TECHNISCHE UNIVERSITÄT WIEN

Selection Guidelines for Backdoor-based Model Watermarking

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Technische Mathematik

eingereicht von

Isabell Lederer, BSc
Matrikelnummer 01526148

ausgeführt am Institut für Information Systems Engineering

der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Univ.Lektor Mag.rer.soc.oec. Dipl.-Ing. Rudolf Mayer

Wien, 15. September 2021

Isabell Lederer	Andreas Rauber

TECHNISCHE UNIVERSITÄT WIEN

Selection Guidelines for Backdoor-based Model Watermarking

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Technical Mathematics

by

Isabell Lederer, BSc
Registration Number 01526148

to the Institute of Information Systems Engineering

at the Faculty of Informatics at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Univ.Lektor Mag.rer.soc.oec. Dipl.-Ing. Rudolf Mayer

Vienna, 15th September, 2021

Isabell Lederer | Andreas Rauber

Erklärung zur Verfassung der Arbeit

Isabell Lederer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. September 2021

Isabell Lederer

Acknowledgements

In the process of planning, writing and elaborating this thesis, I received broad support from a multitude of people to whom I would like to express my gratitude and acknowledgement.

It goes without saying that I am exceptionally thankful for the guidance by Rudolf Mayer. His advice on methodical approach and research were of fundamental importance, and I highly appreciate his consistent and constructive feedback as well as his collegial ways. Equally, I would like to express my sincere appreciation for the supervision by professor Andreas Rauber and his essential consultation throughout the entire progression of this thesis.

I gratefully address my dear fellow students, especially Karine and Nemanja, my colleagues at SBA Research, and those at the TUtheTOP alumni club for their cordial companionship, motivational support and honest feedback. Many thanks to Tanja and Monika for proofreading this thesis.

I would like to thank my friends and family for the continuous motivation, the interest in the subject area, which is foreign to many of them, and the constant participation, especially on particularly challenging days. I am grateful for my brother's enthusiasm for what I do and his sincere admiration in seeing me evolve. I thank Alina for reminding me that life is not all about work and taking me out for various activities.

A very special thank you goes, of course, to my partner Florian, who not only endures my ups and downs but also supports me in all my projects, regardless of how big and ambitious they are. I am enormously thankful for his always encouraging words and his great cooking.

Above all, I would like to thank my mother, whose fundamentally positive, unwavering view of the future has given me the will, self-confidence and perseverance without which I would undoubtedly not have been able to cope with the numerous and occasionally hopeless seeming tasks and challenges of my educational path to date, as well as of my private and professional life.

Kurzfassung

Da die kommerzielle Nutzung von maschinellem Lernen (ML) immer weiter verbreitet ist und die steigende Komplexität von ML-Modellen aufwendiger und damit teurer zu trainieren wird, wächst auch die Dringlichkeit, geistiges Eigentums in diesen Modellen zu schützen. Im Vergleich zu Technologien, die sich auf ein solides Verständnis von Bedrohungen, Angriffen und Verteidigungsmöglichkeiten zum Schutz ihres geistigen Eigentums stützen können, ist die Forschung in dieser Hinsicht bei ML noch sehr fragmentiert. Dies ist mitunter auf das Fehlen einer einheitlichen Sichtweise und einer gemeinsamen Taxonomie dieser Aspekte zurückzuführen.

In dieser Arbeit werden die Erkenntnisse zum Schutz des geistigen Eigentums in ML systematisiert, wobei der Schwerpunkt auf Bedrohungen und Angriffen liegt, die für einige der bisher bestehenden Systeme festgestellt wurden, sowie auf den bisher vorgeschlagenen Schutzmaßnahmen. Wir entwickeln ein umfassendes Bedrohungsmodell für das geistige Eigentum in ML und kategorisieren Angriffe und Abwehrmaßnahmen in einer einheitlichen und konzisen Taxonomie, um auf diese Weise die Brücke zwischen ML und zukunftsweisender Sicherheit zu schlagen.

Später konzentrieren wir uns auf Backdoor-basiertes Watermarking für Deep Neural Networks zur Bildklassifizierung und definieren verschiedene Parameter für eine umfassende Studie dieser Ansätze. Dies ist von grundlegender Bedeutung für die Bewertung der verschiedenen Methoden und die Formulierung des Leitfadens. Schließlich wählen wir eine Teilmenge dieser Parameter aus und vergleichen die Methoden, um eine Empfehlung für eine Watermarking-Methode auf Basis eines ML-Settings zu geben.

Abstract

With commercial uses of Machine Learning (ML) becoming more wide-spread, while at the same time ML models becoming more complex and expensive to train, the Intellectual Property Protection (IPP) of trained models is becoming a pressing issue. Unlike other domains that can build on a solid understanding of the threats, attacks and defences available to protect their IP, the research in this regard in ML is still very fragmented. This is also due to a lack of a unified view and a common taxonomy of these aspects.

In this thesis, we systematise findings on IPP in ML, focusing on threats and attacks identified on these systems and defences proposed to date. We develop a comprehensive threat model for IP in ML, and categorise attacks and defences within a unified and consolidated taxonomy, thus bridging research from both the ML and security communities.

Later on, we focus on backdoor-based watermarking approaches for Deep Neural Networks for image classification and define different parameters and settings for a comprehensive study of these approaches. This will be fundamental for evaluating the different methods and formulating the selection guidelines. Finally, we choose a subset of these parameters and compare the methods in order to provide a recommendation for a watermarking method based on the ML setting.

Contents

CHAPTER 1

Introduction

In many Machine Learning (ML) settings, training an effective model from scratch, especially complex and powerful models such as a Deep Neural Network (DNN), is (i) computationally very expensive, (ii) requires expertise for setting parameters, and (iii) the amount of data needed is often not accessible or expensive to obtain. Security concerns become more prominent when these models are made available to other parties or customers, e.g. in Machine Learning as a Service (MLaaS), or when otherwise licensing model use. Thus, model owners that have invested significant resources to train a model and want to offer it to customers start to consider Intellectual Property Protection (IPP) methods, e.g. watermarking for verifying the ownership, and model access control for preventing unauthorised usage of a model. In the last few years, we, therefore, have seen an increase in research on IPP techniques for ML models. Many watermarking methods, requiring either black-box or white-box access, have recently been proposed, based on techniques such as backdoor embedding via data poisoning and regularisation. At the same time, several studies have shown the vulnerability of some of these schemes against novel attacks. Similar observations hold true for model access control techniques. A comprehensive overview of the field, including a unified nomenclature and taxonomy, as well as an empirical evaluation, is still missing.

Our contributions, in this thesis, are:

- A systematic overview on research related to IPP of ML, focusing on watermarking and fingerprinting, in particular, based on a methodological literature review
- A taxonomy to categorise model watermarking and fingerprinting schemes, based on a methodological literature review and a categorisation of 26 approaches
- An analysis of vulnerability to attacks designed to break the IPP schemes, based on a methodological literature review

- An implementation and evaluation of selected state-of-the-art backdoor-based watermarking schemes by training 191 models
- Guidelines on how to choose a fitting backdoor-based watermarking scheme for a given setting

The remainder of this thesis is structured as follows. Our research methodology is described in Chapter 2. Chapter 3 provides definitions and background to machine learning, deep neural networks, watermarking, and fingerprinting. Chapter 4 introduces our taxonomy of IPP methods, the threat model and attacks. Chapter 5 provides an overview on state-of-the-art watermarking and fingerprinting approaches and discusses the vulnerability to various attacks. Our research questions and study settings are defined in Chapter 6. Chapter 7 provides an empirical comparison of the chosen backdoor-based watermarking methods, including the implementation and evaluation. Conclusions, selection guidelines and future work are discussed in Chapter 8. Appendix A.1 provides a detailed list of dependencies. Additional figures are provided in Appendix A.2.

CHAPTER 2

Methodology

In this chapter, we describe the methodological approach for the literature search, describe the system used behind the literature search and give an idea on the scale of this topic. Furthermore, we are going to introduce the methodology for the empirical evaluation.

2.1 Literature search

In preparation for this master thesis, we performed an extensive literature search and documented every step to make it reproducible. Figure 2.1 shows the workflow of our literature search process. The complete documentation of the search process including the search strings and results with the retrieved literature will be made available in the GitHub project `https://github.com/mathebell/model-watermarking`.

We distinguish between the following types of publications: *formal literature* (FL), i.e. peer-reviewed literature such as book sections, conference papers, journal articles, and

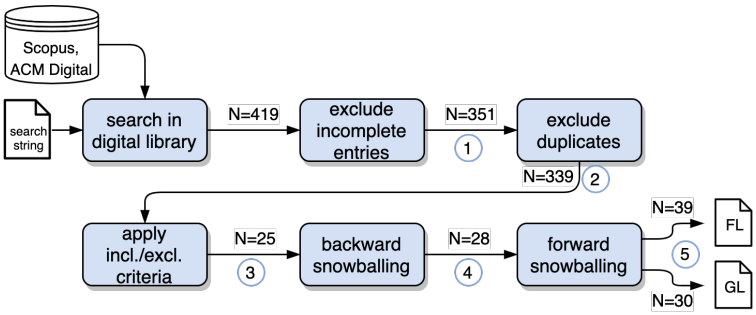
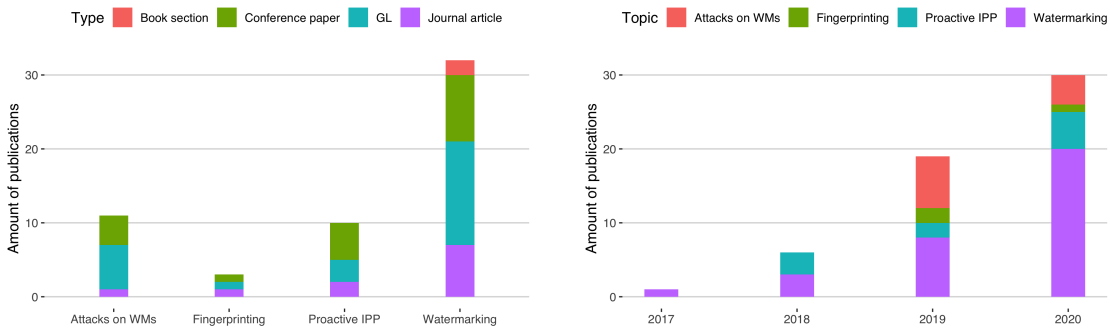


Figure 2.1: Literature search process workflow. In every step we denote the number of publications by $N = x$. The numbers 1 to 6 correspond to the CSV-files which contain all the retrieved literature in the particular step.



(a) Across different topics regarding ML IPP

(b) Across the years for different topics

Figure 2.2: Literature distribution

grey literature (GL), i.e. literature that did not undergo a peer-review process, such as pre-prints (published on repositories such as arXiv, or university repositories, author's websites, etc.) However, in its definition, grey literature does not only consist of pre-prints but can also be formed by blogs, interviews, wikis and many more document types [74], [1]. In this thesis, as the subject of watermarking of ML models has not produced such types of GL yet, it does however only consist of pre-prints.

Figure 2.2a shows the distribution of publications across the different topics in the various literature types. *Attacks on WMs* include papers that propose attacks specifically crafted to disable a watermarking method or render it useless. *Fingerprinting* and *Watermarking* includes papers proposing a fingerprinting or watermarking method for ML models. *Proactive IPP* includes papers on unrobust models and model access methods (cf. Figure 4.1); these are not in the focus and are not discussed in more detail in this thesis. We can see that most papers were published regarding watermarking; however, there is also a significant number of papers on attacks published. Note, that some publications include both a novel attack to a scheme and a novel watermarking scheme, which is immune to this attack. Also, not all publications covering attacks are classified as such, as there is often a specific attack to a scheme, and a novel scheme immune to this attack, proposed in the same publication.

Figure 2.2b, on the other hand, shows the distribution of publications across the publishing years. We see a rise in interest for this topic, with papers on attacks being mostly published in the last two years only.

2.1.1 Inclusion/exclusion criteria

In order to provide reproducible documentation of the literature research, we defined the following inclusion and exclusion criteria to find the most relevant literature for the topic of IPP of ML models. Our inclusion criteria are:

- Literature which proposes an IPP scheme for ML models

- Literature which proposes an attack on an IPP scheme for ML models
- Literature which evaluates or compares earlier schemes

Our exclusion criteria are:

- (Near) Duplicates [1]
- Literature which only *uses* ML for multimedia watermarking, such as image watermarking
- Literature that only *applies* previously published IP protection schemes, without a novel or large-scale evaluation

2.2 Empirical evaluation

As a first step we formulate research questions and define a study setting in Chapter 6, which includes benchmark architectures and datasets. We choose watermarking methods according to our defined selection criteria (cf. Chapter 6) and implement them in Section 7.1. Afterwards, we analyse them based on experiments in Section 7.2. We synthesise our findings, answer the research questions and formulate selection guidelines in Chapter 8.

[1] If the titles are different but the content is very similar, we include all versions of the literature and note that. Later on, we will cite only the most complete version as suggested by Kitchenham et al. [58].

CHAPTER 3

Background

This chapter aims to equip the reader with the necessary background for the rest of the work. The definitions are formulated in a mathematical way. We expect the reader to be familiar with the notation.

3.1 Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) and was defined by Tom Mitchell as "the study of computer algorithms that allow computer programs to automatically improve through experience" [76].

The main objective in ML is the problem $\mathcal{P}$, i.e. the task that the ML model is trained to solve. Common problems in ML are [77]:

- *Classification*: "this is the problem of assigning a category to each example. For example, document classification consists of assigning a category such as politics, business, sports, or weather to each document, while image classification consists of assigning to each image a category such as car, train, or plane. The number of categories in such tasks is often less than a few hundred, but it can be much larger in some difficult tasks such as in text classification or speech recognition."
- *Regression*: "this is the problem of predicting a real value for each item. Examples of regression include prediction of stock values or that of variations of economic variables. In regression, the penalty for an incorrect prediction depends on the magnitude of the difference between the true and predicted values, in contrast with the classification problem, where there is typically no notion of similarity between various categories."
- *Clustering*: "this is the problem of partitioning a set of items into homogeneous subsets. Clustering is often used to analyse very large data sets. For example, in

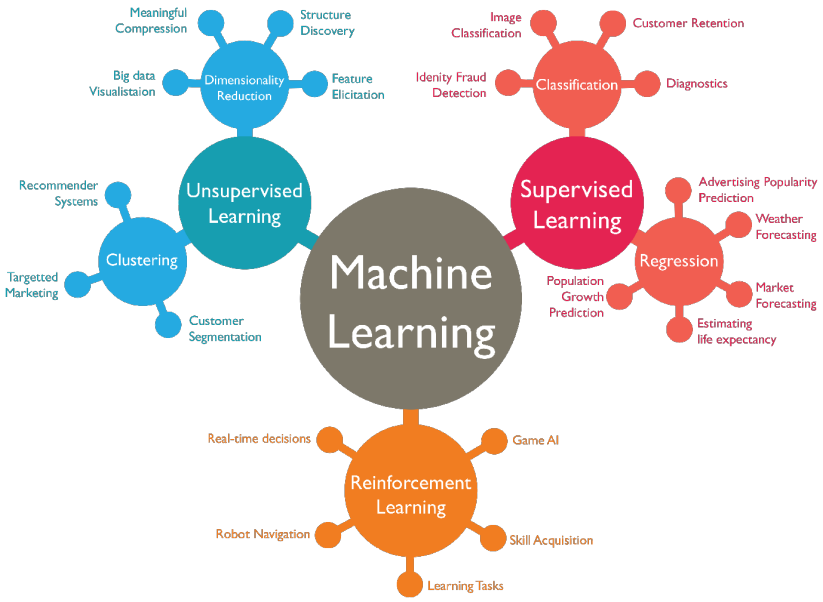


Figure 3.1: Three main categories of ML, their algorithms and use cases. Source: [92]

the context of social network analysis, clustering algorithms attempt to identify natural communities within large groups of people."

These problems are usually assigned to one of the three main categories in ML (cf. Figure 3.1):

1. **Supervised Learning** – the model learns to make predictions based on a set of labelled data, e.g. classification or regression.

2. **Unsupervised Learning** – the model learns to find patterns in an unlabelled dataset, e.g. clustering or dimensionality reduction.

3. **Reinforcement Learning** – the model learns to master a task based on a feedback loop, e.g. playing a game.

We define typical terminology for Machine Learning [77]:

- *Examples*: "Items or instances of data used for learning or testing. In image classification, these examples are images which we will use for learning and testing."

- *Labels*: "Values or categories assigned to examples. In image classification, examples are assigned specific categories, for instance, car, train or plane."

- *Parameters:* are values that control the behaviour of an ML model and are the instances that are updated during model training. We denote parameters, often also called *weights*, by a vector $\mathbf{w}$, but also θ is a commonly used notation.

- *Hyperparameters*: "Free parameters that are not determined by the learning algorithm, but rather specified as inputs to the learning algorithm, such as the learning rate or the batch size."

- *Training set*: "Examples or example-label pairs used to train a learning algorithm."

- *Validation set*: "Examples or example-label pairs used to select appropriate values for the learning algorithm's hyperparameters", or early stopping. Early stopping is a mechanism to stop training when the validation set performs best, in order to prevent from overfitting (cf. Section 3.1.1).

- *Test set*: "Examples or example-label pairs used to evaluate the effectiveness of a learning algorithm. The test set is separate from the training and validation set and is not made available in the learning stage." The training, validation and test sets are pairwise disjoint subsets of the dataset. In Supervised Learning, the training, validation and test sets consist of example-label pairs, in unsupervised learning, on the other hand, only of examples.

- *Loss function*: "A function, that measures the difference, or *loss*, between a predicted label and a true label." We denote the loss function as $\mathcal{L}(\mathbf{w})$, where $\mathbf{w}$ is the ML model's parameter vector, since we usually want to minimise the loss function according to the model's parameters $\mathbf{w}$ (cf. Equation (3.4)). However, in practice, the loss function is computed with the input of the true label and predicted label, and could therefore be denoted as $\mathcal{L}(y, \hat{y})$, where y is the true and $\hat{y}$ the predicted label.

3.1.1 Supervised Machine Learning

In this work, we focus on Supervised Learning, especially image classification with Deep Learning (cf. Section 3.1.1). Let $\mathbf{X} \in \mathcal{D} \subset \mathbb{R}^n$, $n \geq 1$ be an input data point from a dataset $\mathcal{D}$ and y the corresponding label, then we denote $f : \mathbb{R}^n \to \mathbb{R}$ as the function that maps the label to the example $f(\mathbf{X}) = y$. We therefore train a supervised model $\mathcal{F}_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}$ to predict the data as well as possible, i.e. $\mathcal{F}_{\mathbf{w}}(\mathbf{X}) \approx f(\mathbf{X})$, $\forall \mathbf{X} \in \mathcal{D}$ with the trained parameter (weight) vector $\mathbf{w} \in \mathbb{R}^m$. The goal is to train the model in such a way that it predicts the right label also for unseen data, i.e. data that was not in the training set.

In practice, the *performance* of different models is compared via the model's *accuracy* on the test set, the test accuracy, i.e. the fraction of total records that are correctly predicted by the model. The *accuracy error* is then the difference between 100% and the accuracy. In notation of the confusion matrix (cf. Table 3.1) the accuracy on the dataset

	Predicted Yes	Predicted No
Actual Yes	True Positive (TP)	False Negative (FN)
Actual No	False Positive (FP)	True Negative (TN)

Table 3.1: Confusion matrix of a two-class problem.

$\mathcal{D}$ is computed as

$$\text{Accuracy}(\mathcal{F}_{\mathbf{w}}, \mathcal{D}) = \frac{TP + TN}{TP + TN + FP + FN}. \tag{3.1}$$

In the context of watermarking, we will use the terms false positive and false negative regularly, however, with a slightly different meaning. A watermarking method should have both a low false positive and a low false negative rate when triggering watermarks. We will explain the terms in Section 5.1.

For classification problems with a large number of classes, the *top N accuracy* is commonly used, since the model might not be able to predict the right class exactly, but the right class might be among the top N predictions. The previously explained accuracy is then the top 1 accuracy as the prediction is only counted as TP or TN when it hits exactly the right class. For the top 3 accuracy, the prediction is counted as a TP or TN also when the true class is not the first, but among the first 3 predictions.

During model training, the model parameters are learned based on the training data. Some learning algorithms iteratively adapt their parameters, by minimising some kind of a loss function. Training accurate models often requires multiple training iterations, but not always as (simple cases of) linear regression can be solved non-iteratively as well.

There is a number of different ML models, e.g. *Linear Regression*, *Decision Tree* [13], *k-Nearest Neighbor* (k-NN) [8], *Perceptron* [29], etc. In the text below, we describe linear regression and perceptron in more detail.

