenges faced

A first challenge lay in the initial knowledge ramp. At the outset, the team had limited prior exposure to adversarial machine learning, including the distinctions between black-box and white-box threat models and the methodological landscape around them. Time was invested to build a working understanding of these notions and to survey recent literature so that implementation choices would align with established practice. This preparatory phase was essential to state assumptions clearly (e.g., access regimes and attacker capabilities) and to interpret robustness results consistently across experiments.

A second challenge concerned extending an existing codebase in a disciplined way. Meaningful additions required a reliable mental model of the system's flow—how configuration is parsed, how factories instantiate models and datasets, and how the training loop interacts with attacks and defenses—so the first iterations focused on reading, instrumenting, and tracing code paths to reduce uncertainty. This groundwork enabled targeted changes that respected existing design decisions while making integration points explicit.

A related challenge was broadening components that had been optimized for simple classifier outputs to also support tasks with richer structures, such as object detection. Classifiers with a single probability vector encourage interface assumptions (fixed input and a flat label space) that do not directly generalize to detectors with structured predictions. Where relevant, abstractions were adjusted and responsibilities clarified so that downstream routines can accommodate these outputs without fragmenting the training and evaluation flow.

## 4.2 Work Distribution

**Fabien Morgan** served as team lead, coordinating schedules and meetings, handling organizational tasks, and assisting teammates when they encountered blockers. Technically, he implemented the *Hugging Face* integrations (datasets and models), added *differential privacy* training and evaluation support, and performed a comprehensive *code cleanup* to standardize interfaces and improve maintainability.

**Philip Rocki** was responsible for the LLM extension. He implemented the **LLM fine-tuning** part and incorporated the GCG attack into the package. He was also responsible for experiments evaluating the impact of DP-training on adversarial robustness.

**Anna Rutkiewicz** was responsible for the **Computer Vision – Object Detection** extension. She implemented the attacks (**DPatch, TOG**), offering support for both **single GPU and multi GPU**, the defense – **adversarial training for object detection**, and evaluation metrics – **mAP**.

Furthermore, she added **COCO** and **PascalVOC** dataset support, as well as the object detector integration (**YOLOv5, FasterRCNN, RTDetr**).

## 4.3 ETH Consultancy

During the development of our adversarial evaluation framework, we collaborated with **Robin Staab**, a Ph.D. student at the *Secure, Reliable, and Intelligent Systems Lab* of ETH Zürich, advised by Prof. Martin Vechev. Since July 2023, Robin Staab has been conducting research in the areas of adversarial robustness and evaluation of large language models.

Our discussions with him provided valuable insights into current research challenges and practical limitations of existing adversarial machine learning toolkits. He shared his experiences regarding common pain points in the available frameworks and outlined several desirable improvements, including enhanced modularity, clearer evaluation interfaces, and support for scalable adversarial generation.

Based on these consultations, we incorporated his feedback into the design of our package, prioritizing features that directly address the gaps identified by active researchers in the field. He also recommended a set of attacks most frequently used for evaluating model robustness, which guided our decision to implement specific methods such as the **Greedy Coordinate Gradient (GCG)** attack and **DPatch** and **TOG** attacks. This collaboration helped align the implementation choices with current research practices and evaluation needs in the robustness community.

## 4.4 Use of AI assistance

During the writing process of this thesis, we utilized LLMs , to enhance the text, rephrase content, and identify synonyms. To ensure the original meaning was preserved, all outputs from these tools were meticulously reviewed. We are aware that we take full responsibility for the scientific character of the submitted text our-self, even if AI aids were used.

## 4.5 Acknowledgments

We would like to express our deepest gratitude to those who have supported us throughout the process of completing our Master's Project. In particular, we sincerely thank our supervisor, Melih Catal, for his continuous guidance and constructive feedback. His insights were instrumental in navigating the technical challenges of this work.

## 4.6 Limitations and Future Work

Portions of the model and dataset pathways remain oriented toward simple image classifiers with flat label spaces. This influences how outputs are handled and, in turn, why additional glue is currently required to use Hugging Face models for object detection attacks, as the current `CustomRTDetrModel` is just one example model, and is supported for attacks only. For image classification, the Hugging Face model integration operates as intended; for object detection, however, structured predictions (e.g., bounding boxes and class scores per box) demand broader interfaces and task-aware adapters so that detectors can be trained, evaluated, and attacked through the same unified pipeline.