Linear Regression In a general formulation, linear regression finds the best fit line through the data, i.e. it finds the ideal parameter vector $\mathbf{w} = (w_0, w_1, \ldots, w_{m-1})^\top \in \mathbb{R}^m$ (here $m = n + 1$), so that for unknown data $\mathbf{X} = (x_1, x_2, \ldots, x_n)^\top \in \mathbb{R}^n$ the real value is predicted by

$$\mathcal{F}_{\mathbf{w}}(\mathbf{X}) = \sum_{i=1}^{n} w_i x_i + w_0 \tag{3.2}$$

$$= \mathbf{w}^\top \mathbf{X} + w_0, \tag{3.3}$$

where w_0 is the so-called *bias* and often denoted as b. This is done by minimising the mean squared error, which acts as loss function $\mathcal{L}$. Let the labelled training set $\mathcal{D} = \{(\mathbf{X}^1, y^1), (\mathbf{X}^2, y^2), \ldots, (\mathbf{X}^k, y^k)\}$ be of size k, then the corresponding optimisation

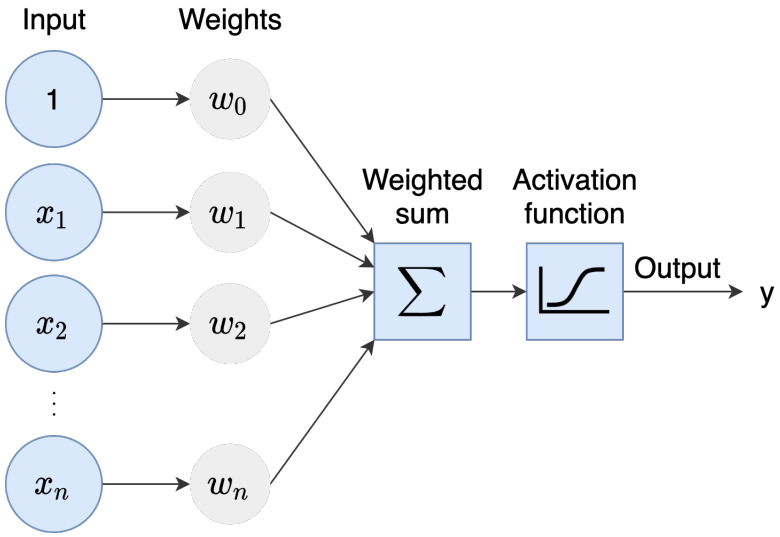


Figure 3.2: A perceptron

problem is

$$\min_{\mathbf{w}\in\mathbb{R}^m} \mathcal{L}(\mathbf{w}) = \min_{\mathbf{w}\in\mathbb{R}^m} \frac{1}{k} \sum_{j=1}^{k} \left(\mathbf{w}^\top \mathbf{X}^j + w_0 - y^j\right)^2. \tag{3.4}$$

Perceptron A *perceptron* is a binary classifier and builds the basis for an *Artificial Neural Network* (ANNs). An ANN, motivated by the human brain, is a collection of *nodes*, or *neurons*, which are connected through *layers*. A layer consists of several nodes and, in its simplest form, a feed-forward layer, passes the information to (and only to) the next layer. Perceptron is the simplest form of ANN. It consists of only one neuron and outputs either 0 or 1. Figure 3.2 shows a perceptron that takes n-dimensional data as input. The output is computed by a linear combination of the input $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ using the weights $\mathbf{w} = (w_0, \ldots, w_n)$

$$\sum_{i=1}^{n} w_i x_i + w_0 = \mathbf{w}^\top \mathbf{X} + w_0 \tag{3.5}$$

and applying a threshold step function with the threshold s

$$y = \begin{cases} 1 & \text{for } \mathbf{w}^\top \mathbf{X} + w_0 \geq s \\ 0 & \text{for } \mathbf{w}^\top \mathbf{X} + w_0 < s \end{cases}. \tag{3.6}$$

During training, the weights of the perceptron are updated iteratively. Given a dataset consisting of example-label pairs $\mathcal{D} = \{(\mathbf{X}^1, y^1), (\mathbf{X}^2, y^2), \ldots, (\mathbf{X}^k, y^k)\}$, we pass each

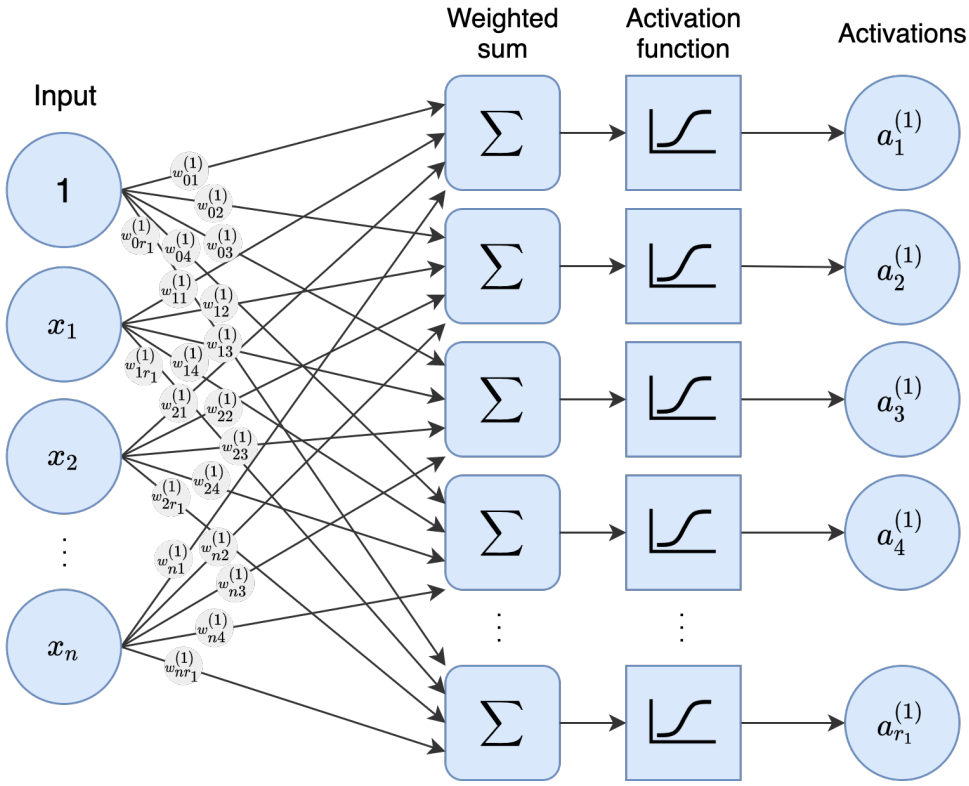


Figure 3.3: Computation of first layer in an MLP.

example through the perceptron and update the weights according to the prediction. Let $\hat{y}^i$ be the predicted label for the input $\mathbf{X}^i$, then the weights are updated by

$$w_i^{\text{new}} = w_i + \alpha(y^i - \hat{y}^i)x_i, \quad i = 1, \dots n,$$

where α is the learning rate, which is an instance of the hyperparameters. This process is repeated until the prediction is correct for all examples in the dataset. More complex types of ANNs are discussed in the following section.

Deep Learning

Deep Learning [33] (DL) is a type of Machine Learning that achieves remarkable results on many tasks, by dividing a problem into many sub-problems. The basis for Deep Learning is a *Multi-Layer Perceptron* (MLP). An MLP is a perceptron with at least one hidden layer. The hidden layer is again composed of perceptrons, as it is visualised in Figure 3.3. A feed-forward *Deep Neural Network* (DNN) is an MLP with multiple (at least two) hidden layers. In general, the term DNN can be any ANN, not only MLP, with more than one hidden layer.

We will now explain the computations in an MLP, but first we have to define the notation. We denote the model's parameters or weights with $\mathbf{w}$, but also θ is frequently used in

other literature. As shown in Figure 3.3, we denote the input to a neural network as $\mathbf{X} = (x_1, x_2, \ldots, x_n)^\top \in \mathbb{R}^n$, the weights connecting the input to the first node of the first hidden layer as $\mathbf{w}_1^{(1)} = \left(w_{11}^{(1)}, w_{21}^{(1)}, \ldots, w_{n1}^{(1)}\right)^\top$, the weights connecting the input to the second node of the first hidden layer as $\mathbf{w}_2^{(1)} = \left(w_{12}^{(1)}, w_{22}^{(1)}, \ldots, w_{n2}^{(1)}\right)^\top$, etc. Let n be the size of the input and r_1 the size of the first hidden layer, then the weights for the first hidden layer can be combined into a matrix (and also analogously for all the other layers):

$$\mathbf{W}^{(1)} = \left(\mathbf{w}_1^{(1)}, \ldots, \mathbf{w}_{r_1}^{(1)}\right) = \begin{pmatrix} w_{11}^{(1)} & \cdots & w_{1r_1}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{n1}^{(1)} & \cdots & w_{nr_1}^{(1)} \end{pmatrix} \tag{3.7}$$

The first hidden layer's activations are denoted as $\mathbf{a}^{(1)} = \left(a_1^{(1)}, a_2^{(1)}, \ldots, a_{r_1}^{(1)}\right)^\top$, the second hidden layer's nodes as $\mathbf{a}^{(2)} = \left(a_1^{(2)}, a_2^{(2)}, \ldots, a_{r_2}^{(2)}\right)^\top$. The weights connecting $\mathbf{a}^{(1)}$ to $\mathbf{a}^{(2)}$ are denoted as $\mathbf{W}^{(2)}$. And finally, if the neural network consists of L hidden layers, then the last hidden layer $\mathbf{a}^{(L)}$ is connected to the output layer $\mathbf{a}^{(O)}$ through $\mathbf{W}^{(L)}$. Let the size of the l-th layer be r_l.

With g being the activation function, the first hidden layer's nodes are then computed as

$$a_1^{(1)} = g\left(\sum_{i=1}^{n} w_{i1}^{(1)} x_i + w_{01}^{(1)}\right) = g\left({\mathbf{w}_1^{(1)}}^\top \mathbf{X} + w_{01}^{(1)}\right), \tag{3.8}$$

$$\vdots$$

$$a_{r_1}^{(1)} = g\left(\sum_{i=1}^{n} w_{ir_1}^{(1)} x_i + w_{0r_1}^{(1)}\right) = g\left({\mathbf{w}_{r_1}^{(1)}}^\top \mathbf{X} + w_{0r_1}^{(1)}\right), \tag{3.9}$$

and directly fed into the computation for the second hidden layer:

$$a_1^{(2)} = g\left(\sum_{j=1}^{r_1} w_{j1}^{(2)} a_j^{(1)} + w_{01}^{(2)}\right) = g\left(\sum_{j=1}^{r_1} w_{j1}^{(2)} g\left(\sum_{i=1}^{n} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_{01}^{(2)}\right), \tag{3.10}$$

$$\vdots$$

$$a_{r_2}^{(2)} = g\left(\sum_{j=1}^{r_1} w_{jr_2}^{(2)} a_j^{(1)} + w_{0r_2}^{(2)}\right) = g\left(\sum_{j=1}^{r_1} w_{jr_2}^{(2)} g\left(\sum_{i=1}^{n} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_{0r_2}^{(2)}\right). \tag{3.11}$$

Iteratively, we obtain the formula for the l-th node in the output layer:

$$a_l^{(O)} = g\left(\sum_{m=l}^{r_L} w_{ml}^{(O)} a_m^{(L)} + w_{0l}^{(O)}\right) \tag{3.12}$$

$$= g\left(\sum_{m=l}^{r_L} w_{ml}^{(O)} \cdots g\left(\sum_{j=1}^{r_1} w_{jk}^{(2)} g\left(\sum_{i=1}^{n} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_{0k}^{(2)}\right) \cdots + w_{0l}^{(O)}\right) \tag{3.13}$$

The activation function g decides, depending on the weighted sum, if a node should be activated or not, i.e. if the output of the node will be used for further computations or not. Common activation functions are the *Heaviside step function*

$$g(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}, \tag{3.14}$$

a piecewise linear function, e.g.,

$$g(x) = \begin{cases} 1 & \text{for } x \geq \frac{1}{2} \\ x + \frac{1}{2} & \text{for } -\frac{1}{2} < x < \frac{1}{2} \\ 0 & \text{for } x \leq -\frac{1}{2} \end{cases}, \tag{3.15}$$

the *sigmoid function* (which is used as a symbolic example in Figure 3.3)

$$g(x) = \frac{1}{1 + e^{-t}}, \tag{3.16}$$

or, also a piecewise linear function, the *Rectified Linear Unit* (ReLU) activation function

$$g(x) = \max(0, x). \tag{3.17}$$

The latter one is commonly used in Deep Learning, as only the neurons with a positive value are activated and therefore not all of them get activated at once as they would with a sigmoid function, being far more computationally efficient. There exist many other variations of ReLU, e.g. *leaky ReLU* [73] and *parameterised ReLU* (PReLU) [46], just to name a few. Leaky ReLU activates neurons also with a negative value, but weights it small:

$$g(x) = \begin{cases} x & \text{for } x > 0 \\ 0.01x & \text{for } x \leq 0 \end{cases} \tag{3.18}$$

The PReLU takes the idea further and introduces a parameter which is learned during training:

$$g(x) = \begin{cases} x & \text{for } x > 0 \\ ax & \text{for } x \leq 0, \end{cases} \tag{3.19}$$

with a being the additional.

In order to solve the classification problem, we need a loss function that will be minimised during model training. The function that is commonly used in classification is the *Categorical Cross Entropy loss* function. It is used whenever an output vector is returned instead of one label. First, the *Softmax* function S is applied to the output vector to get

output probabilities, i.e. values between 0 and 1, with the highest value corresponding to the model's prediction:

$$S\left(a_i^{(O)}\right) = \frac{\exp a_i^{(O)}}{\sum_j \exp a_j^{(O)}} \tag{3.20}$$

Given the input $\mathbf{X}$, a one-hot encoded label vector $\mathbf{y} = (y_1, y_2, \cdots, y_C) \in \{0,1\}^C$, where C is the number of classes in the model, and the output vector of the model $\mathcal{F}_\mathbf{W}(\mathbf{X}) = \mathbf{a}^{(O)}(\mathbf{X})$, the categorical cross entropy loss is then computed by

$$\begin{aligned}
\mathcal{L}(\mathcal{F}_\mathbf{W}(\mathbf{X}), \mathbf{y}) &= -\sum_i^C y_i \log\left(\frac{\exp a_i^{(O)}(\mathbf{X})}{\sum_j \exp a_j^{(O)}(\mathbf{X})}\right) && (3.21)\\
&= -\log\left(\frac{\exp a_p^{(O)}(\mathbf{X})}{\sum_j \exp a_j^{(O)}(\mathbf{X})}\right) && (3.22)\\
&= -\log\left(S\left(a_p^{(O)}\right)\right), && (3.23)
\end{aligned}$$

with p being the position in the output vector corresponding to the true label. The second equation holds since the label vector is one-hot encoded and all of the other terms are cancelled out. Therefore, the cross entropy loss returns high values for a small value in the output probabilities vector and vice versa, working as a penalty function. A perfect prediction with a one-hot encoded output probabilities vector would results in a loss of zero, which the model aims to learn during training.

Since the output vector $\mathbf{a}^{(O)}$ mainly depends on the weights $\mathbf{W} = (\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L)})$ of the DNN, we can formulate the optimisation problem for the DNN as

$$\min_{\mathbf{W}\in\mathbb{R}^m} \mathcal{L}(\mathbf{W}, \mathcal{F}_\mathbf{W}(\mathbf{X}), \mathbf{y}) = \max_{\mathbf{W}\in\mathbb{R}^m} \log\left(\frac{\exp a_p^{(O)}}{\sum_j \exp a_j^{(O)}}\right), \tag{3.24}$$

where here $\mathbf{W} \in \mathbb{R}^m$ denotes the flattened version of the matrix $\mathbf{W}$ with length $m = nr_1 + \sum_{i=2}^{L} r_{i-1}r_i$.

In order to solve this optimisation problem, a multitude of optimisation algorithms has been introduced [33]. *Gradient descent* is the most basic gradient-based optimisation algorithm and forms the basis for the more advanced ones. The algorithm updates the parameters after computing the gradients based on the whole training set and is therefore not recommended to use with large datasets. Let $\mathcal{D} = \{(\mathbf{X}^1, y^1), (\mathbf{X}^2, y^2), \ldots, (\mathbf{X}^k, y^k)\}$ the training set, then the update rule for the parameters is as follows

$$\mathbf{W}^{\text{new}} = \mathbf{W} + \alpha\nabla_\mathbf{w} \sum_{i=1}^{k} \mathcal{L}(\mathbf{W}, \mathcal{F}_\mathbf{W}(\mathbf{X}^i), y^i), \tag{3.25}$$

where α denotes the learning rate. The algorithm converges even with a fixed learning rate, provided that the gradients of the loss function are Lipschitz continuous.

A more commonly used algorithm is the *Stochastic Gradient Descent* (SGD), which estimates the gradients based on a batch of training samples. Given a batch of m training samples $\{(\mathbf{X}^1, y^1), (\mathbf{X}^2, y^2), \ldots, (\mathbf{X}^m, y^m)\}$, the gradient estimate is computed by

$$\hat{\mathbf{g}} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\mathbf{W}, \mathcal{F}_{\mathbf{W}}(\mathbf{X}^i), y^i) \tag{3.26}$$

and the parameters are updated after running through a batch by

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \alpha \hat{\mathbf{g}}. \tag{3.27}$$

Another widely used algorithm is the *Adam* optimiser [56], which is an extension of the SGD. Adam is an adaptive gradient descent algorithm, i.e. it adapts the learning rate per parameter. Some models tend to perform better optimised with Adam than with SGD.

Learning rate schedulers adapt the (global) learning rate during training, dependent on the number of iterations, according to a pre-defined schedule. Two commonly used learning rate schedulers are `MultiStepLR` and `CosineAnnealingLR`. *MultiStepLR* reduces the learning rate every n epochs by a rate of, e.g., 0.1. The learning rate gets smaller every n epochs. *CosineAnnealingLR*, however, has a schedule that decreases the learning rate but also increases, in an overall downward trend.

Convolutional Neural Networks (CNNs) [61] are a special type of DNNs where the hidden layers consist of convolutional, pooling and feed-forward layers. CNNs achieved a breakthrough in image recognition. The convolutional filters are applied to the image and enhance special features in an image, e.g. edges, lines or different shapes, and make it therefore possible to learn how to distinguish between objects. A schematic view of this process is shown in Figure 3.4.

Convolutional layers [33] differ in computation compared to ordinary feed-forward layers as shown before in Equations (3.8) and (3.9) to (3.13). Convolutional layers consist of one of more convolutional *filters* or *kernels*. As the name already indicates, the layer performs a mathematical operation called convolution, which we would like to explain in detail. The following definitions are adapted from [33]. The *continuous mathematical convolution* is an operation on two functions of real arguments, say $f, g : \mathbb{R}^n \to \mathbb{C}$, and computed as

$$(f * g)(x) := \int_{\mathbb{R}^n} f(y) g(x - y) dy. \tag{3.28}$$

The mathematical convolution is a weighted mean value, for which every $f(y)$ is weighted by $g(x - y)$. If we deal with discrete functions, which for computational reasons is necessary, we define the *discrete mathematical convolution* as

$$(f * g)[x] := \sum_{y=-\infty}^{\infty} f[y] g[x - y]. \tag{3.29}$$

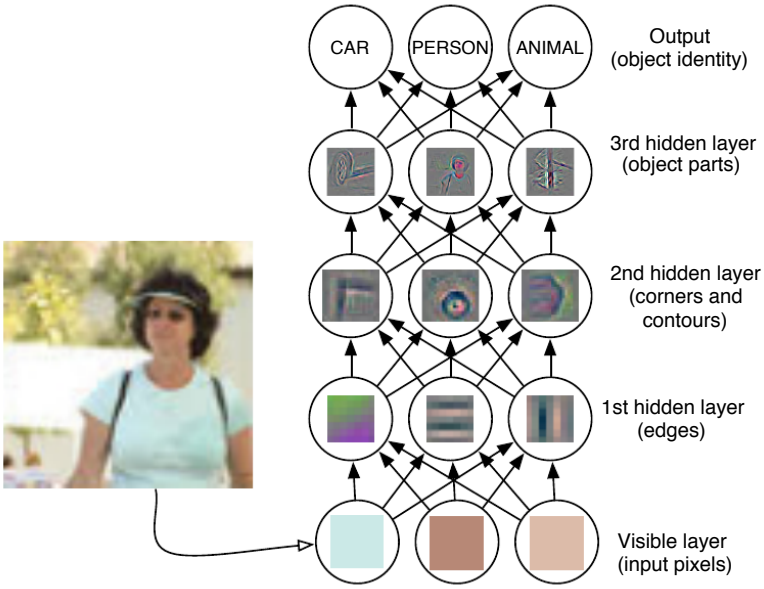


Figure 3.4: Convolutional filters in a Deep Neural Network enhancing features in an image. Usually, the first layers recognise simple features such as lines and corners by comparing the contrast of neighbouring pixels. With this information, the following layers are responsible for recognising whole object parts. In this manner, feeding the pixel information from the input through a series of layers consisting of convolutional, pooling and feed-forward layers, eventually results in a class prediction. Source: [33]

In Machine Learning, however, a *convolution* performs a slightly different operation. Since we usually deal with multi-dimensional input data, e.g. images, of finite size, let the input $\mathbf{X} \in \mathbb{R}^{n \times n}$ and the kernel $\mathbf{K} \in \mathbb{R}^{m \times m}$ be two-dimensional and finite. Then, the convolution is computed as

$$(\mathbf{I} * \mathbf{K})[i,j] = \sum_m \sum_n \mathbf{I}[i+m, j+n]\mathbf{K}[m,n], \quad 1 \leq i,j \leq n-m+1 \tag{3.30}$$

and the output is called *feature map*. A schematic view of the computation is shown in Figure 3.5. Depending on the values of a kernel, the image is processed in a different way, in order to enhance the special features in an image. Those values of a kernel are iteratively updated during training, i.e. the model learns how to enhance features in order to distinguish between different objects. In this way, a CNN is able to classify images depending on special features in an image and therefore better at generalisation compared to image classification with an MLP.

Pooling layers [33] serve as a summary of information after a convolutional layer. They reduce the dimension of data and help to make the representation become invariant to small changes in the input. Common pooling techniques are *max* and *average* pooling. Max pooling takes the maximum value within a rectangular neighbourhood and average pooling takes the average value.

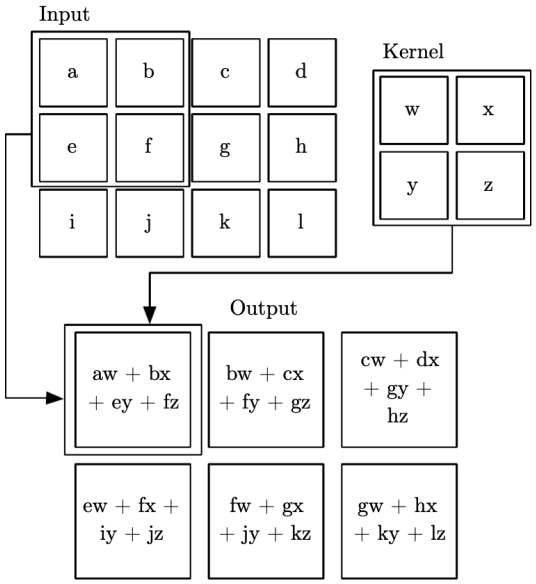


Figure 3.5: Computation of convolutional filter with padding 1. Padding means how much the filter shifts on the input during the computation. With padding 2, e.g., the filter would move forward two values instead of one, in order to compute the next output value. Source: [33]

Recurrent Neural Networks [87] (RNNs) are another special type of DNNs where the connections between neurons are not only limited to the neighbouring layer, like in feed-forward neural networks but can be connected to any other neuron and therefore forming a "memory" for the network. RNNs support sequential data and are especially important for non-static problems like speech recognition. Famous RNNs are the Long Short-Term Memory Network (LSTM) [51] and Gated Recurrent Unit (GRU) [23]. Figure 3.6 shows an illustration for LSTM and GRU. LTSMs and GRUs have internal mechanisms called *gates* that can regulate the flow of information. The gates are trained to distinguish which data in a sequence is important and which is not.

Fine-Tuning

The process of *Fine-Tuning* [31], training a model on different training data with a smaller learning rate, can be used for either improving the model or when using it for a slightly different purpose, in *Transfer Learning* [81]. In this thesis, we use it either to embed a watermark or as a malicious modification to a well-trained model to remove unwanted information, e.g. a watermark.

Overfitting

In training an ML model, *overfitting* [114] is a common phenomenon. Overfitting means that the model is well-trained on the training data, but performs poorly on data that it has not seen before, e.g. the test set. Usually, this happens when the architecture

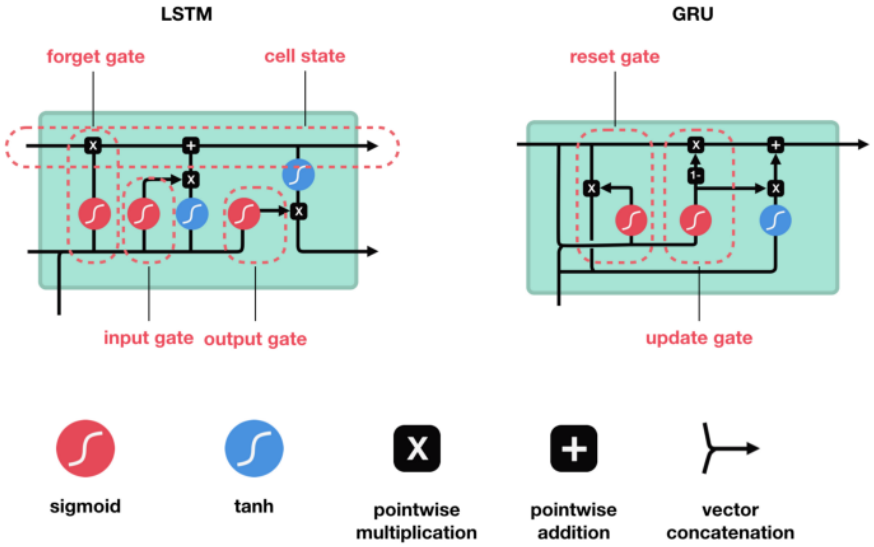


Figure 3.6: LSTM and GRU. Source: [83]

is complex, but there is too little training data available. In this sense, the model has enough degrees of freedom so that it can "memorise" the training data rather than learn how to generalise.

One technique to prevent overfitting in ML models is *regularisation* [114]. During model training, a *parameter regulariser* is used. A regulariser is an additional term in a loss function, often in the form of a penalty term that controls the magnitude of the parameter values. The loss function $\mathcal{L}(\mathbf{w})$ with a regulariser is defined as

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}_0(\mathbf{w}) + \lambda \mathcal{L}_R(\mathbf{w}), \tag{3.31}$$

where $\mathbf{w}$ is the parameter vector, $\mathcal{L}_0$ is the original loss function, $\mathcal{L}_R$ the regularisation term, and λ an adjustable parameter. Several regularisers have been studied in the literature, e.g. the L_2-regularisation, also called Ridge regularisation, or the L_1-regularisation, also called LASSO regularisation.

Another technique to prevent overfitting is *early-stopping* [114]. During model training, the time when the model starts to overfit can be detected. It happens when the test or validation accuracy stops improving, but the train accuracy still does. With early-stopping, we stop the training when the validation loss is minimal. Since we only know that the validation loss is at its minimum when we train several iterations after we reached the minimum. An additional parameter for early-stopping is the *patience*, the number of training iterations after the minimal validation loss was found, to confirm that this is indeed the right timing to stop training.

Parameter Pruning

Parameter pruning [44] is a model compression technique, i.e. it is used to compress the model in order to reduce the storage and computation of the model. It has been shown that by applying parameter pruning, the number of parameters drops by a magnitude without any significant accuracy loss. The model parameters with the smallest absolute value are set to zero because one assumes that the weights with minimal values hold no or negligible information and can be cut out. In this thesis, we consider parameter pruning as either a targeted attack, as the attacker could assume that the pruned weights hold watermark information or as an accidental attack when the attacker wants to drop redundant parameters for storage reasons.

3.1.2 Further ML techniques

In the following, we introduce ML techniques that are mentioned in the thesis, but not as much of importance to the core work.

Federated Learning [112] is an ML technique where multiple parties are involved in training the model on their data, without exchanging the data among each other, mostly for privacy-preserving goals.

Generative Adversarial Networks (GANs) [34] are an ML architecture that uses two DNNs to learn a task – one DNN actually learns the task (the generator) and the other evaluates it (the discriminator). For instance, a common application area is picture generation. The generator learns to create pictures from scratch of, e.g. humans, and the discriminator evaluates its performance by comparing a set of real samples with the generated set. The discriminator has two possible outputs "real" and "fake". The goal of a GAN is that the discriminator outputs "real" for the images generated with the generator.

An *autoencoder* is a special ANN that is commonly used for dimensionality reduction, that is to find a low-dimensional representation of high-dimensional data in an unsupervised manner. This is achieved by learning to copy its input to its output. It consists of an encoder and a decoder, and an internal (hidden) layer that describes a code learned to represent the input.

Knowledge *Distillation* [48] is a type of compression technique that uses knowledge of a neural network (teacher network) to train a new smaller network (student network). A smaller model is computational less expensive and thus more appealing.

Adversarial Examples [96] are a kind of input that is created to fool a model. Usually, an original input (e.g. an image) is perturbed by some specially crafted noise such that the model is unable to classify the generated input correctly. The perturbation is kept minimal, to ideally not be noticeable by a human, or detection methods. Several methods to create Adversarial Examples have been proposed, the most well-known likely being the Fast Gradient Sign Method (FGSM) [36].

3.1.3 Model Extraction Attack

A specific attack against the IP of ML models is the so-called *Model Extraction Attack* (or Model Stealing Attack) [100], which aims to reveal a model's internal characteristic or copy a complete model by only querying its API service. The target can be the architecture, parameters, decision boundary, functionality, or training hyper-parameters of the model.

3.2 Watermarking

Digital watermarking is a well-studied procedure in e.g. multimedia Intellectual Property Protection (IPP) [54] or relational databases [55]. The main idea is to embed a piece of signature in the data, e.g. image or audio, to deter malicious usage. This signature is often intended to be unnoticeable, however, perceptible watermarks are also commonly used in the multimedia domain; examples are logos or copyright notices that are added to images or videos to identify the author. Non-perceptible watermarks, on the other hand, aim to avoid changing the perceptible impression of the data. This is also the type of watermark that we consider for the IPP of ML models. Digital watermarking is thus a form of steganography or information hiding, i.e. the practice of concealing a message within another message. The hidden information must be embedded in such a way that no algorithm can remove or overwrite the watermark. Some recent digital watermarking techniques, e.g. for images, make use of DNNs in the embedding process [122]; similarly, also attacks targeted to remove such watermarks are increasingly using deep learning techniques [91].

In this thesis, we discuss (ML) model watermarking, i.e. the IP that has to be protected is an ML model. Model watermarking is related to multimedia or relational data watermarking, but the techniques differ since the protecting instance changed. Research on watermarking DNNs predominantly addresses image classification (cf. Chapter 5). The introduced terminology is thus strongly influenced by this application of ML, but we believe that the concepts are transferable to other input types as well.

At this point, we want to define the terminology that is common in model watermarking and is used throughout this work. A typical watermarking workflow is shown in Figure 3.7. Watermark *embedding* describes the process in which the watermark is placed into the model, e.g. via fine-tuning. We call watermark *extraction* the process in which the embedded watermark is extracted from the model, but neither in a permanent (which is called *watermark removal*) nor in a malicious way (which is called *watermark detection*). Watermark extraction means that we want to identify if any, and which watermark is placed.

Finally, during watermark *verification*, the extracted watermark is compared to the model owner's watermark in order to prove ownership. Following certain rules (e.g. thresholding the watermark accuracy or (bit) error rate), it is then decided if the watermarks are the same.

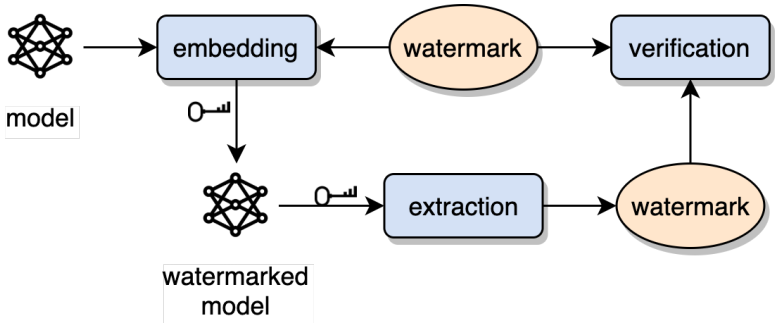


Figure 3.7: A typical watermarking workflow

3.3 Fingerprinting

We can consider fingerprinting as an extension of watermarking. While watermarking has the purpose to verify the *owner* of a digital resource, fingerprinting wants to further trace back to the *(malicious) recipient* of the resource. Therefore, fingerprinting techniques should be capable of embedding multiple, but unique, fingerprints in order to identify the recipient. These multiple fingerprints may be embedded in the same model, or in different versions of the model before distributing it. Similar to watermarking, fingerprinting is already widely used in multimedia areas like images, audio and video [62], relational databases [115], and other digital data types.

CHAPTER 4

Taxonomy of IPP for ML models

In this section, we provide a comprehensive taxonomy of IPP methods for ML models, the threat model and attacks.

We first define our threat model, to then discuss potential schemes to mitigate risks of those threats. We then provide an overview of attacks against those IPP mechanisms.

4.1 Threat Model

To define a threat model we first need to understand the motives of an attacker (or adversary or malicious user). The model owner, i.e. the person that invested resources to obtain an ML model for a specific task, wants to offer the model to some target audience for use. The most prominent reasons for an attacker to redistribute such a model would be (i) having no (or not enough) training data, expertise, time or computational power to train such a model themself, and/or (ii) the unwillingness to agree with the license terms of the obtained model or the fees for using it in a Machine Learning as a Service (MLaaS) setting. We call the model that has to be protected the *target model*, and the attacker's model, which arises from the target model, the *adversary model*. As a threat model, we consider either one of the following situations:

1. **Legal copy:** The model owner distributes the model publicly, either for free, e.g. via a platform such as Model Zoo [2], but with a restrictive license, or for a fee. The attacker then obtains this model and redistributes it via a lucrative API service.

2. **Illegal copy:** The model owner distributes the model as a pay-per-query API service. The attacker then performs a Model Extraction Attack and provides his own lucrative API service.

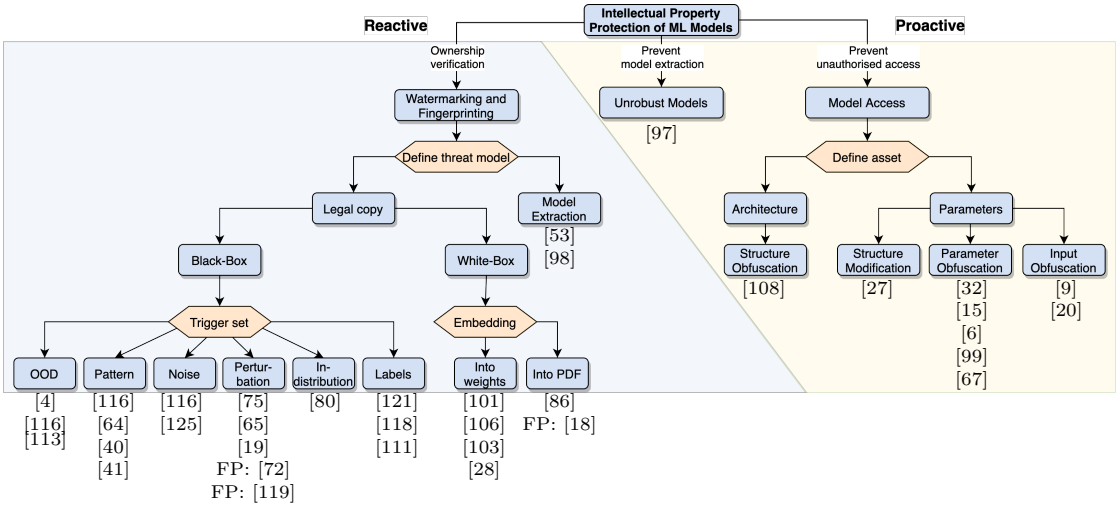


Figure 4.1: Taxonomy of Intellectual Property Protection mechanisms for Machine Learning models. Note, that not all considered papers are referenced in this diagram.

Regardless of how the attacker obtained the model, in both cases, the IP of the model owner is illegally utilised. For both cases, we will discuss methods to protect the IP of the model owner. It is important to differentiate between those two cases, as this has a large impact on selecting potential defence mechanisms.

4.2 IPP Methods

We developed a comprehensive taxonomy of IPP methods for ML models, depicted in Figure 4.1.

A principal categorisation of model watermarking methods is by *white-box* or *black-box* watermarking methods. White-box approaches embed the watermark in the model parameters or other model characteristics. With that in mind, the model owner would need the get the stolen copy from the attacker for the watermark verification process. This scenario seems unrealistic in most settings (e.g. an attacker offering an API service based on the model never discloses the model itself), and is the likely reason why black-box watermarking tends to be more popular, as the model owner can verify ownership with as little as a set of trigger inputs and the corresponding responses of the adversary model.

A further distinction is between (i) *reactive methods*, which try to react to a threat event, or (ii) *proactive methods*, which means the defender takes initiative, to prevent a threat event. Regarding the goal of the protection, we distinguish methods that enable to verify the ownership of a model, by *model watermarking* and *model fingerprinting*, and are thus *reactive*, and methods that, e.g., want to prevent unauthorised model access, and are thus *proactive*. Ownership verification is a weak form of protection, as it requires the unauthorised usage of the model to be known (or at least suspected), and further

requires some form of access to the model. Model access control, on the other hand, shall prevent such illegal use, by rendering the model useless to unauthorised users. This is comparable to preventing unauthorised use of, e.g., software.

Watermarking against the threat of a model extraction attack is mostly achieved by special black-box watermarking techniques that survive such an attack, i.e. the hidden information is "stolen" along with the model itself. In case a user initially obtained a legal copy of the ML model but is then using it in a way not according to the licensing terms, more techniques are available. White-box approaches for this case embed the ownership information directly into the model parameters, or in their probability density function (PDF). Black-box approaches mostly rely on specific input samples, so-called *trigger sets*, that will cause the model to behave in a way that is unexpected for the task, and unknown to the attacker. The techniques mainly differ in the way these triggers are constructed.

Model access control methods can be distinguished by the asset they want to protect. Most work focuses on the protection of the model parameters, either by encryption, other obfuscation techniques, or requiring a specific method to transform the inputs. If the model structure (or architecture) is to be protected, obfuscation techniques for those are employed.

4.3 Attack Model

In Section 5.5 we will introduce specific attacks against IPP mechanisms, namely *detection*, *overwriting*, *invalidation* and *removal*. Therefore, this section provides the attack models to which we will refer later on. Let us assume that the attacker obtains a legal copy of the target model, and either knows or assumes that the model has an IPP in place.

We consider the following cases as attack models for watermarking:

- **Watermark detection:** The attacker wants to detect if there is a watermark in the model, e.g. to then perform a targeted watermark removal or overwriting. If the watermark is not secured with an additional mechanism (e.g. a private key for extraction), the attacker could also claim ownership.
- **Watermark overwriting:** The attacker wants to overwrite an existing watermark by placing their own watermark and making the model owner's watermark useless.
- **Watermark invalidation:** The attacker wants to disable the watermark function, without actually removing it from the model, so that it cannot be verified.
- **Watermark removal:** The attacker wants to modify the model in a way such that the model owner's watermark extraction algorithm will no longer result in proving correct ownership, ownership of the real model owner.

Most of these attacks are also valid against *fingerprinting*.

Against **model access control** mechanisms, an attacker mostly would want to **remove** or **invalidate** (or potentially overwrite) the mechanism to gain unauthorised access to use the model (as black-box) or to reveal either the model architecture or model parameters for other purposes.

CHAPTER 5

State of the Art: Model Watermarking, Fingerprinting and Attacks

There are several connotations of watermarking, and generally, information hiding, along the machine learning process, as depicted in Figure 5.1. Sablayrolles et al. [89] propose a technique that *traces data usage*; it marks (training) data in a special way, so that a ML model trained on that data will bear a watermark that can be identified (cf. ① in Figure 5.1). The main body of work, and the focus of this section, consider ML models as the objects that need protection and in which the watermark is embedded (cf. ② in Figure 5.1). Abdelnabi et al. [3] propose a special form of watermarking. Their scheme is not watermarking a *model*, but the *output* of a text generating model (cf. ③ in Figure 5.1). They assume that an attacker could use the model for generating whole articles. In such a case, the watermark can be extracted from the generated text and prove the illegitimate usage of the model. In some settings, it is further considered that a marked output (prediction, or data) is generated with the explicit goal to trace the usage of this data, e.g. for training by an attacker (cf. ④ in Figure 5.1). This is a special form of ①, as the data origin is different, and of ②, as the adversary model is *implicitly marked* (cf. Section 5.3.7).

We want to point out that information hiding techniques for ML models may have other applications than watermarking. For instance, Song et al. [94] propose a technique to hide data from a private training dataset in the internals of a model (e.g. the weights) trained upon said dataset. This way, an attacker that does not have direct access to the training data, but can only let their model be trained on the data by the data owner, can exfiltrate this data via the derived machine learning model, i.e. perform a data exfiltration attack (cf. ⑤ in Figure 5.1). We, however, consider techniques which hide information about a lawful owner of the model.

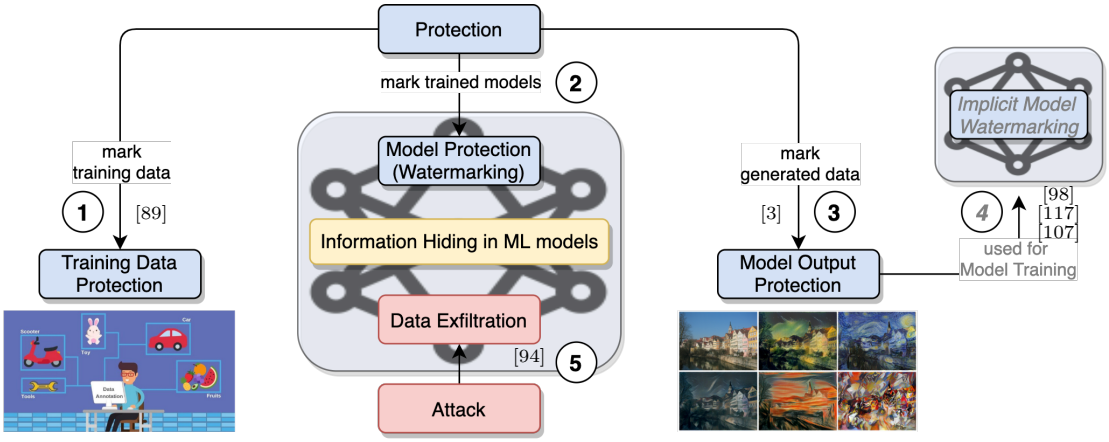


Figure 5.1: Different notions of information hiding along a ML process

The vast majority of watermarking methods for ML models is designed specifically for DNNs. The main reason for this is not only the high value of DNNs, as they require large datasets and long training time, but also the number of "degrees of freedom" in a DNN. Large DNNs thus have, compared to other ML models, more "space" for hiding marks. Most authors evaluate the schemes on image classification task. However, some authors extend the methods to other tasks, like audio classification [53], image captioning [66], image processing [85, 107, 117] (where the output is an image/data, rather than a prediction), or specific learning settings, such as GANs [93], Federated Learning with DNNs [10], Graph Neural Networks [120], and Deep Reinforcement Learning [11].

5.1 Requirements

A watermarking scheme should fulfil a couple of requirements. Literature is not coherent in the naming of these requirements of watermarking (and fingerprinting) methods, and we, therefore, aim at providing a common nomenclature. To this end, we collect all the requirements that were proposed in the papers included in our literature review and list them in Table 5.1, identifying also terms used as synonyms, along with references to the respective publications.

The most important requirements are **effectiveness**: the watermark shall be embedded in a way that the model owner can prove ownership anytime, **fidelity**: the model's accuracy shall not be degraded because of the watermark embedding, and **robustness**: the watermark embedding should be robust against several kinds of attacks, including fine-tuning, model compression and other, specifically crafted attacks. The remaining requirements are listed in Table 5.1. Note, that non-trivial ownership is sometimes used as a synonym for integrity, meaning that innocent models are not being accused of ownership piracy, but also as a requirement that an attacker cannot easily claim ownership without

Table 5.1: Requirements for Watermarking techniques. The notation is not consistent throughout the papers, but the terms in the left column are the most prominent ones. These requirements mostly apply also to Fingerprinting methods

Property	**Description**	**Other terms used in papers**
Effectiveness	The model owner should be able to prove ownership anytime and multiple times if needed	Authentication [64], Functionality [65]
Fidelity	The accuracy of the model should not be degraded after embedding the watermark	Funcionality-preserving [64, 106, 4], Loyalty [75], Utility [98] (Image WM: Transparency [84]; Relational Data WM: Usability [55])
Robustness	The embedded watermark should resist a designated class of transformations	Unremovability [4, 98]
Security	The watermark should be secure against brute-force or specifically crafted evasion attacks	Secrecy [93], Unforgeability [4, 106]
Legality	An adversary cannot produce a watermark for a model that was already watermarked by the model owner	Ownership piracy resilient [4, 106], Non-ownership piracy [98]
Integrity	The watermark verification process should have a negligible false positive rate	Low false positive rate [41, 40], Non-trivial ownership [64, 4, 106], Uniqueness [85]
Reliability	The watermark verification process should have a negligible false negative rate	Credibility [19]
Efficiency	The watermarking embedding and verification process should be fast	
Capacity	The watermarking scheme should be capable of embedding a large amount of information	Payload [40]

knowing the watermarking scheme and embedded watermark. Moreover, authentication is more a subset of effectiveness than a real synonym since it only requires that there is a provable association between an owner and their watermark. **Feasibility** is used as a combination of robustness and effectiveness [65], and **correctness** as a combination of effectiveness, reliability, and integrity [72]. Fingerprinting should fulfil two more requirements: **uniqueness** – the fingerprint can be uniquely identified with the user, and **scalability** – the fingerprinting scheme should be able to embed multiple fingerprints, either in the same model or in multiple versions of the model. A fingerprinting method that embeds multiple fingerprints, e.g. [18], could not only trace back the attacker but also identify collaboration between several malicious users.