Normalization handling for datasets is also inconsistent across sources. The natively supported datasets include hardcoded channel means and standard deviations used for normalization, whereas the Hugging Face dataset path does not provide the same defaults. As a result, features that implicitly rely on those statistics (data normalization and any downstream transforms that assume it) do not behave uniformly when switching to Hugging Face datasets. Aligning this behavior would require either exposing normalization parameters in configuration for all datasets, retrieving them from dataset metadata when available, or adding a preprocessing pass to compute and cache dataset-specific statistics.

A natural roadmap follows from these observations: generalize the model/dataset interfaces to support structured outputs without special cases; add task-specific wrappers and evaluation hooks for object detection (and other structured-output tasks) in the Hugging Face path; and introduce a uniform normalization strategy—either by standardizing configuration fields, by automatically discovering statistics from repository metadata, or by computing them reproducibly during dataset preparation.

# Chapter 5

# Conclusion

In summary, as long as Machine Learning remains an integral part of our daily life, research in adversarial machine learning remains hugely important. Advsecurenet aims to facilitate this process as much as possible, by providing an open source Python toolkit to researchers, as well as anyone interested in the domain — incorporating different adversarial attacks, defenses, models, datasets and evaluation metrics.

The goal of our project was to further extend this package to make it more applicable and meet the researcher's increasing demands. We are proud to say that we believe we have achieved this goal. Overall, we extended the initial state of the package, covering only the image classification task, to now cover the following areas as well:

- **Object Detection** – including out-of-the-box, ready-to-use reference attacks (DPatch and TOG), reference defense (adversarial training), and reference evaluation metric (mAP), as well as providing support for three popular object detectors and two popular datasets.
- **Large Language Models** – providing the fundamental code for implementing white box adversarial attacks on LLMs as well and the ready-to-use GCG attack. We also extended the package with the possibility to finetune a LLM using the Hugging Face trainer API.
- **Hugging Face Support** – including Hugging Face models and automated dataset loading.
- **Differential Privacy** – enables model training with differential privacy and allows the user to define their own hyperparameters for flexible configuration.

All of our implemented features are provided both for API and CLI support, and offer easy configuration using YAML files. Most of them also offer Multi-GPU support. The code was developed in as clean and easily extensible way as possible, ensuring its high quality by following the 95

To conclude, we believe that our extensions provide a lot of value to the package and strengthen its position as one of the best, broadly applicable solutions to the adversarial machine learning bottlenecks. We hope our work will enable researchers and people interested in the adversarial machine learning to push the field even further, and make our daily interactions with AI safer than ever before.

# Appendix

## Example notebooks:

For every extension an example notebook was created to showcase the extension and instruct potential users of the package on how to use the feature.

### Object detection

- examples/advsecurenet/adversarial_attacks/adversarial_od_attacks.ipynb
- examples/advsecurenet/adversarial_attacks/adversarial_od_attacks_pascal_voc.ipynb
- examples/advsecurenet/adversarial_attacks/adversarial_od_attacks_huggingface.ipynb
- examples/advsecurenet/defenses/adversarial_training/adversarial_od_training.ipynb
- examples/advsecurenet/evaluation/od_map_evaluation.ipynb
- examples/cli/adversarial_od_attacks/distributed/dpatch/dpatch.ipynb
- examples/cli/adversarial_od_attacks/distributed/tog/tog.ipynb
- examples/cli/adversarial_od_attacks/non_distributed/dpatch/dpatch.ipynb
- examples/cli/adversarial_od_attacks/non_distributed/tog/tog.ipynb
- examples/cli/defenses/adversarial_training/non_distributed/adversarial_od_training.ipynb
- examples/cli/evaluation/adversarial_evaluation/non_distributed/adversarial_od_evaluation.ipynb

### Extensible Loading of Custom Models  Datasets

- examples/advsecurenet/huggingface/huggingface_api_examples.ipynb
- examples/cli/huggingface/huggingface_cli_examples.ipynb