We provide an overview of all the watermarking and fingerprinting schemes considered in this thesis, and whether they are meeting the above-mentioned requirements, in Table 5.2. We observe that all schemes fulfil the above-identified most important requirements of fidelity, effectiveness and integrity, except for [39], which on purpose gives up on

Table 5.2: Requirements met by watermarking and fingerprinting schemes. We distinguish two degrees: ∼ indicates: the respective authors claim the scheme fulfils this property; ✓ indicates: the authors show empirically that the property is fulfilled.

	white-box						**black-box**																								
Property	[86]	[18]	[101]	[106]	[103]	[28]	[16]	[116]	[4]	[64]	[40]	[41]	[125]	[75]	[65]	[19]	[119]	[72]	[80]	[98]	[53]	[121]	[118]	[111]	[113]	[39]	[93]	[66]	[107]	[117]	[85]
Effectiveness	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fidelity	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A	✓
Robustness	✓	✓	✓	✓	✓	✓	✓	✓	✓	∼	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
Security	✓	✓	∼	∼	∼		∼	✓	∼		✓		✓	∼	✓	✓						✓					✓		✓		
Legality				∼											✓				∼												
Integrity	✓	✓		✓		✓			✓	✓	✓	✓			✓	✓	✓	✓				✓									
Reliability	✓	✓					∼									✓	✓		✓												
Efficiency	✓	✓	∼		∼	✓	✓							∼		✓				✓		✓									
Capacity	✓		✓		✓	✓					✓					✓											✓	✓	✓		∼

robustness in favour of reversibility: the authors point out that the application of their scheme is not IPP, but integrity authentication, and that all existing watermarking methods are irreversible – once the watermark is embedded, it cannot be removed to restore the original model without degrading the model's performance. They argue that irreversible watermarking schemes are permanently modifying the model's internals, and thus destroying the integrity of the model, which could have severe consequences especially in applications for the medical or defence domain, etc. To make the scheme reversible they sacrifice the robustness requirement, inspired by traditional image integrity. For Zhang et al.'s method [117], the fidelity requirement does not apply since it is not well-defined for image processing. To determine for a model that outputs an image (or other complex data) whether a watermarked version of such a model is comparable to the original one, one would need to define a similarity measure to compare if the two outputs are equivalent.

Watermarking methods can be categorised into two main fields, *white-box* and *black-box* watermarking. White-box means that the model owner needs access to the stolen model's parameters or other model characteristics, in either step of the IPP method process, i.e. also during watermark extraction and verification. A black-box method generally only needs access to the model's prediction, e.g. via an API service, to observe matching input and output from the ML model, using it in a similar fashion as an *oracle*. In the following sections, we discuss both of the watermarking types.

5.2 White-box Watermarking

White-box watermarking requires full access to the model in order to verify the watermark. Usually, the model owner creates a T-bit signature vector $\mathbf{b} \in \{0,1\}^T$ that is a set of arbitrary binary strings that should be independently and identically distributed (iid) [86]. This binary vector serves as a watermark and is usually embedded into the model by fine-tuning with regularisation. We call this type of embedding scheme *regulariser based*. The general workflow for a regulariser based embedding scheme is illustrated in Figure 5.2a.

The first framework for embedding a watermark into a DNN was proposed by Uchida et al. [101][1] in 2017. They follow the idea of embedding a signature into the model, particularly in the DNN's weights. While it would be possible to directly alter the model's parameters, as it would be done for watermarking relational data, this would degrade the model's performance. They thus describe three ways of embedding the watermark: while training, while fine-tuning, or by using the distilling approach [48]. The fine-tuning approach is especially interesting when the model owner wants to place individual watermarks (i.e. fingerprints) before distributing to different users, in order to trace back the recipient in case of copyright infringement (cf. Section 5.4.1). The model is trained with a regulariser term, given the signature $\mathbf{b} \in \mathbb{R}^T$, the averaged weights vector $\mathbf{w} \in \mathbb{R}^M$ and a specially crafted *embedding matrix* $\mathbf{M} \in \mathbb{R}^{T \times M}$. The embedding matrix $\mathbf{M}$ can be considered a

[1] Slightly extended version in [79]

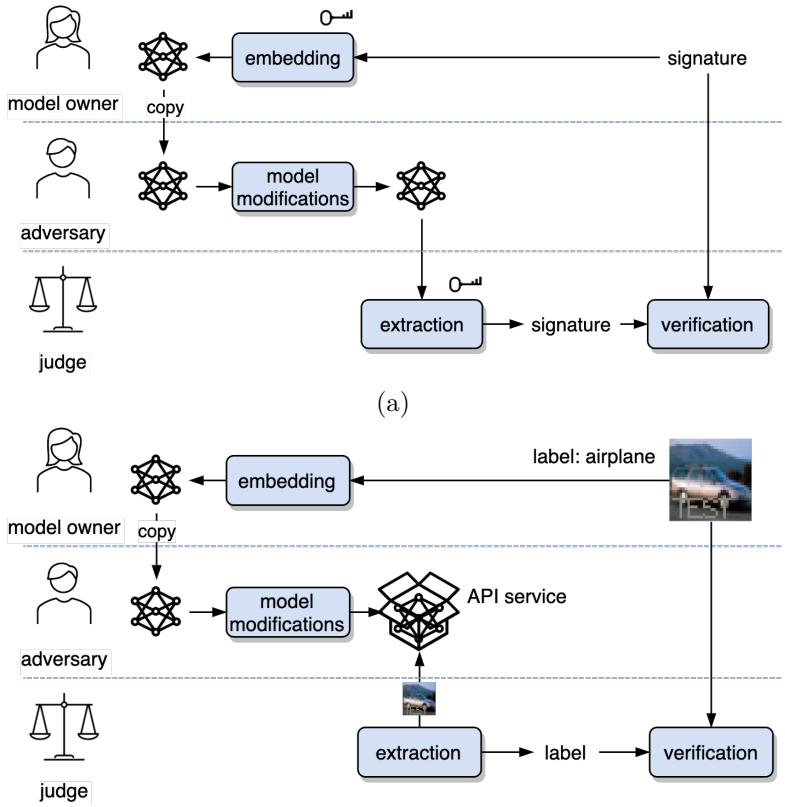


Figure 5.2: Typical workflows for (a) white-box watermarking and (b) black-box watermarking

secret key for the embedding and extracting process. The watermark is extracted by applying $\mathbf{M} \in \mathbb{R}^{T \times M}$ to the weights vector $\mathbf{w} \in \mathbb{R}^{M}$ and then applying a step function:

$$\tilde{\mathbf{b}}_j = s(\sum_{i=1}^{M} \mathbf{M}_{ji}\mathbf{w}_i), \tag{5.1}$$

where $s(\cdot)$ is the unit step function. The resulting vector $\tilde{\mathbf{b}}$ is then compared with the signature $\mathbf{b}$ and the BER (bit error rate) is computed. Ownership is proven by thresholding the BER.

Subsequently, Rouhani et al. [86] propose a watermarking framework that proves to be more robust against watermark removal, model modifications and watermark overwriting than [101]. The method is regulariser based and encodes the signature in the PDF of activation maps obtained at different DNN layers, by one additional regularisation term $\mathcal{L}_1(w)$ that ensures that selected activations are isolated from other activations, to avoid creating a detectable pattern of alterations and another $\mathcal{L}_2(w)$ to enforce that the distance between the owner-specific WM signature and the transformation of isolated

activations is minimal:

$$\mathcal{L}(w) = \mathcal{L}_0(w) + \lambda_1 \mathcal{L}_1(w) + \lambda_2 \mathcal{L}_2(w) \tag{5.2}$$

For extraction, they save a list consisting of the selected Gaussian classes, the trigger images and the projection matrix. In the verification process, the trigger images are used as input for the model to then analyse the activations. The scheme can be employed in a white-box or black-box setting, depending on whether just the output layer or also hidden layer activations are assumed to be available in case a watermark verification is needed.

Wang et al. [106][2] generalise both of the above presented algorithms into a white-box scheme. They show that the previous schemes are vulnerable to watermark detection (cf. Section 5.5), as the weight distribution deviated from those of non-watermarked models. The authors claim that this arises from the additive regularisation loss function(s). Therefore, they propose a new scheme that is particularly robust against detection attacks. Inspired by the training of GANs, they train a watermarked target DNN $\mathcal{F}_{tgt}$, which is competing against a detector DNN $\mathcal{F}_{det}$ that aims to discover whether a watermark is embedded.

Wang et al. [103] follow a similar approach and propose a white-box scheme for DNNs that makes use of an additional DNN for the watermark embedding process. The target model is trained in parallel with an embedding model, which will be kept secret after the embedding process. The scheme is regulariser based and the watermark is verified by feeding the selected weights into the embedding model and thresholding the output vector. They empirically show that their scheme achieves better fidelity, robustness and capacity compared to [101].

Feng et al. [28] combine a binarisation scheme and an accuracy compensation mechanism to reduce the model's accuracy degradation that results from fine-tuning. They use spread-spectrum modulation on the signature $\mathbf{b}$ and embedding it in different layers to reduce the risk of the watermarked weights being set to zero during a pruning attack. The binarisation scheme then transforms the selected weights per layer so that the overall energy, i.e. the second norm of the selected weights in one layer remains unchanged. The overall energy is defined as

$$||sw^j||_2 = \sqrt{\sum_{i=1}^{T} (sw_i^j)^2}, \tag{5.3}$$

where sw^j is the selected weights vector in the j-th layer of the selected layers. Therefore, the embedding position of the watermark cannot be discovered easily by an attacker.

As the last step, they use a "compensation mechanism" in fine-tuning, which aims to reduce the impact of watermark embedding on the model's performance. In particular,

[2] Newer and slightly changed version in [105]

they propose to keep the watermarked weights unchanged and fine-tune only all the other weights.

The first (and so far only) white-box framework for Automatic Speech Recognition (ASR) was introduced by Chen et al. [16], called *SpecMark*. The framework embeds the watermark in the spread spectrum of the ASR model without re-training it. They evaluate *SpecMark* on the DeepSpeech model and conclude that it does not have any impact on fidelity.

5.3 Black-box Watermarking

Black-box watermarking only needs access to the model outputs for watermark verification, which makes watermark verification far more practical. A typical workflow is shown in Figure 5.2b.

Only two of the existing black-box watermarking frameworks [53, 98] address the second threat model case (illegal copy) in Section 4.1. All the other watermarking methods are not reliably robust against model extraction attacks, and therefore primarily address the first case (legal copy).

All of the frameworks that are defending against the legal copy case utilise *(defensive) backdooring*. A *backdoor* consists of a so-called *trigger set* of input-output pairs, which are only known to the backdoor creator (in most cases, the model owner), and triggers a behaviour that is not predictable by others. We call the input images of the trigger set *trigger images*, sometimes *watermarks*, and the corresponding labels *trigger labels*.

Existing black-box watermarking methods concentrate on either creating suitable trigger images (inputs) or trigger labels. Depending on the scheme, different trigger images are used for watermarking:

- **Out-of-distribution (OOD)** trigger images are completely unrelated to the dataset, e.g. abstract images in a handwritten digit dataset.
- **In-distribution** trigger images are taken from the original training dataset and re-labelled wrongly.
- **Pattern based** trigger images originate from the training dataset but are marked with a pattern, e.g. logo, text or other designed pattern – comparable to patterns embedded in images for "conventional" data poisoning attacks (e.g. [38]).
- **Noise based** trigger images are images from the training dataset with added noise (i.e. no systematic pattern), either visible or invisible to the human eye.
- **Perturbation based** trigger images are slightly perturbed images and lie near the classification boundary, thus when re-labelled, they force the model to slightly shift its classification boundary.

(a) Out-of-distr. [4] (b) In-distr. [80] (c) Pattern [116] (d) Noise [116] (e) Perturb. [75]

Figure 5.3: Examples for the various types of trigger images, intentionally labelled as a different class ((a), (b) as "cat", (c), (d) as "airplane", (e) as "9")

Figure 5.3 shows examples for all these five types of trigger images. Similar to embedding backdoors as an attack to reduce the availability or integrity of a model, the main objective is that the model will accurately behave on the main classification task, but will fail the classification on the trigger images in the way the model owner has designated.

Zhang et al. [116] propose the first black-box watermarking scheme in 2018 and introduce three types of trigger images: *unrelated* (OOD), *content* (pattern based) and *noise* (based). Their work forms the basis for many following papers.

5.3.1 Out-of-distribution

Similar to and shortly after Zhang et al. [116], Adi et al. [4] propose to include abstract images as trigger images in the training dataset. Those abstract images are completely unrelated to the main classification task, thus it is highly unlikely that a model that has not seen this data point (i.e. one not watermarked) will label it as the designated class.

One of the first watermarking schemes for image processing models was proposed by Quan et al. [85]. The main difference to classification is that the output is, like the input, an image and not a label – thus they are generating input-output pairs which consist of trigger images and verification images. They propose to use OOD images (or random noise) as trigger images and create the verification images by applying a simple image processing method to the trigger images (ideally not the one trained by the model). The idea is the same as in (defensive) backdooring. They generate input-output pairs where each consists of a trigger image and a verification image. The model is then fine-tuned on the union of the training dataset and the trigger set.

Yang et al. [113] empirically show that distillation is an effective watermark removal attack. Therefore, they propose a new watermarking scheme, called *ingrain*, that the authors claim to be especially robust against distillation. The main idea is that the watermark information is carried by the predictions of the original training data but the watermark extraction is done by querying trigger images that are drawn from a different distribution than the original images. Comparing to [4] and [116], the target model is not trained on the union of the original training dataset and the trigger set, but only on the original dataset, and instead makes use of another model, the *ingrainer model*, which influences the target model by a regulariser term to the loss function. The ingrainer

model has the same architecture as the target model and is only trained on the trigger set, in particular, to overfit the trigger set.

5.3.2 Pattern

An improved pattern based technique was proposed by Li et al. [64]. They show that previous schemes [4, 116] are vulnerable to *ownership piracy* attacks, in which an attacker aims to embed his own watermark into an already watermarked model. They propose a watermarking scheme that is especially robust against such attacks using *dual embedding*: the model is trained to classify (i) data with a pre-defined binary pattern correctly (*null embedding*), and (ii) data with an inverted pattern (binary bits are switched) incorrectly (*true embedding*). In more detail, for a pre-defined binary valued pattern p and a very large number λ, the pattern p is dual embedded into the model $\mathcal{F}_w$ if for all $X \in \mathbb{R}^n$ holds

$$\mathcal{F}_w(\text{apply}(X, p, \lambda)) = \mathcal{F}_w(X) = y, \tag{5.4}$$

$$\mathcal{F}_w(\text{apply}(X, \text{inv}(p), \lambda)) = \hat{y} \neq y, \tag{5.5}$$

where $\text{apply}(X, p, \lambda)$ applies the pattern p with magnitude λ (value λ for bit 1 and value $-\lambda$ for bit 0) to the image X. Equation (5.4) refers to null embedding and Equation (5.5) to true embedding. They observe that null embedding does not degrade the model's accuracy if the number of pixels in the pattern is sufficiently small. Furthermore, they evaluate the robustness against model extraction attacks and conclude that, with enough (at least the same amount of) in-distribution data, the attacker is able to make a copy of the model without the watermark. For out-of-distribution data, the attacker would need 12.75 times more input data to reach similar accuracy.

Guo et al. [40] propose to embed a pattern into the trigger images that can be clearly associated with the model owner's signature, e.g. a logo. The pattern should be embedded with little visibility so that the original model would still classify the trigger images to their original labels. The signature is used as a key to determine the pattern and then embedded in the image.

Guo et al. [41] propose an evolutionary algorithm-based method to generate and position trigger patterns, based on [116] and [40]. Their algorithm is based on Differential Evolution [95], an evolutionary algorithm and a metaheuristic that searches for the optimal solution for an optimisation problem. Using this trigger pattern generation they demonstrated an improvement in integrity and robustness.

5.3.3 Noise

Zhu et al. [125] propose a watermarking scheme especially to defend against watermark overwriting (forging attacks). They propose to use one-way hash functions for both generating the trigger image and label. The framework takes an initial image and creates a hash chain of trigger images, as shown in Figure 5.4. They show experimentally that

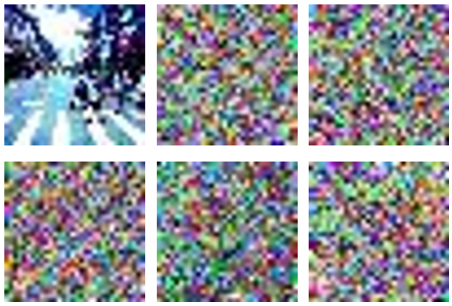
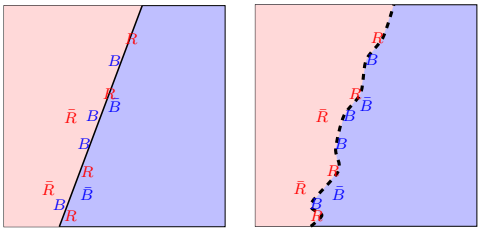


Figure 5.4: The upper left image is the initial image and the following five are trigger images resulting from a hash chain [125].



(a) (b)

Figure 5.5: (a) The data points are divided into "true adversaries" (R and B) and "false adversaries" ($\bar{R}$ and $\bar{B}$). The label for the true adversaries is changed, the label for the false adversaries stays unchanged. (b) After fine-tuning the decision boundary changes. [75]

their proposed scheme is robust against forging attack even if the attacker knows the trigger set generation algorithm.

5.3.4 Perturbation

Another black-box scheme was proposed by Merrer et al. [75]. The goal is to slightly shift the decision boundary of the model. This is achieved by generating adversarial examples [96] for images close to the boundary, and changing the class for those adversaries. After fine-tuning the model, the decision boundary is adapted. An illustration of this decision boundary shifting for a two class boundary is given in Figure 5.5.

Li et al. [65] especially address evasion attacks. They propose a framework closely related to the idea of GANs. They use three DNNs: encoder, discriminator, and target model. The encoder takes the original image and aims to embed a logo into the image such that the difference is imperceptible. The resulting trigger images are fed into the discriminator together with the original image to evaluate the encoder's success. A difference in the original and trigger images is essential for the watermarking scheme, yet the goal is to make the difference as small as possible. This framework especially deals with the trade-off between security and effectiveness. The smaller the difference, the better security against evasion attacks, and the larger, the better the effectiveness of

the embedded watermark.

Chen et al. [19] propose the watermarking framework *BlackMarks*, which encodes the signature within the distribution of the output activations. To encode the class predictions, the authors design a scheme that maps the class predictions to binary bits, by clustering the original classes into two categories, represented by bit 0 and bit 1. The trigger images are created in a way that for the bit 0, an adversarial example that would belong to the cluster represented by the bit 1 is created, and is labelled with a uniformly randomly chosen class from the cluster represented by the bit 0. Trigger images for the bit 1 are created vice-versa. The watermark is extracted by querying the trigger images and encoding each class to a binary value, which should result in the owner's binary signature to prove ownership.

5.3.5 In-distribution

Namba et al. [80] propose an attack, called *query modification*, that investigates the query for trigger images in order to invalidate the watermark (cf. Section 5.5). With that in mind, they propose a scheme that is more robust especially against query modification but also model modifications like fine-tuning and model compression (e.g. pruning). The query modification as an attack exploits the fact that trigger images differ from original training images. Therefore, they propose to use trigger images that are selected from the training sample distribution. The trigger images are thus undetectable, however, the model is more likely to overfit to the (on purpose) wrongly labelled triggers, and thus more susceptible to removal attacks via e.g. pruning. They want to counter pruning by ensuring that the predictions do not depend on a large number of small model parameters that would likely be pruned. Thus, the model is first trained as usual with the original training set. Then, the watermark is embedded by exponentially weighting the parameters and training the model on the union of the training dataset and the trigger set, which enforces the predictions to depend on a small number of large parameters instead. In exponential weighting the model parameters $\mathbf{w}^l$ are changed in every layer l by

$$\mathbf{w}^l_{\text{exp},i} = \frac{\lambda \exp |\mathbf{w}^l_i|}{\max_i \lambda \exp |\mathbf{w}^l_i|} \mathbf{w}^l_i, \tag{5.6}$$

where $\mathbf{w}^l_i$ denotes the i-th component of the parameter vector $\mathbf{w}^l$ in layer l and λ is an adjustable parameter for the intensity of the weighting.

5.3.6 Trigger labelling

Zhong et al. [121] propose to label the trigger images with a completely new label, rather than assigning one from the existing labels, so that the watermark embedding has only little impact on the original classification boundaries. Any pattern based trigger image can be used in this context. They compare their work to [116] in their experimental evaluation and show that the proposed scheme achieves a zero false-positive rate, i.e. excellent integrity, and is more robust against fine-tuning and model compression.

Zhang et al. [118] observe that frequently, trigger images are created in a systematic way, and it is thus easier for an attacker to re-create them. Therefore, they propose to include unpredictability in the labels assigned to the trigger images. They use a chaos-based labelling scheme for trigger images that ensures that an attacker cannot produce a valid trigger set, even if he knows the trigger pattern.

Xu et al. [111][3] propose, similar to *BlackMarks* [19], a watermarking scheme that carries the watermark information within the output activations. The trigger pairs consist of a trigger image and a serial number (SN) which is constructed individually by the owner's rules. The SN is composed by using the probabilities in the last layer. It is worth noting that any rule for generating it is feasible. For instance, for three classes the SN could be "235" following the rule to reduce the digits by one order, resulting in [0.2, 0.3, 0.5], but also any other, more complex rule can be used. The proposed scheme does not rely on the specific choice of trigger images since it focuses on the output activations.