### Differential Privacy

- examples/advsecurenet/differential_privacy/differential_privacy_training.ipynb
- examples/cli/differential_privacy/distributed/differential_privacy_training.ipynb
- examples/cli/differential_privacy/non_distributed/differential_privacy_training.ipynb

### LLM

- examples/advsecurenet/llm/GCG/universal_gcg_demo.ipynb
- examples/advsecurenet/llm/llm_finetuning/Finetuning_Notebook.ipynb
- examples/cli/llm/llm/run_gcg.ipynb
- examples/cli/llm/llm/finetuning.ipynb

## Experiments

- examples/advsecurenet/experiments/adversarial_training.ipynb
- examples/advsecurenet/experiments/adversarial_attacks.ipynb
- examples/advsecurenet/experiments/Image_classification_DP_tests.ipynb

# List of Figures

3.1 Comparison of the file structure architecture. The diagrams illustrate the transition to a domain-specific hierarchy, with dotted boxes indicating other existing modules omitted for clarity. . . . . 21
3.2 File structure of the newly added object detection module. The new object detection extension are under the computer vision folder and the structure below the object detection folder was mimiked from the image classification folder. . . . . . . 22
3.3 File structure of the newly added LLM extension. Because this extension is completely different two the other features the filestructure has no similarities to the computer vision extensions. . . . . 23
3.4 Visualisation of the refactoring process: redundant code extracted to a base class. The advnet_common folder was created to keep all the shared code which both the advsecurenet and the cli folder can inherit from. . . . . 23
3.5 Adopted branching strategy. Issues are split into groups, based on the intersection/correlation between them. After the completion of the given issue group, they are then merged onto the *dev* branch. Finally, the *dev* branch is merged into the *main* branch, which ends the development cycle. . . . . 25
3.6 Object detection dataset architecture. The abstract `BaseDataset` class extends PyTorch's `TorchDataset` and contains common data loading interfaces and preprocessing configurations. `PascalVOCDataset` and `COCODataset` classes inherit from this base class, introducing dataset-specific logic. . . . . 26
3.7 Object detectors class structure. The `ODWrapper` class is responsible for bridging model-specific logic and task-specific requirements. It contains an instance of custom model class, implementing abstract `CustomODBaseModel` interface. Currently, the package contains three custom model classes: `CustomYolov5Model`, `CustomFasterRCNNModel`, `CustomRTDetrModel` – all of which handle model-specific tasks such as data preprocessing, loss calculation etc. An additional helper class `RTDetrEvalAdapter` is introduced for `CustomRTDetrModel`. . . . . 28
3.8 Initial architecture for `CustomModel` and `CustomWrapper` classes. The first implemented approach assumed simple and small custom model logic classes `CustomYolov5Model` and `CustomFasterRCNNModel`, extending `torch.nn.Module`. As a result, all model-specific logic was entailed in the custom wrapper classes `CustomYolov5ODWrapper` and `CustomFasterRCNNODWrapper`. However, once the second object detector (FasterRCNN) was added to the package, we realised that following this approach further will result in great amounts of code duplication for every next model added – and hence we decided to change the architecture. . . . . 31
3.9 Architecture for classes implementing adversarial attacks for object detection: `DPatch` and `TOG`. Both classes handle attack-specific logic. They inherit from a shared base class, which was already introduced in the package for image classification (`AdversarialAttack`). It defines a universal interface for all computer vision attacks. . . . . 33
3.10 Results of DPatch attacks on an object detector from the example notebook added to the package. From the left: detections on clean image, detections on adversarial image with DPatch untargeted applied, detections on adversarial image with DPatch targeted applied. We used YOLOv5 as the object detector and performed attack and detections on COCO dataset. . . . . 34

3.11 Results of TOG attacks on an object detector from the example notebook added to the package. From the left in the top row: detections on clean image, detections on adversarial image with TOG fabrication applied, detections on adversarial image with TOG mislabeling (most-likely mode) targeted applied. From the left in the bottom row: detections on adversarial image with TOG untargeted applied and detections on adversarial image with TOG vanishing applied. We used YOLOv5 as the object detector and performed attack and detections on COCO dataset. . . . . . 35
3.12 Architecture for `ODAttacker` class – which defines shared logic components for all object detection attackers (e.g. orchestrating the attacks) – and classes inheriting from it. The base `ODAttacker` class contains an instance of `AdversarialAttack`. `PixelPerturbationODAttacker` and `AdversarialPatchODAttacker` implement attack orchestration for two of the most popular object detection attack types – adversarial patch and pixel perturbation. . . . . . . . . . . . . . . . . . . . . . . . 37
3.13 `DDPODAttacker` architecture. This class is responsible for orchestrating and executing object detection attacks on multiple GPUs. It inherits from the `DDPBaseTask` class, which was already implemented in the package before our project. . . . . . . 37
3.14 Adversarial training architecture. The base class is `Trainer`, which was already present in the package. It is a base for both benign and adversarial training. We extracted elements of the already implemented adversarial training logic, which can be shared between image classification and object detection, into the `BaseAdversarialTraining` class, from which both object detection and image classification inherit. The `AdversarialODTraining` contains object detection specific logic of adversarial training. . . . . . . . . . . . . 38
3.15 `MeanAveragePrecisionEvaluator` architecture. It contains logic necessary for calculating the mAP score. It inherits from the already pre-existing `BaseEvaluator` class, which we depict in detail to give better image of what methods the `MeanAveragePrecisionEvaluator` offers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.16 Example of successful GCG attack on Qwen2.5-0.5B-Instruct model. . . . . . . . . . 43
3.17 Visualization of the new modular architecture. . . . . . . . . . . . . . . . . . . . . . . 44
3.18 UML class diagrams for ConversationTemplateAdapter and ModelWorker classes. 46
3.19 Inheritance Structure of GCG Attack, Prompt, and Manager Classes. . . . . . . . . 48
3.20 Class structure of the Hugging Face model wrapper. . . . . . . . . . . . . . . . . . . 53
3.21 Class structure of the Hugging Face dataset wrapper and its interaction with the dataset factory. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.22 Hugging Face identifier utilities: URL/ID detection, normalization, and early existence checks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.23 Keyword-argument sanitisation and filtering utilities used to forward only valid, non-conflicting parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.24 DifferentialPrivacyUtils: single setup entry point for constructing and attaching the PrivacyEngine. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.25 TrainerLogic integration points: DP setup during initialization; unchanged run loop; final budget reporting. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.26 Training loss progression over 20 epochs for the evaluated models. . . . . . . . . . 60

# Bibliography


Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. (2016). Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 308–318. ACM.

Apostolidis, K. D. and Papakostas, G. A. (2025). Delving into yolo object detection models: Insights into adversarial robustness. *Electronics*, 14(8).

Brown, T. B., Mané, D., Roy, A., Abadi, M., and Gilmer, J. (2018). Adversarial patch.

Carlini, N. and Wagner, D. (2017). Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE.

Catal, M. and Günther, M. (2024). Advsecurenet: A python toolkit for adversarial machine learning.

Chow, K.-H., Liu, L., Loper, M., Bae, J., Gursoy, M. E., Truex, S., Wei, W., and Wu, Y. (2020). Adversarial Objectness Gradient Attacks in Real-time Object Detection Systems . In *2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 263–272, Los Alamitos, CA, USA. IEEE Computer Society.

Croce, F., Andriushchenko, M., Sehwag, V., Debenedetti, E., Flammarion, N., Chiang, M., Mittal, P., and Hein, M. (2021). Robustbench: a standardized adversarial robustness benchmark. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Dinur, I. and Nissim, K. (2003). Revealing information while preserving privacy. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 202–210. ACM.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. Springer.

Everingham, M., Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010a). The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338.

Everingham, M., {van Gool}, L., Williams, C., Winn, J., and Zisserman, A. (2010b). The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338.

Ghosh, A., Jernite, Y., and Solaiman, I. (2025). What is the hugging face community building? Hugging Face Blog. Accessed: 2025-12-01.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015a). Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015b). Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2022). Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., and Murphy, K. (2017). Speed/accuracy trade-offs for modern convolutional object detectors.

Hugging Face (2024a). Create and manage a hugging face account & repositories. https://huggingface.co/docs/hub/repositories-how-to. Documentation.