5.3.7 Countering Model Extraction

Only a few schemes address the second threat model case in Section 4.1, i.e. robustness against model extraction attacks. Jia et al. [53] are the first to propose a scheme, called *entangled watermark embedding* (EWE). The main idea is to create a watermarked model that is not specialised into "sub-models", where one part of the model is capable of the main classification task, and the other for watermark detection (which normally is lost during the model extraction attack). This is achieved by a regulariser term ensuring that the trigger images lead to similar activation patterns as the original images. Thus, both trigger images and original images cause a similar behaviour of the model, thereby increasing the robustness against model extraction.

Szyller et al. [98] propose the framework *Dynamic Adversarial Watermarking of Neural Networks* (DAWN), which is not embedding a signature into the target model itself but dynamically returning wrong classes from the API service for a fraction of queries to mark an adversary model created via a model extraction attack. It is worth noting that the scheme is not able to differentiate between an attacker and a benign client – all clients obtain a fraction of wrong predictions. It is ensured that the same query always returns the same output (correct or modified). Whenever a wrong prediction is returned, the input-output pair is saved so that in need of a watermark verification, they can be used for triggering the stolen model. This approach thus realises ④ in Figure 5.1.

Zhang et al. [117] and Wu et al. [107] propose, independently of each other, a similar approach to [98], as they are hiding an invisible watermark in the outputs of the image processing model, but for *all* outputs. When an attacker trains a new (surrogate) model on the input-output pairs of the original model, the watermark will be learned too and can be verified with black-box access (cf. ④ in Figure 5.1). The major difference to [98] is that in the case of an image processing model, the output is another image, and thus

[3]Previous pre-print version in [110]

there is more data to embed the watermark in. Both papers are not explicitly addressing model extraction attacks, but we believe they are a suitable defence.

5.3.8 Watermarking for specific ML settings

Existing watermarking schemes are not suitable for Federated Learning (FL), as pointed out by Atli et al. [10]. Embedding a watermark in such a setting is different, because the model owner has no access to training data, and the training is performed in parallel by several clients. Atli et al. propose to include an independent and trusted party between the model owner and the clients. The independent party has the ability to embed a black-box watermark based on backdoors into the model at every aggregation step. Furthermore, they propose a watermarking pattern that is a specific noise pattern.

Skripniuk et al. [93] propose the first, and until now only, watermarking scheme specially crafted for GANs. The existing watermarking schemes are limited to DNNs that map from images to classes, and thus could not be transferred to GANs. The authors watermark the input images and then transfer the watermarked images to the GAN model. First, the image steganography system has to be trained, which consists of an encoder and decoder, and subsequently, *all* the training data together with a secret watermark is fed to the encoder resulting in watermarked data. The watermarked data is then used to train the GAN model. For verification, the model owner only needs an output image of the GAN, and applies the decoder on it to compare the result with the secret watermark. Thus, the proposed scheme needs only black-box access for verification.

Lim et al. [66] are the first to propose watermarking for a recurrent neural network (RNN). Specifically, they consider an image captioning model, implemented as a simplified variant of the *Show, Attend and Tell* model [109], where the output of the model is a sequence (the caption). The proposed framework is similar to [27], but does not embed the verification information (the owner's key) into the model weights, but into the signs of the hidden states of the RNN. During model inference time, the key is required as input to the model by an element-wise combination with the input data. Without the correct key provided, the effectiveness of the model drops significantly.

5.4 Fingerprinting of ML Models

Imagine a software company is selling their ML model to different customers, but the model got illegally redistributed by one of them. The company would like to gather evidence on the leak and therefore could embed fingerprints in the ML model before selling the product, in order to trace back the user when needed. We can think of fingerprinting as an extension of watermarking on a user level. At the time of performing this survey, fingerprinting for ML models was not extensively discussed, with only three papers published.

Note that there is another notion of fingerprinting. In cyber security, a (unique) identifier for an object (either hardware, software, or a combination thereof) is generally referred

to as a fingerprint, e.g. such as in browser fingerprinting [26] or device fingerprinting [59]. The application scenario for employing these techniques is often in tracking devices (resp. their users). Because this context differs from what we considered so far, we call this form *Fingerprinting as unique identification*.

5.4.1 Fingerprinting as User-specific Watermark

Chen et al. [18] propose a white-box fingerprinting framework, *DeepMarks*, that is able to embed unique fingerprints. The verification process not only detects the attacker, but detects also if multiple, and if so which, users collaborated in order to create an unmarked model. The embedding process works similar to the one in [86]. They propose to assign a unique binary vector (fingerprint) to each user and embed the fingerprint information in the PDF of the weights before distributing the models to the users.

Although *DeepMarks* is the only paper considering especially fingerprinting, we believe that a couple of the above introduced watermarking schemes can be extended to fingerprinting. To name a few, [101] embeds a unique signature into the weights of the DNN, [65] embeds a unique logo into the trigger images and [40] generates unique trigger images based on a signature. All of them could embed user-specific watermarks. Moreover, [111] relies on serial numbers that can be created in indefinitely many ways, assigning each to a user.

5.4.2 Fingerprinting as Unique Model Identifier

Cao et al. [14] propose a framework to obtain a unique identifier of DNNs. Based on the idea that two different models have different classification boundaries, they propose to "fingerprint" the classification boundary of the model. If the model was changed in the slightest way, the classification boundary would slightly shift too. This scheme, however, does not fit our threat model, as we want to recognise the original model even after (especially minute) model modifications. We, therefore, require a fingerprinting framework to be more robust to model modifications.

Zhao et al. [119] and Lukas et al. [72] modify this idea of fingerprinting as unique identification. Both propose a scheme in which the adversary model, created by applying modifications to the target model, has the same fingerprint as the target model. They both introduce a novel algorithm for creating transferable adversarial examples [70].

Note that in Section 5.3 we describe how black-box watermarking methods use perturbation based trigger images (adversarial examples), which are used during training so that the models learn how to (purposefully) misclassify them. In the context of fingerprinting as a unique model identifier, the authors want to create an adversarial example from an already trained ML model. The key aspect is that the generated images are not only adversarial examples for the target model, but also for the adversary model, i.e. they are transferable to the adversary model. This fits our first threat model case in Section 4.1.

5.5 Attacks on Watermarking Methods

If the attacker knows or suspects that a model is protected, he could try to change the model in order to remove or overwrite the protection. Regarding watermarking, most authors claim that their techniques are robust against various model modifications like *fine-tuning*: re-training the model with new data and *parameter pruning*: setting small parameter values to zero, which is a model compression technique [43, 124]. But further attacks were proposed that are aiming to remove, overwrite, detect or invalidate state-of-the-art watermarking schemes, which we will analyse. We want to point out that most of the techniques, except of *EWE* [53] and *DAWN* [98], are not robust against model extraction (model stealing) attacks [100], as pointed out by Mosafi et al. [78].

In Table 5.3, we summarise the attacks on watermarking schemes. Each line corresponds to an attack and each column to a (type of) watermarking scheme. The table shows which attack defeats which kind of watermarking. We list only schemes that were proven to be successfully defeated – missing schemes in the table do not imply strong robustness. We can see that an attack usually addresses either white-box or black-box watermarking schemes. The four trigger image types OOD, pattern, noise and perturbation based seem to be defeated in a similar way. In-distribution watermarks are more difficult to detect or remove, probably because of the fact that they do not differ from the original training data distribution.

5.5.1 Watermark Overwriting

Li et al. [64] show that previous schemes [4, 116] are vulnerable to watermark overwriting (ownership piracy attack). They apply the schemes to four image classification tasks, and assume that an attacker would have access to around 10% of the original training images. Experimentally, the authors conclude that an attacker could successfully embed the pirate watermark by training the model with training data that is adapted to the pirate watermark.

5.5.2 Watermark Detection

Most black-box watermarking methods are based on backdoors. Therefore, backdoor detection algorithms like Neural Cleanse [102] and Fine-Pruning [68] could be a potential threat to these methods. Wang et al. [106] show that regulariser based watermarking schemes are vulnerable to watermark detection by the use of a property inference attack [30]. Knowing the embedding algorithm, they train a set of shadow models (i.e. models with similar architecture and similar data), some of which will be watermarked, and some not. From these models, they extract weights as representative features, and subsequently, train a model on these features to distinguish between watermarked and not-watermarked models.

Table 5.3: Which attack defeats which watermarking technique based on the evaluation of the papers. A $\sim$ denotes that the authors claim that their attack can be extended easily to defeat this watermarking technique but did not provide an evaluation for that.

			Watermarking techniques					
			OOD	pattern	noise	perturbation	in-distribution	regulariser based
Attacks on watermarks	invalidation	Hitaj et al. [49]	[4], [116]$\sim$			[75]$\sim$		
		Namba et al. [80]	[116]	[116]	[116]	[75]		[86]
	overwriting	Li et al. [64]	[4], [116]	[116]	[116]			
	detection	Wang et al. [106]						[101], [86]
	detection, removal	Wang et al. [104]						[101]
		Shafieinejad et al. [90]	[4], [116]	[40], [116]	[116]			
	removal	Liu et al. [69]	[4], [116]	[40], [116]	[116]			
		Aiken et al. [5]	[4], [116]	[116]	[116]			
		Guo et al. [42]	[4]	[116]		[75]		
		Chen et al. [21]	[4], [116]	[116]		[75]	[80]	
		Yang et al. [113]						[101], [86], [18]

5.5.3 Watermark Removal

Some of the attacks exploit the fact that a watermarked model actually learns two tasks: the main classification task and the watermarking task. This fact can be also observed in the increase of the model parameters' variance during watermark embedding [104].

Wang et al. [104] are one of the first to reveal serious vulnerabilities in watermarking schemes, in particular watermark detection and removal. They observe that in regulariser based watermarking methods like [101], the variance of the distribution of model parameters, they call this *weights variance* or *weights standard deviation*, increases during watermark embedding. The watermark is removed by embedding one or several additional watermarks into the model, following the embedding scheme in [101]. Since every additional watermark might increase the weights variance, they propose to lower the weights variance by adding a L_2 regulariser. Following this procedure, the authors show that the old watermark cannot be extracted, thus the model owner cannot claim ownership anymore. It shall be noted that although additional watermarks are placed into the watermarked model, the main objective is to "neutralise" the old watermark rather than to use the new watermarks to claim ownership.

Shafieinejad et al. [90] analyse the robustness of backdoor based watermarking schemes. In particular, they propose a model extraction (stealing) attack that trains a substitute model (cf. e.g. [82]). This is performed by querying the original model with a public dataset from the same domain and using the resulting label to train their own model. As the public dataset contains none of the trigger images, the watermark is "lost" in the process. Furthermore, similar to Wang et al. [106], they propose to use property inference for watermark detection.

Liu et al. [69] propose *WILD*, a framework against backdoor-based watermark techniques based on fine-tuning. They argue that it is hard for adversaries to collect a large amount of unlabelled data within the same domain as the original training data, as required by [90], and that using non-domain data impacts the effectiveness of the substitute model. Their special contribution is that only a small portion of training data is needed. They utilise Random Erasing [123], i.e. removing random segments from the input images, to augment the data. This augmented data alone is, however, not enough to remove a watermark via fine-tuning, because of the high diversity of potential watermarks. They note that backdoor patterns are mostly learned by the high-level feature spaces produced by the convolutional layers, and not by the fully connected layers. They therefore utilise the augmented data to fine-tune the model and add a regulariser (penalty) term that ensures a minimal distance in distribution between the high-level feature space of the augmented and the clean dataset, so that a backdoor pattern would not be learned. The authors reveal that it is more difficult to remove OOD, compared to pattern based and noise based watermarks.

Guo et al.'s removal attack [42] has two aspects: (i) input data pre-processing consists of pixel-level alterations such as embedding imperceptible patterns and spatial-level transformation such as affine and elastic transformation, aiming at making the trigger

image unrecognisable by the model, i.e. not predicting the designated label; (ii) fine-tuning with data that can be unlabelled and from a different. This step aims at restoring the accuracy of the model on normal samples, which might suffer from the input data pre-processing. Using the watermarked model as an oracle to obtain labels, these input samples are then pre-processed in the same manner and used for fine-tuning the model. The authors empirically show that their watermark removal attack can remove various types of watermarks without knowledge about the watermark embedding and labelled training samples.

Chen et al. [21][4] propose *REFIT*, a watermark removal framework based on fine-tuning. The basis of their work is the *catastrophic forgetting* phenomenon of ML models [35], which shows that models that are trained on a series of tasks could easily forget the previously learned tasks. Their attack model assumes that the attacker has neither knowledge of the watermark nor the watermarking scheme, and has limited data for fine-tuning. They first show that in case the training data is known, the watermark can be removed by fine-tuning, when choosing the learning rate appropriately. To adapt to having only limited data that do not come from the original dataset, they include two techniques: elastic weight consolidation (EWC) and augmentation with unlabelled data (AU). EWC slows down the learning of parameters that are important for previously trained tasks, in particular the main classification task, by adding a regulariser term to the loss function. AU increases the number of in-distribution labelled fine-tuning data. Unlabelled data is retrieved from the internet and labelled by the pre-trained model. In most cases, the model labels the data by their true classes based on the training for the main classification task, since the model has not seen the data before and the watermarked model was trained in a way so that it fulfils the integrity requirement. The authors show that the proposed framework successfully removes the watermark without degrading the model's test accuracy from various state-of-the-art watermarking schemes, as shown in Table 5.3.

Aiken et al. [5] propose a method for watermark removal based on previously proposed backdoor removal attacks [102, 68], assuming an attacker with a small amount (less than 1%) of original training data. Their technique involves three steps: (i) they first reconstruct the perturbations (backdoor patterns) that are required to flip a sample to the other class, using the method from [102]. (ii) they then superimpose the pattern on their clean training data, to identify neurons that are responsible for recognising the backdoored images, similar to [68]. These neurons are then reset by setting their incoming weight so that they produce zero activation. (iii) Finally, the model is fine-tuned on the clean and backdoored training data, while labelling the backdoored training data to the class that is least likely to be watermarked, to prevent the re-appearance of the neurons reset in the previous step. They show that their technique defeats the watermarking schemes [116] and [4] by effectively removing neurons or channels in the DNN's layers that contribute to the classification of trigger images.

[4] Previous version in [22]

5.5.4 Watermark Invalidation

Watermark invalidation does not aim to remove the watermark, but find a way to render it useless.

Hitaj et al. [49][5] propose two such attacks: an *ensemble attack* and a *detector attack*. The ensemble attack uses several different models behind an API, obtained from, e.g., Model Zoo [2], querying all models and choosing the output that was given by most of the models. If one of the models is watermarked and triggered with a specific input for the watermark extraction process, then most likely only the watermarked model will output the expected label, while the remaining models will classify correctly. Therefore, the trigger output will not be returned, and the verification fails. The detection attack tries to avoid a trigger response; it trains a neural network, the *detector*, that predicts if the query is intending to trigger a watermark. If the input is recognised as a trigger image, either a random class can be returned or no class at all. The detector is a binary classifier that needs to distinguish between *clean* and *trigger* input. Clean input is collected from other datasets or by web scraping. Trigger input is generated from a portion of samples gathered from web. It shall be noted that this kind of attack is not able to invalidate *pattern based*, *noise based* and *in-distribution* watermarks, as the detector cannot be trained well for watermark detection without further information about the watermark.

Namba et al. [80] propose a watermark invalidation attack called *query modification processing*, consisting of two steps: trigger sample detection and query modification via autoencoder (AE). An autoencoder can reduce the effect of trigger images by diluting the pattern embedded in the original image or eliminating the embedded noise. Because the application of an autoencoder to non-trigger images impacts negatively the performance of the model, it is not recommended to use the AE on every query. Similar to [49] they propose to first detect if the input could be a trigger image queried during a watermark verification process. They propose three ways to perform the detection: measuring the effect of the autoencoder on the image in the input space, measuring the effect in the output space, or both. The proposed method has been demonstrated on various watermarking schemes and proved to successfully invalidate the watermarks.

5.6 Surveys and empirical studies

As the first work in this field, an empirical study by Chen et al. [17] investigates five model watermarking schemes [101, 86, 75, 4, 116], and performs an evaluation of fidelity of the models, as well as estimating the robustness against three attacks (model fine-tuning, parameter pruning, and watermark overwriting), thus providing an important early comparison of the effectiveness of techniques. We discuss the differences to our work in Chapter 6. A survey specifically on watermarking machine learning models was published as a pre-print by Boenisch [12].

[5] Previous pre-print version in [50]

CHAPTER 6

Defining research questions and study setting

The experiments in the following chapter form the basis for our analysis of the watermarking methods, with which we want to answer the following research questions.

How can we define the most fitting watermarking method depending on the ML setting?

Research on watermarking ML models is growing and so is the number of papers proposing new watermarking methods. Some of the papers do a comprehensive evaluation of their watermarking method, testing it on different datasets and architectures but rarely compare it to all existing methods. Moreover, usually authors pick the best results for their paper without pointing out the weaknesses of the presented method. With the independent implementation and evaluation in this thesis, we want to find potential influences of a ML setting on the **effectiveness**, **fidelity** and **robustness** of a watermarking method. We answer this question by designing a study setting, in which we are going to measure the **effectiveness**, **fidelity** and **robustness** by computing the watermark accuracy (for effectiveness) and test accuracy (for fidelity) after embedding the watermark, and computing the watermark accuracy after an attack (for robustness).

We break down this question into three specific subquestions:

1. **To what extend is a more complex model able to hold more watermark information (a bigger trigger set) without compromising test accuracy?** A more complex model has more parameters and therefore more "space" to hide a watermark. On the other hand, a model with fewer parameters might rather give up on the main classification task in order to overfit on the trigger set. We want to answer this question by performing experiments with various trigger set sizes and

architectures and measuring the difference in test accuracy after the watermark embedding.

2. **To what extend does the trigger set size influence the effectiveness, fidelity and robustness of a watermarking method?** A bigger trigger set size could mean better robustness, since an attacker would need more time or data to, e.g., remove the watermark by fine-tuning. On the other hand, a bigger trigger set could also result in worse fidelity. For effectiveness, a smaller trigger set size, could mean that the model does not learn the watermark at all, as the ratio compared to the original dataset is too low and it prioritises on the original dataset. These are hypotheses that we want to follow under this research question.

3. **To what extend does the complexity of the model influence the effectiveness, fidelity and robustness of the watermarking method?** Similar to the question above, we assume that the complexity of a model plays a role on the effectiveness, fidelity and robustness of a watermarking method. With our experiments, we want to find out if our assumption is true.

We choose our subset of watermarking methods according to the following criteria. The chosen watermarking method must

- be a black-box and backdoor-based watermarking method, as these are more practical than white-box methods.
- focus on trigger set generation rather than trigger labelling, as trigger images are the first step to analyse and optimise. An analysis of watermarking methods regarding trigger labelling are kept for future work.
- be a watermarking method that does not build upon another watermarking method, as in a first step we want to identify the aspects which make the basic methods perform better.

Following these criteria, we decided on the watermarking methods that are listed in Table 6.1. This table includes also the experimental setup in the corresponding papers. Our choice of the datasets and architectures is very much inspired by the choice in the papers.

A work by Chen et al. [17] compares 5 watermarking methods [101, 86, 116, 4, 75], both white-box and black-box watermarking methods. It is worth noting that this is not an independent comparison, since the authors are also the authors of one of the considered watermarking methods, DeepSigns [86], and it performs best in most of the results. DeepSigns can be implemented as both white-box and black-box (cf. Chapter 5) and therefore they are comparing it to one white-box and three black-box methods. Even though the method *can* be considered as black-box, it is not backdoor-based and relies on the prediction vector for watermark verification, which might not be available in all

Table 6.1: Study settings in selected papers.

Method	Type	Dataset	Architecture
ExponentialWeighting [80]	in-distribution	MNIST, CIFAR-10, CIFAR-100, GTSRB	ResNet32
FrontierStitching [75]	perturbation	MNIST	MLP [1], CNN [2], IRNN [3]
PiracyResistant [64]	pattern	MNIST, CIFAR-10, GTSRB, Youtube Faces	custom DNN
ProtectingIP [116]	pattern, noise, OOD	MNIST, CIFAR-10	custom DNN
WeaknessIntoStrength [4]	OOD	CIFAR-10, CIFAR-100, ImageNet	ResNet-18
WMEmbeddedSystems [41]	pattern	MNIST, CIFAR-10	LeNet-5, VGG-16, ResNet50, DenseNet-121

[1] `https://keras.rstudio.com/articles/examples/mnist_mlp.html`
[2] `https://keras.rstudio.com/articles/examples/mnist_cnn.html`
[3] `https://keras.rstudio.com/articles/examples/mnist_irnn.html`

settings. They implement the methods and train the black-box watermarked models with a trigger set size of 20 on four different architectures, and the white-box watermarked models on three different architectures. Both watermarking types are trained on MNIST and CIFAR-10, the black-box type also on ImageNet. They evaluate the models regarding fidelity, robustness against fine-tuning, parameter pruning and watermark overwriting, and integrity, i.e. the watermarking method should have a minimal false positive rate. Although the comparison lacks on different settings and independence, the evaluation of fidelity and robustness is done in a similar fashion to ours. However, they do not present results on effectiveness, i.e. the watermark accuracy of the watermarked model.

In order to answer the research questions, we have to evaluate the watermarking methods in a common and independent study setting. There are some parameters that we would like to fix to specific values, and others that we would like to vary in order to draw conclusions. We modify the following parameters, as these are the usual parameters that are modified by the research papers. We, however, vary also the size of trigger set, which is rarely done by other authors.

- **Complexity of architecture**: we choose eight state-of-the-art architectures, inspired by the experimental setup in the proposed papers (cf. Table 6.1), i.e. SimpleNet, LeNet-1/3/5 for MNIST and DenseNet, ResNet-18/34/50 for CIFAR-10.
- **Size of training set and images**: a variation in the size of training set and examples is unfortunately in this stage of the work not possible. We have chosen two datasets which both have a similar size of training set and a similar (small) size of images.
- **Color depth**: we choose one RGB and one greyscale dataset, i.e. CIFAR-10 and MNIST.
- **Size of trigger set:** we vary between three trigger set sizes, i.e. 0.04%, 0.2% and 1% of the training set.
- **Embedding type**: we use two embedding types. Embedding from scratch – training the model on the union of the training dataset and the trigger set from the very beginning of training; and embedding on a pre-trained model – embedding the watermark as a fine-tuning step after training the model only on the original data.
- **Complexity of attacks**: we choose two common attacks, i.e. parameter pruning and fine-tuning, to test the robustness, they both differ in overhead.

In the following sections, we present the datasets and neural networks that we use for the experiments.

6.1 Datasets

We use four different datasets: two datasets (MNIST and CIFAR-10) for training the models, and another two datasets (EMNIST and CINIC-10) for carrying out the fine-tuning attacks:

- **MNIST [63]:** is a grayscale handwritten digits dataset consisting of 60,000 training examples and 10,000 test examples. All images have been size-normalized and centered in a fixed-size image of 28×28 pixels.
- **CIFAR-10 [60]:** is a real-world image dataset containing the classes airplane, car, bird, cat, deer, dog, frog, horse, ship, truck. The dataset consists of 50,000 training examples and 10,000 test examples, all in colour (RGB) and sized 32×32 pixels.
- **EMNIST [24]:** is another grayscale handwritten digits dataset. It is derived from the NIST Special Database 19 [37] and converted to fixed-size images of 28×28 pixels. The EMNIST dataset consists of six different splits, namely ByClass, ByMerge, Balanced, Letters, Digits and MNIST. We use the Digits split, which

Dataset	Size	Color	Training set	Test set
MNIST	28×28	grayscale	60,000	10,000
EMNIST Digits	28×28	grayscale	240,000	40,000
CIFAR-10	32×32	RGB	50,000	10,000
CINIC-10	32×32	RGB	90,000	90,000

Table 6.2: Characteristics of the datasets used in the evaluation

Figure 6.1: Representative examples for each class from the training sets of MNIST, CIFAR-10, EMNIST and CINIC-10.

contains of 280,000 example-label pairs. The EMNIST dataset structure matches directly with MNIST, which makes it convenient to use for our experiments. We use this dataset for fine-tuning attacks on models that were originally trained on MNIST.

- **CINIC-10 [25]:** is another real-world image dataset that extends CIFAR-10 by adding additionally selected images from ImageNet [88], resized to 32×32 pixels to match the ones from CIFAR-10. images. It is 4.5 times larger than CIFAR-10, but still smaller (and comprising fewer classes) than ImageNet, and thus creates a bridge between these two datasets. As the classes are exactly the same as for CIFAR-10, we can use this dataset for fine-tuning attacks on models that were originally trained on CIFAR-10.

The properties of the datasets are summarised in Table 6.2 and examples of images from each class are shown in Figure 6.1. The datasets that we choose for fine-tuning originally consist of more data samples than MNIST and CIFAR-10. However, we use only a subset of these datasets, since fine-tuning is usually performed using a smaller dataset than the original one. In our experiments, we downsample EMNIST and CINIC-10 to the size of MNIST and CIFAR-10, respectively.

6.2 Neural Networks

We use eight different neural networks for our experiments, from which four are trained on CIFAR-10 and another four on MNIST, as the different complexities of the datasets (grayscaled handwritten digits and real-world images) require different types of models.

- **LeNet [63]** is a group of CNNs developed by Yann LeCun in the 1998. LeNets consist of two convolutional layers, two pooling layers and one (LeNet-1), two (LeNet-4) or three (LeNet-5) dense layers. An illustration of the original architecture of LeNet-5 is shown in Figure 6.4. In our experiments, we use LeNet-1, LeNet-3 and LeNet-5 for training on MNIST. LeNet-3 is a variation of the original LeNet-4 with six instead of four filters in the first convolutional layer. In our implementation, we used max-pooling for downsampling, but also average-pooling is quite common to use.

- **ResNet [47]** stands for Residual Network. ResNets solve the problem of *vanishing gradient*, i.e. the problem of when a network has too many layers, the gradients of the loss function can get zero, thus weights would not get updated and the network would not learn. ResNets counter this problem by utilizing *skip connections* (also called *shortcuts*) to jump over some layers. In our experiments, we use ResNet-18, ResNet-34 and ResNet-50 and train on CIFAR-10. The architectures differ in the number of layers. The number indicates the number of layers: ResNet-18 consists of 18 layers, ResNet-34 of 34 layers, etc. An illustration of a ResNet-18 is shown in Figure 6.2.

- **DenseNet [52]** stands for Densely Connected Convolutional Networks. DenseNets are a type of neural networks that build on the ideas of ResNets. Instead of including skip connections, in DenseNets, each layer has direct access to the gradients of the loss function and the original input image, since each layer takes *all* preceding feature-maps as input. A DenseNet architecture consists of several *dense blocks*. Such a dense block is shown in Figure 6.3, where one can see typical connections between each of the layers. We use the DenseNet-121 in our experiments for CIFAR-10. We sometimes use just the generic term DenseNet, but always mean DenseNet-121.

- **SimpleNet [45]** is convolutional neural network consisting of 13 layers. It was designed to be simple and reasonably deep and still perform similar to deeper and more complex architectures. We use SimpleNet for training on MNIST.

To get an idea for the complexity of the models, we summarised the amount of trainable parameters together with the benchmark test accuracies and the test accuracies of our trained models in Table 6.3. We can see that the models trained on MNIST reach very high results: close to or above 99% accuracy on the test set. The results for CIFAR-10 are around 95% accuracy on the test set.

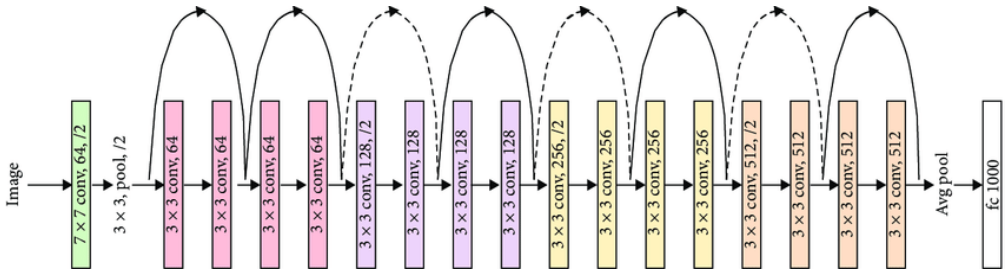


Figure 6.2: ResNet-18 architecture. Source: [7]

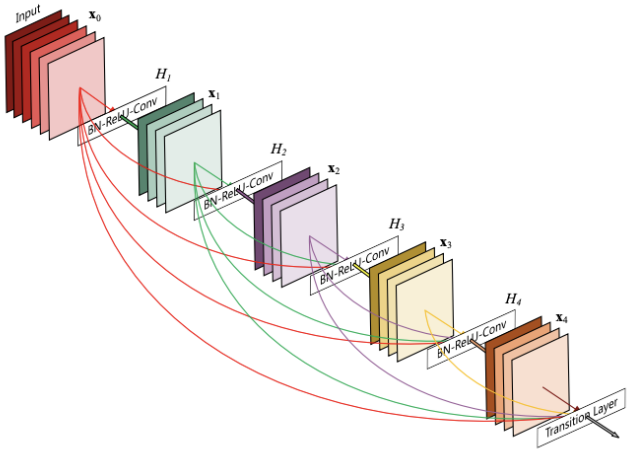


Figure 6.3: A 5-layer dense block. A DenseNet consists of several dense blocks. Source: [52]

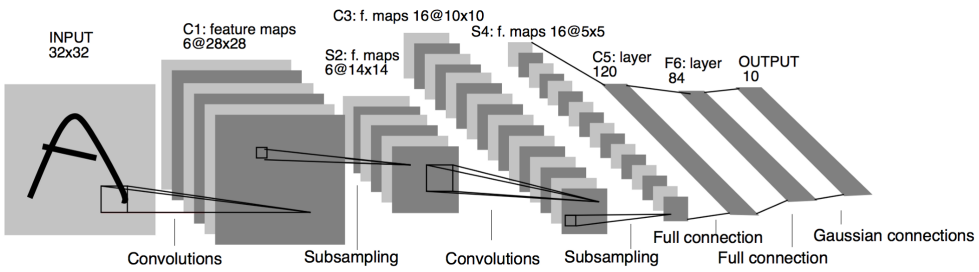


Figure 6.4: Architecture of LeNet-5. Source: [63]

Table 6.3: Amount of trainable parameters and the state-of-the-art test accuracy, as well as, our test accuracy of the trained models.

Architecture	Dataset	Trainable parameters	Benchmark test acc.	Test acc.
LeNet-1	MNIST	7,206	98.3% [1]	98.768%
LeNet-3	MNIST	69,362	-	99.229%
LeNet-5	MNIST	107,786	99.15% [1]	99.229%
DenseNet-121	CIFAR-10	3,272,856	95.04% [2]	94.581%
SimpleNet	MNIST	5,497,226	99.75% [3]	99.589%
ResNet-18	CIFAR-10	11,173,962	95.00% [4]	95.122%
VGG-16	CIFAR-10	14,857,034	92.63% [5]	-
ResNet-34	CIFAR-10	21,282,122	95.95% [4]	95.212%
ResNet-50	CIFAR-10	23,513,162	95.00% [4]	94.391%

[1] `http://yann.lecun.com/exdb/mnist/`
[2] `https://github.com/kuangliu/pytorch-cifar`
[3] `https://paperswithcode.com/sota/image-classification-on-mnist`
[4] `https://github.com/mbsariyildiz/resnet-pytorch`
[5] `https://github.com/chengyangfu/pytorch-vgg-cifar10`

Table 6.4: Trigger set sizes used for training models with various watermarking methods.

Architecture	Dataset	Trigger set sizes	Ratio of dataset size
DenseNet-121	CIFAR-10	20, 100, 500	0.04%, 0.2%, 1%
ResNet-18	CIFAR-10	20, 100, 500	0.04%, 0.2%, 1%
ResNet-34	CIFAR-10	20, 100, 500	0.04%, 0.2%, 1%
ResNet-50	CIFAR-10	20, 100, 500	0.04%, 0.2%, 1%
SimpleNet	MNIST	12, 24, 120, 600	0.02%, 0.04%, 0.2%, 1%
LeNet-1	MNIST	24, 120, 600	0.04%, 0.2%, 1%
LeNet-3	MNIST	24, 120, 600	0.04%, 0.2%, 1%
LeNet-5	MNIST	24, 120, 600	0.04%, 0.2%, 1%

We train multiple versions of the watermarked models, varying the trigger set size, which are listed in Table 6.4. We choose a fixed ratio of the training set, to be able to compare the models for MNIST and CIFAR-10. Note that if the dataset is very large, then even a trigger set size of 0.02% of the dataset size could be still (too) large. Therefore, it would be interesting for future work to evaluate the watermarking methods also on a substantially larger dataset. In this way, we could find out if the trigger set size must reach a specific ratio of the dataset size or requires an absolute number to be effective. Unfortunately, the difference in size between MNIST and CIFAR-10 is not enough for such an experiment.

The trigger set sizes from Table 6.4 are used for all watermarking methods, except *WeaknessIntoStrength*, for which the authors provided a trigger set of only 100 images for download. For this method, we train with a trigger set size of 20 and 100 on *all* architectures. This leads to a total number of **191 trained models**.

Table 6.5: Embedding and fine-tuning time (with learning rate 0.01) for *WeaknessIntoStrength* with 100 trigger images. The time is given in the format (hh:mm:ss).

Arch	**Dataset**	**Embedding**	**Iterations**	**Fine-tuning**	**Iterations**
DenseNet	CIFAR-10	05:09:18	190	02:28:43	100
ResNet-18	CIFAR-10	02:28:46	191	01:21:00	100
ResNet-34	CIFAR-10	04:02:41	195	01:51:41	100
ResNet-50	CIFAR-10	06:16:30	195	02:40:16	100
SimpleNet	MNIST	00:14:12	21	00:40:26	100
LeNet-1	MNIST	00:06:31	21	00:13:11	100
LeNet-3	MNIST	00:06:41	25	00:13:09	100
LeNet-5	MNIST	00:06:47	21	00:13:42	100
Sum		**18:31:26**		**09:42:09**	

Table 6.6: Training times for additional experiments in format (dd:hh:mm:ss).

Method	**# models**	**Embedding**	**Fine-Tuning**	**Both**
FrontierStitching	56	05:08:27:22	05:12:40:06	10:21:07:27
WMEmbeddedSystems	24	02:06:52:53	02:17:10:32	05:00:03:25
Sum				**15:21:10:52**

6.2.1 Training time

All models were trained on a NVIDIA GeForce RTX 2080. To determine the overall training time, we show the training time for embedding the watermark and fine-tuning the watermarked models in Table 6.5, exemplary for *WeaknessIntoStrength* trained with 100 trigger images. We can see from this table that the training time not only depends on the number of iterations, but also on the complexity of the model. For example, a ResNet-34 needs roughly 60% of the time of a ResNet-50 with the same number of training iterations.

If we take $\frac{18.5}{8} \approx 2.3$ hours, as measured in Table 6.5, as the average embedding time for one single model, we get a total embedding time of around 440 hours or around 18.4 days. An estimation of the total fine-tuning time (for one learning rate) results in around 9.6 days, i.e. 19.2 days for both learning rates (cf. Section 7.1.2). This gives us nearly **38 days** ($18.4 + 19.2 = 37.6$) of training to embed the watermark and performing the fine-tuning attacks. Note, that in this number we neither include the time for generating and verifying the watermarks, nor the time for the pruning attacks, nor the time used for the additional experiments for *FrontierStitching* and *WMEmbeddedSystems* (cf. Section 7.2.2) and other experiments that we performed along the way when testing different parameters. The exact embedding and fine-tuning times for the additional experiments for *FrontierStitching* and *WMEmbeddedSystems* are listed in Table 6.6, which adds another **16 days** for the additional experiments on top of the **38 days**, leading to a total of 54 days of compute time.

Table 6.7: Hyperparameters configuration for the architectures. **lr** stands for learning rate, **bs** for batch size, **wm_bs** for watermarking batch size, i.e. the batch size for the trigger set, and **epochs** the number of training iterations.

Architecture	optimizer	lr	scheduler	bs	wm_bs	epochs
DenseNet-121	SGD mom.=0.9 decay=0.0005	0.1	CosineAnnealingLR T_max=200	64	32	200
ResNet-18	SGD mom.=0.9 decay=0.0005	0.1	CosineAnnealingLR T_max=200	64	32	192
ResNet-34	SGD mom.=0.9 decay=0.0005	0.1	CosineAnnealingLR T_max=200	64	32	195
ResNet-50	SGD mom.=0.9 decay=0.0005	0.1	CosineAnnealingLR T_max=200	64	32	192
SimpleNet	SGD mom.=0.9 decay=0.0005	0.1	MultiStepLR n=20, gamma=0.1	64	32	24
LeNet-1	ADAM	0.001	MultiStepLR n=20, gamma=0.1	64	32	48
LeNet-3	SGD mom.=0.9 decay=0.0005	0.1	MultiStepLR n=20, gamma=0.1	64	32	55
LeNet-5	SGD mom.=0.9 decay=0.0005	0.1	MultiStepLR n=20, gamma=0.1	64	32	42

6.3 Setting hyperparameters

We specify the hyperparameters for training the models in Table 6.7. We choose this configuration based on training non-watermarked models on a few different configurations and selecting the model with the minimal validation loss. We test different configurations by varying between the SGD and Adam optimiser as well as the `MultiStepLR` and `CosineAnnealingLR`, which are both commonly used learning rate schedulers. We test the default learning rate for the optimiser, which is $\alpha = 0.1$ for SGD and $\alpha = 0.001$ for Adam, and kept those as we reached results comparable to the benchmarks (cf. Table 6.3). We use this hyperparameters configuration for training the non-watermarked models as well as the watermarked models.

6.3.1 Setting watermark-specific hyperparameters

Some of the papers, e.g. [40], do not mention all of the specific parameters that were used in their experiments. We, therefore, have to experimentally find the optimal parameters for those watermarking methods. We introduce a *ranking system* that helps us to decide on the optimal parameters.

For the experiments, we first fix the trigger set size to 100, as 100 is a trigger set size commonly used in related work, and then train several models, varying one parameter at a time. We train on all architectures and perform pruning attacks as well as fine-tuning attacks. In this manner, we can evaluate the fidelity and robustness of the different models.

Every model for each watermarking method will be evaluated in three categories: *fidelity*, *robustness against pruning*, *robustness against fine-tuning*. The ranking systems works as follows: for each category, the model with the best evaluation gets the n points (where n is the number of different values for the varied parameter) and the one with the lowest performance in the category gets one point. The points are summed up across the categories resulting in a ranking for each of the parameter's values.

For fidelity, the performance is measured by the difference of the test accuracy compared to the non-watermarked model and for robustness against fine-tuning or pruning, by the watermark accuracy after a fine-tuning attack or pruning attack.

Note that we weight all three categories equally. One can argue that, e.g, robustness is the most important property and would therefore weight this category stronger, but such a weighting is very much dependent on the general setting and threat model in which the watermark is being employed. We, therefore, refrain from optimising to a specific assumption.

CHAPTER 7

Empirical comparison of existing watermarking methods

In this chapter, we present the implementation in Section 7.1, where the watermarking methods are discussed in more detail, and the evaluation in Section 7.2, where we present the results and evaluate the watermarking methods.

7.1 Implementation

For an independent comparison of watermarking methods, we implement and compare them in a common framework. We implement or adapt, if there was an existing implementation, the chosen watermarking methods in Python 3.7.10 using PyTorch 1.8.1. A complete list of dependencies is provided in Appendix A.1. The implementation and results can be found in the GitHub project `https://github.com/mathebell/model-watermarking`.

7.1.1 Framework for watermarking methods

In this section, we present the framework that we used and discuss the watermarking methods in more detail, focusing on the implementation.

Algorithm 7.1: General framework

Input: dataset $\mathcal{D}$, initialised or pre-trained model $\mathcal{F}_{\mathbf{w}}$, hyperparameters β
Output: watermarked model $\mathcal{F}_{\mathbf{w}}^{M}$, trigger set $\mathcal{T}$

```
wm_method ← WMMethod.init(β)
T ← wm_method.gen_watermark(F_w, D)
F_w^M ← wm_method.embed_watermark(F_w, D, T)
wm_acc ← wm_method.verify_watermark(F_w^M, T)
if wm_acc ≥ threshold then
    print Watermark was verified.
else
    print Watermark was not verified.
end if
```

Algorithm 7.1 describes the general framework that was used for all the watermarking methods. *WMMethod* stands for the class of a watermarking method. Each of the watermarking methods are implemented in their own class, namely:

- *WeaknessIntoStrength* [4]
- *ProtectingIP* [116]
- *PiracyResistant* [64]
- *ExponentialWeighting* [80]
- *FrontierStitching* [75]
- *WMEmbeddedSystems* [40]

The meaning and usage of each of the member functions, i.e. functions that are defined inside the class, in Algorithm 7.1 are the following:

- *WMMethod.init*(β) initialises the watermarking method depending on the given hyperparameters that are represented by β. The hyperparameters consist of network specific parameters like, e.g., the batch size for the training set (batch_size), the number of training iterations without watermarks (epochs_wo_wm) and the learning rate α, and parameters that are specifying the watermarking method, e.g. the batch size for the trigger set (wm_batch_size), the embedding type (embed_type) and the trigger set size (trg_set_size).

- *WMMethod.gen_watermark*($\mathcal{F}_{\mathbf{w}}, \mathcal{D}$) generates the trigger set $\mathcal{T}$. The trigger set is mostly generated using the original dataset $\mathcal{D}$ and, e.g., placing a trigger pattern onto the image. In some cases such as perturbation based watermarking methods, the watermark generation process makes use of the model itself $\mathcal{F}_{\mathbf{w}}$ and is, therefore, also passed to the function.

- *WMMethod.embed_watermark*($\mathcal{F}_{\mathbf{w}}, \mathcal{D}, \mathcal{T}$) embeds the watermark into the model, i.e. trains the model, besides on the original dataset $\mathcal{D}$, also on the trigger set $\mathcal{T}$. The model is either first trained on the original training set and then fine-tuned with the union of the original training set and the trigger set, or it is trained from the beginning with the unified set. The output of this function is the watermarked model $\mathcal{F}^M_{\mathbf{w}}$.

- $WMMethod.verify\_watermark(\mathcal{F}^M_{\mathbf{w}}, \mathcal{T})$: verifies the watermark. It tests the trigger set $\mathcal{T}$ on the watermarked model $\mathcal{F}^M_{\mathbf{w}}$, i.e. it passes the trigger set to the watermarked model and compares the output of the model with the trigger labels. The output of the function is the accuracy of the model on the trigger set, the watermark accuracy. The threshold, which decides if a watermark can be verified or not, has to be specified in the hyperparameters β.

As the watermarking methods differ very much on the used trigger images, the function *gen_watermark*() is the most customised one. We explain this function for each of the watermarking methods in the following sections.

WeaknessIntoStrength

This method uses out-of-distribution trigger images (OOD). The authors of this method choose a set of abstract images that are completely unrelated to the original dataset (cf. Figure 5.3a) and provided them for download. The trigger set is specifically constructed and is, unfortunately, limited to 100 images, which imposes limitations in the experiment setup. For this method, we thus train the models only with a trigger set size of 20 and 100. However, the method ProtectingIP-OOD serves as an alternative for this method, since it also uses out-of-distribution trigger images, but not "abstract" ones (cf. Section 7.1.1).

The authors compare two different embedding types, namely *pretrained* and *fromscratch*. *Pretrained* means that the watermarks are embedded in a fine-tuning process, *after* the model was already trained and has converged on the clean training set. The pre-trained model is then fine-tuned with the union of the training dataset and the trigger set. When embedding *fromscratch*, we train the model from the beginning with the union of the training dataset and the trigger set. We also used both embedding types for this method and compare the results in Section 7.2.3.

Since the trigger images are already provided and do not have to be generated, the function *WeaknessIntoStrength.gen_watermark*(β) only loads the right amount of trigger images from their storage location (specified by the trigger set size in β).

ProtectingIP

This method considers three watermarking types, namely pattern, noise and OOD:

- **Pattern** trigger images are created by placing a simple text "TEST" in the colour grey on the bottom part of random images from the training dataset (cf. Figure 5.3c).
- **Noise** based trigger images are created by placing a random noise on a random subset of images from the training dataset (cf. Figure 5.3d).
- **OOD** trigger images are formed by a subset of the MNIST dataset for training on CIFAR-10 and vice versa.

The watermarking type is specified in the hyperparameters β and the function *ProtectingIP.gen_watermark*(β) performs the trigger set generation based on the specified watermarking type. The watermark is embedded by training the model from scratch on the union of the original training dataset and the trigger set.