Hugging Face (2024b). Dataset cards. https://huggingface.co/docs/hub/dataset-cards. Documentation.

Hugging Face (2024c). Datasets — loading and dataset hub. https://huggingface.co/docs/datasets. Documentation.

Hugging Face (2024d). Hugging face hub — overview. https://huggingface.co/docs/hub/index. Documentation.

Hugging Face (2024e). Hugging face hub — repositories and identifiers. https://huggingface.co/docs/hub/repositories-getting-started. Documentation.

Hugging Face (2024f). Hugging face hub — revisions, branches and commits. https://huggingface.co/docs/hub/repositories-versions. Documentation.

Hugging Face (2024g). Model cards. https://huggingface.co/docs/hub/model-cards. Documentation.

Jing, L., Wang, R., Ren, W., Dong, X., and Zou, C. (2024). Pad: Patch-agnostic defense against adversarial patch attacks.

Jordan, M. I. and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.

Kaggle (2025). Kaggle — datasets (homepage figure, october 2025). https://www.kaggle.com/datasets.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.

Lian, J., Pan, J., Wang, L., Wang, Y., Chau, L.-P., and Mei, S. (2025). Padetbench: Towards benchmarking physical attacks against object detection.

Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., and Belongie, S. (2017). Feature pyramid networks for object detection.

Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. (2015). Microsoft coco: Common objects in context.

Liu, J., Levine, A., Lau, C. P., Chellappa, R., and Feizi, S. (2022). Segment and complete: Defending object detectors against adversarial patch attacks with robust patch detection.

Liu, L., Ouyang, W., Wang, X., Fieguth, P., Chen, J., Liu, X., and Pietikäinen, M. (2019a). Deep learning for generic object detection: A survey.

Liu, X., Yang, H., Liu, Z., Song, L., Li, H., and Chen, Y. (2019b). Dpatch: An adversarial patch attack on object detectors.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. (2018). Mixed precision training. In *International Conference on Learning Representations*.

Mironov, I. (2017). Rényi differential privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 263–275. IEEE.

ModelScope (2025). Modelscope — datasets catalogue (october 2025). https://www.modelscope.cn/datasets.

Neubeck, A. and Van Gool, L. (2006). Efficient non-maximum suppression. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 3, pages 850–855.

Nicolae, M.-I., Sinn, M., Tran, M. N., Buesser, B., Rawat, A., Wistuba, M., Zantedeschi, V., Baracaldo, N., Chen, B., Ludwig, H., et al. (2018). Adversarial robustness toolbox v1.0.0. *arXiv preprint arXiv:1807.01069*.

Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. (2016). The limitations of deep learning in adversarial settings. *2016 IEEE European Symposium on Security and Privacy*, pages 372–387.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection.

Ren, S., He, K., Girshick, R., and Sun, J. (2016). Faster r-cnn: Towards real-time object detection with region proposal networks.

Singh, S., Yadav, A., Jain, J., Shi, H., Johnson, J., and Desai, K. (2024). Benchmarking object detectors with coco: A new path forward.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014a). Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014b). Intriguing properties of neural networks.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Wang, J. and Kanwar, V. (2019). BFloat16: The secret to high performance on cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

Wu, H., Rowlands, S., and Wahlström, J. (2024). A human-in-the-middle attack against object detection systems. *IEEE Transactions on Artificial Intelligence*, 5(10):4884–4892.

Yuan, X., He, P., Zhu, Q., and Li, X. (2019). Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824.

Zhang, H. and Wang, J. (2019). Towards adversarially robust object detection.

Zhao, Z.-Q., Zheng, P., tao Xu, S., and Wu, X. (2019). Object detection with deep learning: A review.

Zhou, Z., Li, B., Song, Y., Yu, Z., Hu, S., Wan, W., Zhang, L. Y., Yao, D., and Jin, H. (2024). Numbod: A spatial-frequency fusion attack against object detectors.

Zou, A., Wang, Z., Carlini, N., Nasr, M., Kolter, J. Z., and Fredrikson, M. (2023). Universal and transferable adversarial attacks on aligned language models. https://arxiv.org/abs/2307.15043. arXiv preprint arXiv:2307.15043v2.