PiracyResistant

This watermarking method relies on an advanced pattern generation. First, a binary pattern is created based on a unique signature, which serves as an additional security measure. As already discussed in Section 5.3.2, the method uses what the authors call *dual embedding*: the model is trained to classify (i) data with a pre-defined binary pattern correctly (called the *null embedding*), i.e. an image with the pattern gets assigned the original label, and (ii) data with an inverted pattern (binary bits are switched) incorrectly (called the *true embedding*), i.e. an image with the inverted pattern gets assigned a different label. This label is also defined by the unique signature. Note that the watermark embedding consists of two parts, the null and true embedding, thus dual embedding does not mean that two watermarks are embedded.

The pattern size is fixed to 6×6 pixels and λ, the strength of pixel change, to 2000, as suggested by the paper's authors.

ExponentialWeighting

This method uses in-distribution trigger images, i.e. images from the training dataset, but purposely labelled wrongly. The trigger label is the next label after the true label in the dataset's *label vector*. For example, the label vector for CIFAR-10 is [airplane, car, bird, cat, deer, dog, frog, horse, ship, truck]. The function *ExponentialWeighting.gen_watermark*(β) randomly samples the trigger set from the original trigger set and assigns the trigger labels in the previously explained manner.

In this method, the model is first trained on the clean training data, i.e. without trigger images. Alternatively, a pre-trained benchmark model is loaded. Then, the exponential

weighting is activated in the layers. In every layer, the weights are transformed in the forward pass by Equation (5.6). We set $\lambda = 2$, as it was used in the authors' experiments. We implemented the exponential weighting in PyTorch by creating sub-classes of the convolutional and feed-forward layer classes. To these sub-classes, we added member functions that activate and deactivate the exponential weighting, i.e. the additional transformation in the forward pass. We used these sub-classes of layers for all architectures, since they work as usual layers when not activated, thus not harming the other watermarking methods.

FrontierStitching

This method is perturbation based. The trigger images are created as adversarial examples of the non-watermarked pre-trained model. The adversarial examples are created by the Fast Gradient Sign Method (FGSM). FGSM expects a parameter ϵ, which controls the intensity of the adversarial perturbation. As a next step, the adversarial examples are divided into *true* and *false* adversarials[1] (cf. Figure 5.5). A *true* adversarial is an adversarial example for which the model predicts a different label than the original true label. A *false* adversarial, on the other hand, is an adversarial example for which the model still predicts the original true label, i.e. the adversarial image did not manage to fool the model. Both true and false adversarials are saved as trigger images, with the original true label as the trigger label. The sets of true and false adversarials have the same size, each constituting half of the trigger set size.

We would find it interesting to analyse how the perturbation parameter ϵ influences the behaviour of the watermarked model. We therefore vary ϵ in our experiments (cf. Section 7.2.2). For the main analysis, however, we choose $\epsilon = 0.25$ as the default parameter, following the authors' experiments.

WMEmbeddedSystems

This method is pattern based. The pattern is created based on a unique signature, and subsequently then embedded with strength λ. Given the original image $\mathbf{X}$, the pattern p and the strength λ, the trigger image is created by

$$\mathbf{X}^{\text{trg}} = \mathbf{X} + \lambda p \tag{7.1}$$

Two hyperparameters that can be varied in this method are λ – the magnitude or strength of the pattern on the trigger images – and the number of bits embedded in the trigger images. We fixed the number of bits to 128, following the authors. For λ we performed several experiments in Section 7.2.2, to find an appropriate value for λ, as the authors did not provide such information in their paper.

[1] The publication calls them true and false *adversaries*. We, however, call the attacker of the model an adversary and therefore choose the term *adversarial*.

Table 7.1: Attacks used in the papers.

	Pruning	**Fine-Tuning**	**Other attacks?**
WeaknessIntoStrength [4]		✓(STL-10)	
ProtectingIP [116]	✓	✓[1]	
PiracyResistant [64]	✓	✓[2]	Neural Cleanse [102], Model Extraction Attack [100], Piracy Attack
ExponentialWeighting [80]			Query Modification [80]
FrontierStitching [75]	✓	✓[2]	
WMEmbeddedSystems [40]			

[1] use half of the original test set for fine-tuning and the other half for evaluating the model
[2] do not specify the dataset for fine-tuning

7.1.2 Attacks

We provide a list of attacks that are used in the papers in Table 7.1. We can see from that table that most of the papers test their watermarking methods against pruning and fine-tuning. *ExponentialWeighting* presents their own attack *Query Modification*, a watermark invalidation. It detects if the queried sample is a trigger image by applying an autoencoder. If the model predicts for the changed image a different label than for the queried image, it probably is a trigger image and the prediction for the changed image is returned. They test it on several other methods and confirm that for [116, 75, 86] the watermark accuracy after a query modification is very low and therefore introduce a new watermarking method, which resists this kind of attack. The authors of *WMEmbeddedSystems* do not use pruning or fine-tuning, as they assume that a user of the watermarked model and thus a possible attacker would not have enough computational power or technical expertise for such an attack, otherwise they would have trained their own model. As a second argument, they state that a fine-tuned model is a different model and it is questionable whether the model owner can claim the ownership of a fine-tuned model.

Parameter Pruning

Since many of the papers (cf. Table 7.1) considered parameter pruning as a valid attack for testing robustness, we implemented this attack as well.

For this attack, we take the watermarked model and set those weights that have the smallest absolute value to zero. We vary the *pruning rate* – the rate of weights that are pruned in the model – between 10% and 90% with a step-size of 10 percentage points.

We consider the pruning attack as plausible if the attack does degrade the model's test accuracy only at the maximum of 3.5%, since an attacker would likely not use a model that is significantly degraded. Therefore, we choose the maximum pruning rate at which the test accuracy does not fall under this threshold and then compare the watermark accuracies. We chose 3.5% as a value in accordance to [71]; for example, [68] chose a very

similar value of 4% when defending against backdoor attacks, which is essentially the same task as removing a watermark based on backdoors.

Note that there is also a different notion of pruning, also called *fine-pruning* [68], that relies on the availability of a clean set, to prune by activation value. This could also be a potential threat for the watermark, however, we keep this for future work.

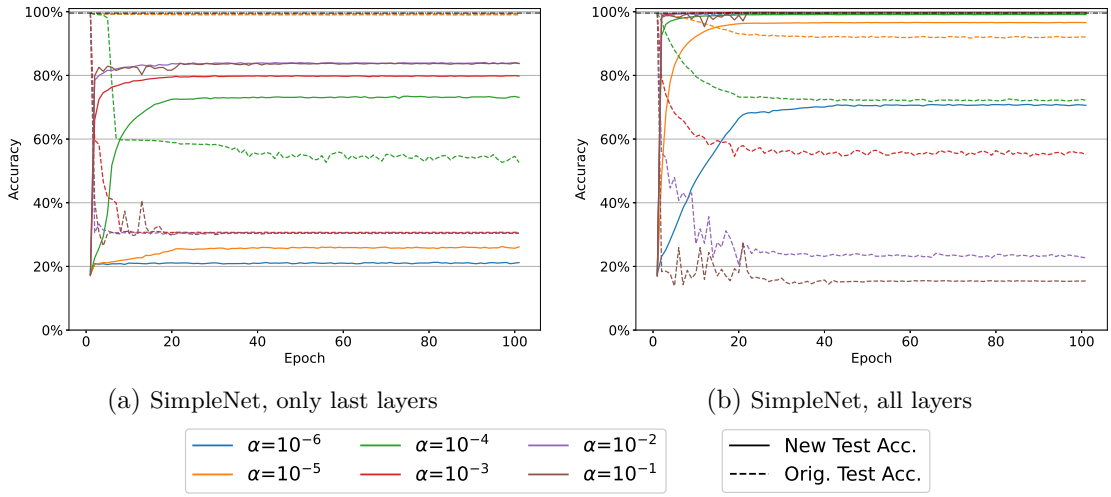
Fine-Tuning

Fine-tuning is another common attack. Some of the papers fine-tune on the same dataset (without trigger set) as the training was done, others use a different and smaller dataset and some do not mention which dataset was used (cf. Table 7.1). As we assume that an attacker would not have access to the original training data, or only to a portion of it, we perform fine-tuning on a different, but similar dataset. The EMNIST dataset consists of different images than MNIST, but was build from the same database NIST, while around 22 % of the CINIC-10 dataset consist of CIFAR-10 images, the other part of ImageNet. Fine-tuning with EMNIST and CINIC-10, therefore, simulates a strong attack, as we assume that the attacker knows the distribution of the original dataset. Our fine-tuning datasets do originally consist of more example-labels pairs, but we randomly downsample them to the same size as the original training set, as we assume that an attacker would have less or the same amount of data at most.

Fine-Tuning is usually performed with a smaller learning rate than training, to not "overwrite" too much of the previously learned knowledge. Different learning rates achieve different success, as we analyse in Section 7.2.1. For the evaluation, we choose both a smaller learning and a larger learning rate. A smaller learning rate would correspond to *transfer-learning*, and a larger learning rate rather to *re-training*. Transfer learning aims to exploit the fact that the trained model generalises well and requires little adaption to the new task – with a small learning rate only small changes are made to the model. On the other hand, re-training would mean that the model learns the new dataset from scratch and forgets the old dataset. We will discuss this phenomenon in more detail in Section 7.2.1. We implemented both attacks since they have different motivations and are plausible attacks – transfer learning could happen as an accidental attack and re-training as an attack for purposefully removing the watermark. One could argue that this is not a plausible attack, since the model forgets the old dataset and the attacker could train his own model from scratch, but still, we want to analyse how the watermark behaves during such a strong attack.

7.2 Evaluation

In this section, we evaluate the experiments. In particular, we first evaluate the experiments for different fine-tuning settings in Section 7.2.1 and the influence of watermark-specific parameters in Section 7.2.2. Then, we compare our results to the results in the papers in Section 7.2.3 and afterwards evaluate and compare the watermarking methods



(a) SimpleNet, only last layers

(b) SimpleNet, all layers

Figure 7.1: Fine-tuning on non-watermarked SimpleNet. The plot on the left side correspond to fine-tuning only the last layer and the one on the right hand side to fine-tuning all layers. The black dash-dotted line corresponds to the benchmark test accuracy of the non-watermarked model.

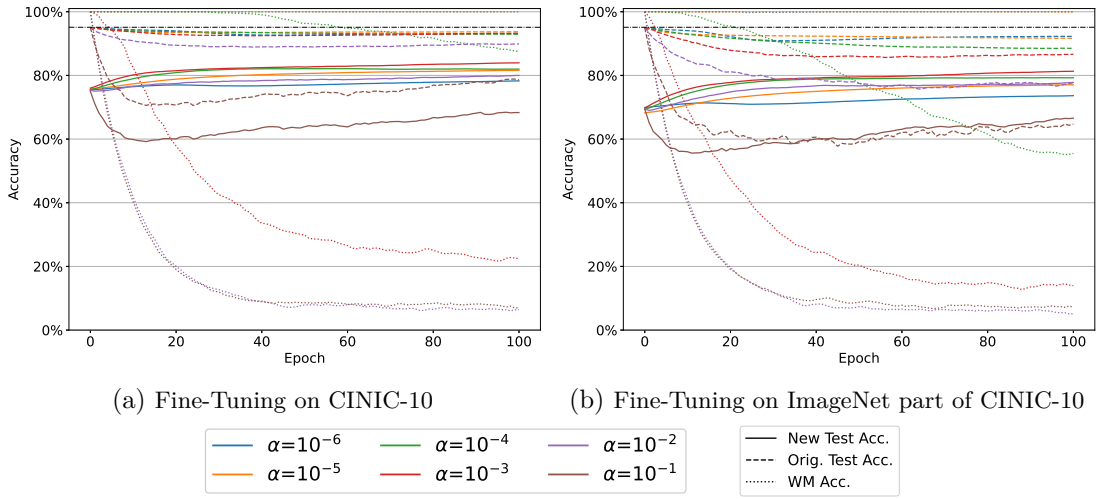
one to each other regarding **effectiveness**, **fidelity** and **robustness** in Section 7.2.4, Section 7.2.5 and Section 7.2.6, respectively.

Effectiveness is evaluated by measuring the watermark accuracy after watermark embedding, fidelity by measuring the difference between the test accuracy of the watermarked and non-watermarked model, and robustness by measuring the watermark accuracy after an attack.

7.2.1 Evaluation of Fine-Tuning

In this section, we present the findings when testing several settings for fine-tuning.

First, we compare fine-tuning that trains only the last layer with fine-tuning that trains on all layers. We fine-tune non-watermarked models and conclude that fine-tuning on all layers leads to better results, as we can see, e.g. on SimpleNet, in Figure 7.1. When fine-tuning only the last layer, the model seems to be unable to learn the new data properly. For a small learning rate, e.g. $\alpha = 10^{-6}$, when fine-tuning only the last layer, the test accuracy of the new dataset stays at 20% for all iterations, while the test accuracy on the original test set stays at the benchmark. With the same learning rate and fine-tuning all layers, the model is able to learn to predict the new dataset with around 70% accuracy. For a large learning rate, e.g. $\alpha = 10^{-2}$, the original test accuracy drops to around 30%, regardless of whether fine-tuning only the last layer or all layers. At the same time, when fine-tuning all layers, at least the new test accuracy reaches almost the benchmark score, whereas when fine-tuning only the last layer, the new test accuracy reaches only around 85%.



(a) Fine-Tuning on CINIC-10

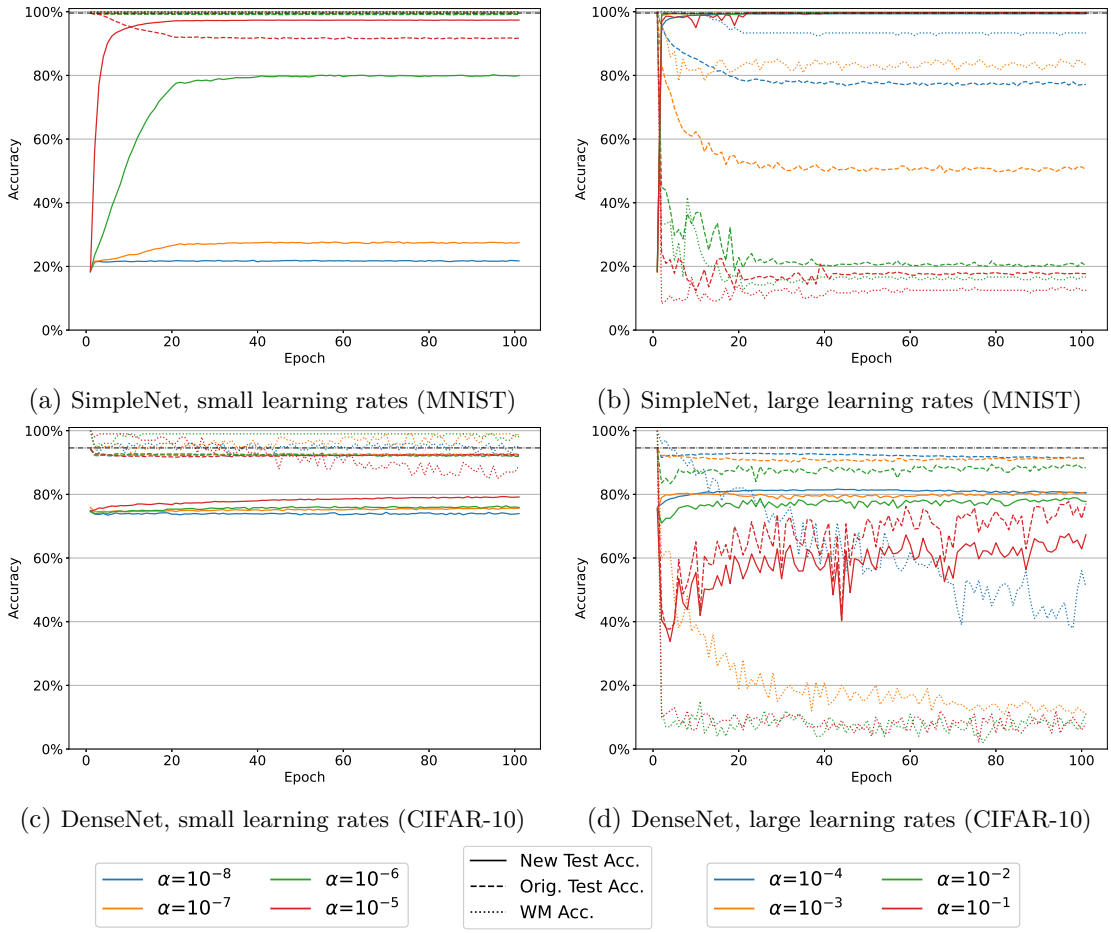
(b) Fine-Tuning on ImageNet part of CINIC-10

Figure 7.2: Fine-Tuning on both, CINIC-10 and only on the ImageNet part of CINIC-10. In both cases, 50,000 images are randomly chosen from the corresponding dataset. The underlying model is a ResNet-18 that was trained with *ProtectingIP-pattern* and 100 trigger images. The black dash-dotted line corresponds to the benchmark test accuracy of the non-watermarked model. For clearity reasons, the lines in the plot are smoothed. The original plots are provided in Figure A.3.

In summary, the original test accuracy behaves similarly in both settings. Even though it is a bit worse when fine-tuning all layers, the new test accuracy reaches better results when fine-tuning all layers. It, of course, depends on the motives of an attacker, but we believe that an attacker would either perform transfer-learning where both the new and the original test accuracy are at an acceptable level ($\alpha = 10^{-5}$) or would try to purposefully overfit the new data (e.g. $\alpha = 10^{-1}$), in order to let the model "forget" the original data (and watermark). We analyse the influence of different learning rates in the following.

From now on, in the following experiments and also for the evaluation of the robustness of the watermarking methods, we always fine-tune on all layers.

Since CINIC-10 is a dataset consisting of 90,000 training images, from which 70,000 are drawn from ImageNet, and we downsample the fine-tuning dataset to 50,000 training images, we could either randomly choose 50,000 training images from CINIC-10 or only from the ImageNet part of CINIC-10. In Figure 7.2 we show the results for fine-tuning on CINIC-10 and on the ImageNet part of CINIC-10 for learning rates from 10^{-6} to 10^{-1}. We conclude that the choice of the dataset does not make much difference on the new test accuracy. However, fine-tuning on the ImageNet part leads to both a quicker original test accuracy and watermark accuracy drop for higher learning rates. In the following, we fine-tune using CINIC-10, as the attacker would also use the most similar dataset for the purpose of transfer-learning. For the purpose of re-training, we see that the watermark accuracy drops below 20% for large learning rates, anyhow, so the differences in these versions of the data do not have a large effect.



(a) SimpleNet, small learning rates (MNIST)

(b) SimpleNet, large learning rates (MNIST)

(c) DenseNet, small learning rates (CIFAR-10)

(d) DenseNet, large learning rates (CIFAR-10)

Figure 7.3: Fine-tuning on SimpleNet and DenseNet, watermarked with ProtectingIP-pattern. The plots on the left side correspond to fine-tuning with smaller learning rates and the ones on the right side to fine-tuning with larger learning rates. The black dash-dotted line corresponds to the benchmark test accuracy of the non-watermarked model.

In another experiment, we test several learning rates in order to see how the choice of learning rate influences (i) the test accuracy on the original dataset, (ii) the test accuracy on the new dataset, and (iii) the watermark accuracy. For selecting a learning rate for attacks on all other watermarking methods, we fine-tune exemplary only on the models that were watermarked with *ProtectingIP-pattern* and with 100 trigger images. Figure 7.3 shows the results for DenseNet (on CIFAR-10) and for SimpleNet (on MNIST). Although the behaviour differs much between the two datasets, the behaviour for the models across one dataset is very similar. That is why we focus on DenseNet and SimpleNet as examples here and provide the plots for the other models in Appendix A.2.1. We can conclude from Figure 7.3b that fine-tuning on MNIST models with a larger learning rate tends to fit the new data too much and therefore the model "forgets" the original dataset and watermark, i.e. the test accuracy on the original dataset and the watermark

(a) 0.0001 (b) 0.001 (c) 0.01 (d) 0.1 (e) 0.25 (f) 0.5 (g) 1.0

Figure 7.4: Examples for FrontierStitching trigger images for different values of ϵ, created with FGSM on LeNet-1.

accuracy drops already after a few training iterations. This phenomenon is referred to as *catastrophic forgetting* [57]. A model fine-tuned with a smaller learning rate, on the other hand, does not tend to overfit and either learns to adapt to the new data with only a little accuracy drop on the original test data ($\alpha = 10^{-5}$ in Figure 7.3a) or is unable to learn the new data at all ($\alpha = 10^{-8}$ in Figure 7.3a).

Considering CIFAR-10, shown in Figures 7.3c and 7.3d, we do not see this catastrophic forgetting regarding the old training data for any of the learning rates, perhaps because the datasets are too similar. However, we do clearly see that for larger learning rates the watermark accuracy drops, i.e. the model "forgets" the watermarks more quickly.

7.2.2 Influence of watermark-specific hyparameters

In the following subsections, we discuss two methods, namely *FrontierStitching* and *WMEmbeddedSystems*, which we analysed regarding the influence of a method-specific parameter.

FrontierStitching

In this section, we analyse how the parameter ϵ for FGSM influences fidelity and robustness of the watermarking method. We use [0.0001, 0.001, 0.01, 0.1, 0.25, 0.5, 1.0] as ϵ values. Figure 7.4 shows examples of trigger images for the different values of ϵ.

When it comes to effectiveness, all models reach 100% watermark accuracy, except of LeNet-5 with $\epsilon = 0.25$, which has a watermark accuracy of 68%. We see later in Section 7.2.4 that, at least for $\epsilon = 0.25$, the watermark accuracy on LeNet-5 drops with the trigger set size.

In the first experiment, we test the influence of the parameter ϵ on the validation loss difference, e.i. the validation loss of the non-watermarked model subtracted from the validation loss of the watermarked model. We expect that watermarked models trained on trigger images with a larger perturbation lead to a higher performance drop, i.e. the validation loss is higher than for models trained on less perturbed trigger images.

The results are summarised in Figure 7.5. We conclude from this experiment that the influence of ϵ is very much dependent on the architecture. The LeNets tend to perform better with a higher ϵ, as the relative difference in validation loss drops from 1.0-1.4 to

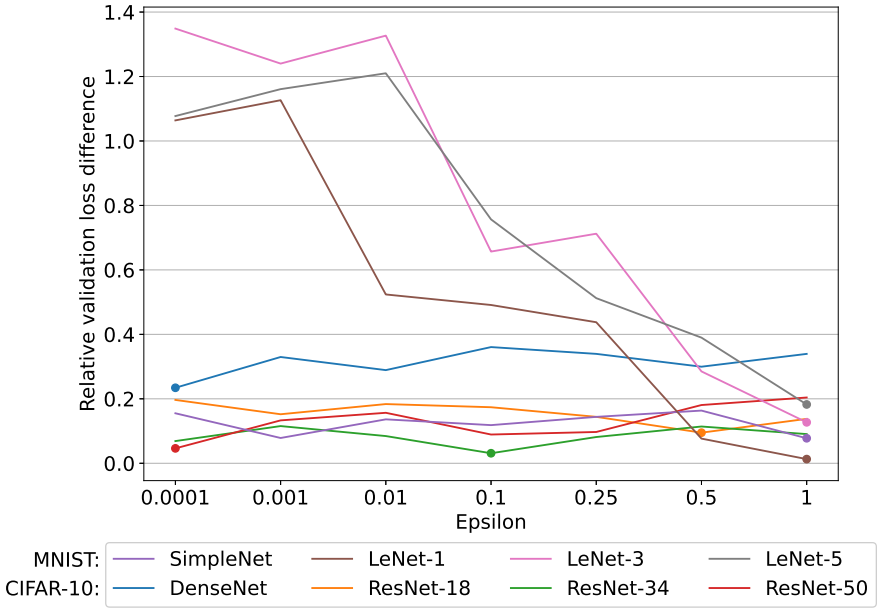
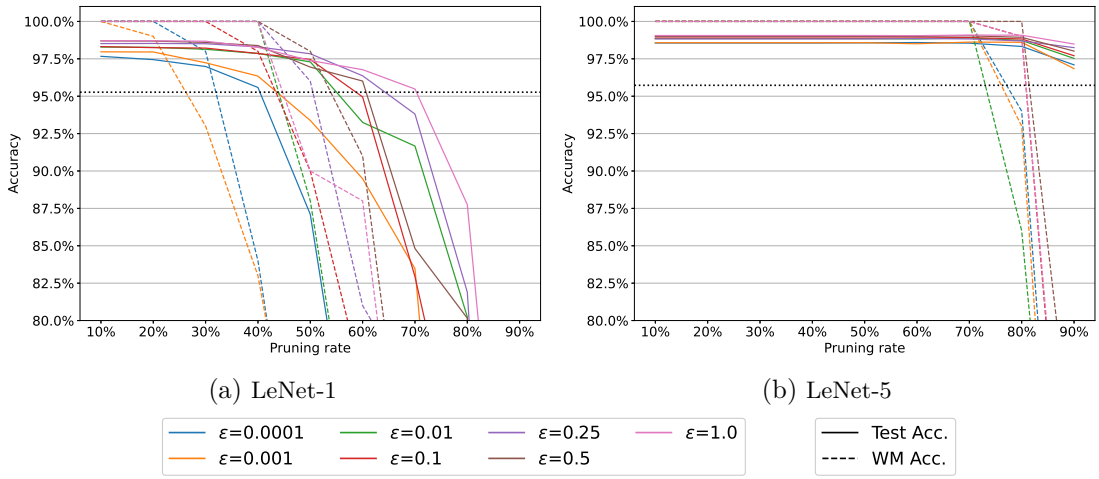


Figure 7.5: FrontierStitching with various values for ϵ (strength of perturbation). The plot shows the relative validation loss difference, i.e. the difference between the validation loss of the watermarked model and the non-watermarked benchmark model divided by the validation loss of the benchmark model. For all values the WM Accuracy is 100%. The dots in the plot represent the minimal validation loss difference for the respective architecture.

0-0.2. The value of ϵ does not affect the other models much, as the relative validation loss difference is between in a range of ± 0.1 for all ϵ. For DenseNet and ResNet-50, however, we see a trend that a smaller ϵ leads to better fidelity.

Since a good watermarking method should not only achieve a good fidelity but also good robustness, we perform pruning and fine-tuning on these models to test robustness. The results for pruned models depend very much on the complexity of the architecture. Figure 7.6 shows the results for LeNet-1 and LeNet-5. The pruning attack affects the smaller architecture much more than the more complex one. Even with a small pruning rate of 20%, both the test and watermark accuracy of LeNet-1 start to drop. Compared to this, a LeNet-5 does not change in performance until 70% of the weights are pruned. This could indicate that LeNet-5 has spare capacity and is more complex than needed for the task.

As already discussed in Section 7.1.2, we consider a pruning attack as plausible when the test accuracy drops 3.5% at maximum. In Figure 7.6 this threshold is marked with a black dotted line. For instance, we see that the maximal plausible pruning rate for LeNet-5 in Figure 7.6b is 90% for all values of ϵ, whereas for LeNet-1 in Figure 7.6a it very much depends on the parameter ϵ. The maximal plausible pruning rate for LeNet-1, e.g., for the values 0.0001 and 0.001 is 40%, for 0.5 it is 60% and 1.0 it is 70%.



(a) LeNet-1 (b) LeNet-5

Figure 7.6: Model accuracy after pruning attacks with pruning rates from 10% to 90%. The black dotted line indicates the threshold for the maximal plausible pruning attack.

As the third and last experiment, we perform a fine-tuning attack on the models. First, we assume that an attacker would perform a fine-tuning attack with the purpose of removing the watermark and would therefore choose a relatively high learning rate (cf. Section 7.2.1, where we tested several learning rates and analysed the behaviour of the attacked model). We choose 0.01 as the learning rate for both MNIST and CIFAR-10, based on the experiments in Section 7.2.1. Unfortunately, for all models, the watermark accuracy after the fine-tuning attack with a large learning rate is below 20%. Second, we perform a fine-tuning attack with a relatively low learning rate (cf. Section 7.2.1). We choose 10^{-5} for MNIST and 10^{-4} for CIFAR-10, based on the experiments in Section 7.2.1. After fine-tuning with this small learning rate, the watermark accuracy stays above 50% for most of the models. We summarised the results in Table 7.2.

WMEmbeddedSystems

Since the authors of the paper do not mention which specific value λ (the magnitude or strength of the pattern on the trigger images) they used, we have to find the optimal value for λ. We want to find a specific value for λ for each dataset, but not for each architecture, since the watermark generation does not depend on the model itself and should therefore be created in the same manner for all architectures trained on the same dataset. We average the points for each λ, grouped by the dataset. The ranking system is presented in Table 7.3. The winning magnitude λ for CIFAR-10 is 0.1 and for MNIST 1.0.

Table 7.2: Watermark accuracies after fine-tuning attack on models trained with FrontierStitching.

CIFAR-10				**MNIST**			
Arch.	ϵ	lr=10^{-2}	lr=10^{-4}	Arch.	ϵ	lr=10^{-2}	lr=10^{-5}
DenseNet	0.0001	6%	65%	SimpleNet	0.0001	12%	100%
	0.001	8%	68%		0.001	12%	97%
	0.01	9%	74%		0.01	9%	100%
	0.1	10%	68%		0.1	12%	94%
	0.25	11%	75%		0.25	9%	92%
	0.5	15%	53%		0.5	10%	96%
	1	13%	24%		1	12%	92%
ResNet-18	0.0001	11%	76%	LeNet-1	0.0001	12%	46%
	0.001	17%	82%		0.001	12%	54%
	0.01	11%	86%		0.01	10%	43%
	0.1	13%	86%		0.1	10%	59%
	0.25	4%	94%		0.25	11%	57%
	0.5	9%	67%		0.5	11%	74%
	1	9%	53%		1	10%	63%
ResNet-34	0.0001	15%	60%	LeNet-3	0.0001	8%	51%
	0.001	14%	69%		0.001	7%	66%
	0.01	8%	78%		0.01	6%	53%
	0.1	13%	57%		0.1	11%	74%
	0.25	12%	71%		0.25	9%	48%
	0.5	12%	49%		0.5	11%	37%
	1	13%	48%		1	9%	51%
ResNet-50	0.0001	8%	93%	LeNet-5	0.0001	7%	45%
	0.001	8%	97%		0.001	9%	40%
	0.01	9%	99%		0.01	6%	65%
	0.1	15%	100%		0.1	7%	55%
	0.25	15%	66%		0.25	8%	21%
	0.5	8%	99%		0.5	5%	58%
	1	14%	65%		1	6%	57%

Table 7.3: Results for ranking system for WMEmbeddedSystems. The points are averaged for each dataset and the bold numbers indicate the highest average for each dataset and therefore the winning ϵ.

	Arch.	$\lambda = 0.1$	$\lambda = 0.5$	$\lambda = 1.0$
CIFAR-10	DenseNet	**6**	**6**	**6**
	ResNet-18	**7**	4	**7**
	ResNet-34	**8**	4.5	5.5
	ResNet-50	**8.5**	4.5	5
	Average	**7.375**	4.75	5.875
MNIST	SimpleNet	4	6	**8**
	LeNet-1	3.5	**4.5**	4
	LeNet-3	**7**	6	5
	LeNet-4	**7**	5	6
	Average	5.375	5.375	**5.75**

Table 7.4: Fidelity results from Adi et al. ([4], Table 1), and our experiments.

	Adi et al.'s results		Our results	
Model	Test Acc.	WM Acc.	Test Acc.	WM Acc.
No WM	93.42	7.0	95.122	-
Fromscratch	93.81	100.0	94.94	100.0
Pretrained	93.65	100.0	94.61	100.0

7.2.3 Comparing to State of the Art

In this section, we compare our results to the results in the literature. Note that several results are not fully comparable because of the different study settings, e.g. a custom architecture or unknown trigger set size. Some of the methods focus on a special attack and therefore do not present data that is relevant to our work. *PiracyResistant* [64] focuses on piracy resistance and *ExponentialWeighting* [80] on their own crafted attack. Not knowing all hyper-parameters used in literature further hinders achieving the exact results from related work.

WeaknessIntoStrength

Adi et al. [4] train a ResNet-18 on CIFAR-10, CIFAR-100 and ImageNet with 100 trigger images. We compare the author's results for ResNet-18 on CIFAR-10 with our results. For fidelity, the authors presented results on the test and watermark accuracy after watermark embedding *fromscratch* and *pretrained*. We extract the same information from our models and summarise both, our and the author's results, in Table 7.4. While the authors have a marginal increase, we have a marginal drop in accuracy in the watermarked model compared to the non-watermarked one, but both differences seem negligible. We can also confirm that embedding *fromscratch* leads to better results than embedding the watermark in a *pretrained* model.

We tested both embedding types on all models. For instance, Figure 7.7 shows the behaviour of DenseNet during watermark embedding for both embedding types. We conclude that embedding *fromscratch* leads to better fidelity, which is consistent with the conclusions by the authors (cf. Figure 7.7).

ProtectingIP

We present the results on effectiveness from Zhang et al. [116] and our results in Table 7.5. They tested the watermarks on a custom DNN and on MNIST and CIFAR-10. We reach slightly better results, as our models for *ProtectingIP* reach all 100% watermark accuracy (cf. Table 7.9). It is worth noting that the paper does not provide information on the trigger set size and we, therefore, cannot conclude where the difference in effectiveness could stem from.

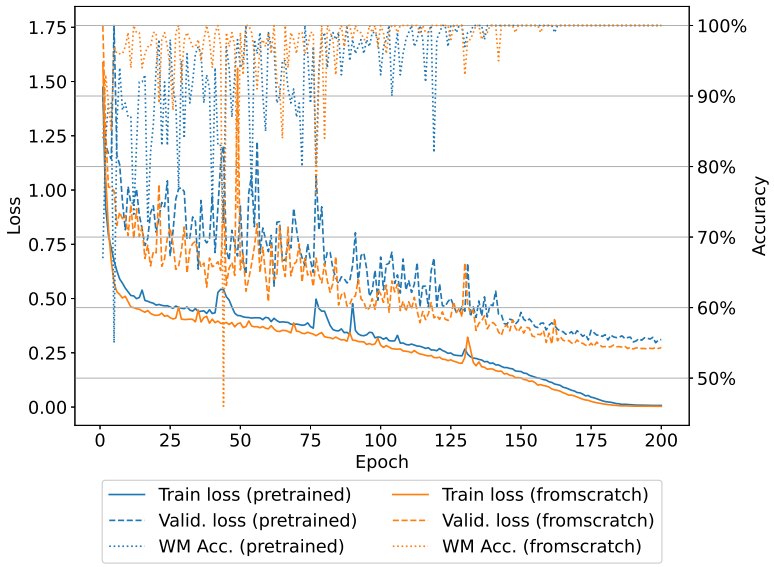


Figure 7.7: Behaviour of DenseNet during training with embedding type *pretrained* and *fromscratch*.

Table 7.5: Effectiveness results from Zhang et al. ([116], Table 1), and our results.

	Zhang et al.'s results		Our results	
	MNIST	**CIFAR-10**	**MNIST**	**CIFAR-10**
Method	WM Acc.	WM Acc.	WM Acc.	WM Acc.
ProtectingIP-pattern	100%	99.93%	100%	100%
ProtectingIP-OOD	100%	100%	100%	100%
ProtectingIP-noise	100%	99.86%	100%	100%

Regarding robustness against pruning, we present the authors' results right beside ours in Table 7.6. Our results for CIFAR-10 are from ResNet-18 trained with 100 trigger images and for MNIST from LeNet-5 trained with 120 trigger images. We compare only the watermark accuracy, since we used different architectures, and mark in green and red where our models perform better resp. worse.

The most noticeable difference is that for the MNIST model watermarked with *ProtectingIP-pattern* and *ProtectingIP-OOD*, our watermark accuracy drops quicker. Also, for almost all models when 90% of the parameters are pruned, the watermark accuracy of Zhang et al. is higher than ours. We again want to mention that Zhang et al. [116] used custom DNNs, which makes the results difficult to compare. Their custom DNN for MNIST has 312,202 trainable parameters, which are three times as much compared to our LeNet-5 with 107,786 trainable parameters. Their custom DNN for CIFAR-10 has 1,146,826 trainable parameters, which are only 10% compared to our ResNet-18 with 11,173,962 trainable parameters. As they use a DNN with more parameters for MNIST, the model might have spare capacity, i.e. might be too large for the task, and thus shows such high

Table 7.6: Pruning results from Zhang et al. ([116], Table 3 and Table 4), compared with our results. "Test" stands for test accuracy, "WM" for watermark accuracy and "Pr. rate" for pruning rate.

MNIST												
Pr. rate	*ProtectingIP-pattern*				*ProtectingIP-OOD*				*ProtectingIP-noise*			
	Our results		Zhang's results		Our results		Zhang's results		Our results		Zhang's results	
	Test	WM	Test	WM	Test	WM	Test	WM	Test	WM	Test	WM
10%	98.50%	100%	99.44%	100%	98.61%	100%	99.43%	100%	98.48%	100%	99.4%	100%
20%	98.50%	100%	99.45%	100%	98.58%	100%	99.45%	100%	98.44%	100%	99.41%	100%
30%	98.30%	100%	99.43%	100%	98.55%	100%	99.41%	100%	98.34%	100%	99.41%	100%
40%	98.17%	100%	99.40%	100%	98.45%	100%	99.31%	100%	98.26%	100%	99.42%	100%
50%	98.05%	100%	99.29%	100%	98.22%	100%	99.19%	100%	98.21%	100%	99.41%	100%
60%	97.33%	98.33%	99.27%	100%	97.63%	95.83%	99.24%	100%	98.04%	100%	99.3%	99.9%
70%	95.84%	89.17%	99.18%	100%	96.26%	74.17%	98.82%	100%	97.96%	100%	99.22%	99.9%
80%	86.98%	71.67%	98.92%	100%	96.60%	75.00%	97.79%	100%	97.34%	100%	99.04%	99.9%
90%	71.50%	51.67%	97.03%	99.95%	85.02%	35.00%	93.55%	99.9%	87.56%	45.83%	95.19%	99.55%
CIFAR-10												
Pr. rate	*ProtectingIP-pattern*				*ProtectingIP-OOD*				*ProtectingIP-noise*			
	Our results		Zhang's results		Our results		Zhang's results		Our results		Zhang's results	
	Test	WM	Test	WM	Test	WM	Test	WM	Test	WM	Test	WM
10%	95.02%	100%	78.37%	99.93%	95.28%	100%	78.06%	100%	94.78%	100%	78.45%	99.86%
20%	95.04%	100%	78.42%	99.93%	95.31%	100%	78.08%	100%	94.77%	100%	78.5%	99.86%
30%	95.01%	100%	78.2%	99.93%	95.34%	100%	78.05%	100%	94.78%	100%	78.33%	99.93%
40%	94.90%	100%	78.24%	99.93%	95.28%	100%	77.78%	100%	94.84%	100%	78.31%	99.93%
50%	94.75%	100%	78.16%	99.93%	95.25%	100%	77.75%	100%	94.80%	100%	78.02%	99.8%
60%	91.91%	100%	77.87%	99.86%	95.01%	100%	77.44%	100%	94.66%	100%	77.87%	99.6%
70%	74.76%	99.00%	76.7%	99.86%	94.26%	100%	76.71%	100%	94.31%	100%	77.01%	98.46%
80%	54.19%	73.00%	74.59%	99.8%	88.63%	89.00%	74.57%	96.39%	89.38%	94.00%	73.09%	92.8%
90%	44.15%	31.00%	64.9%	99.47%	15.43%	12.00%	62.15%	10.93%	13.75%	13.00%	59.29%	65.13%

robustness against pruning. The results for CIFAR-10 are not that clear: our models do perform better than the MNIST models but not as good as we would expect. We conclude that robustness against pruning not only depends on the complexity of the architecture but also the structure and the learned parameters itself.

FrontierStitching

Merrer et al. [75] tested their watermarking method on two models (a CNN and an MLP, cf. Table 6.1) on MNIST. Our results are for LeNet-1 and LeNet-3 trained with 120 trigger images. We present their and our results for robustness against pruning in Table 7.7. Note that the authors chose a larger step size for pruning, of 25%, and thus not all results have a matching counterpart with our results. In Table 7.7, the grey rows indicate non-plausible pruning attacks, attacks where the test accuracy fall more than 3.5%. One may assume that their models are less complex than ours because of the quicker test and watermark accuracy drop. In fact, their both models have more trainable parameters (1,199,882 for CNN and 235,146 for MLP) compared to 69,362 for LeNet-3 and 7,206 for LeNet-1 (cf. Table 6.3). The results for pruning are very much dependent on the model – not only on the complexity of the architecture but the distribution of parameters in the trained model. Therefore, we cannot fully compare the results, as we used different architectures.

Table 7.7: Pruning results from Merrer et al. ([75], Table 2), and our results. The grey cells indicate a non-plausible pruning attack. For a plausible attack and watermark accuracy above 50% the cell is green and below it is red.

Merrer et al.'s results				Our results		
Model	Pr. rate	Test	WM	Model	Test	WM
CNN	20%	-	-	LeNet-3	98.7%	100%
	25%	98.7%	100%		-	-
	30%	-	-		98.7%	100%
	50%	88.5%	100%		98.7%	100%
	70%	-	-		98.7%	100%
	75%	42.6%	0%		-	-
	80%	-	-		98.7%	100%
	90%	25.5%	0%		98.2%	80.8%
MLP	20%	-	-	LeNet-1	98.5%	100%
	25%	97.3%	100%		-	-
	30%	-	-		98.5%	100%
	50%	96.6%	100%		97.6	85%
	70%	-	-		87.7%	55.8%
	75%	95.3%	7.7%		-	-
	80%	-	-		80.7%	43.3%
	90%	15.7%	0%		39.8%	28.3%

Table 7.8: Fidelity results from Guo et al. ([40], Table 2), and our results.

		Guo et al.'s results		Our results	
Dataset	Model	Test Acc.	WM Acc.	Test Acc.	WM Acc.
MNIST	LeNet-5	98.99%	10%	99.23%	-
	LeNet-5 WM	98.48%	98.38%	99.18%	100%
CIFAR-10	ResNet-50	94.53%	2.2%	94.39%	-
	ResNet-50 WM	94.25%	99.98%	94.47%	100%
	DenseNet	94.73%	2.2%	94.58%	-
	DenseNet WM	94.23%	99.97%	94.10%	100%

WMEmbeddedSystems

Guo et al. [40] tested their watermarking method for MNIST on LeNet-5, and for CIFAR-10 on ResNet-50, DenseNet and VGG. We compare the results on fidelity for the architectures ResNet-50, DenseNet and LeNet-5. For comparison, we take the models that were trained with 100 (CIFAR-10) and 120 (MNIST) trigger images. The authors' and our results are shown in Table 7.8.

Similar to the authors, our watermarked models have a minimal test accuracy drop, below 0.5%, and in the case of ResNet-50 even a test accuracy gain, compared to the non-watermarked models. Our models, however, reach 100% watermark accuracy for this method, which are slightly better results than those presented by the authors.

Table 7.9: Watermark accuracy on watermarked models. A checkmark ✓indicates 100% watermark accuracy.

Method	Type	Trg set size	DenseNet	ResNet-18	ResNet-34	ResNet-50	SimpleNet	LeNet-1	LeNet-3	LeNet-5
ProtectingIP	pattern	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	✓
		500/600	✓	✓	✓	✓	✓	✓	✓	✓
	noise	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	✓
		500/600	✓	✓	✓	✓	✓	✓	✓	✓
	OOD	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	✓
		500/600	✓	✓	✓	✓	✓	✓	✓	✓
Weakness IntoStrength	OOD	20	✓	✓	✓	✓	✓	✓	✓	✓
		100	✓	✓	✓	✓	✓	99.00%	✓	✓
Piracy Resistant	pattern	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	✓
		500/600	✓	✓	✓	✓	✓	✓	✓	✓
Exponential Weighting	in-distrib.	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	✓
		500/600	✓	✓	✓	✓	✓	✓	✓	✓
Frontier Stitching	perturb.	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	65.83%
		500/600	✓	✓	✓	✓	✓	65.67%	45.33%	45.33%
WM Embedded Systems	pattern	20/24	✓	✓	✓	✓	✓	✓	✓	✓
		100/120	✓	✓	✓	✓	✓	✓	✓	✓
		500/600	✓	✓	✓	✓	✓	✓	✓	✓

7.2.4 Effectiveness

From our implementations and experiments, we can confirm that almost all of the watermarking methods achieve a watermark accuracy of 100%. The detailed results are summarised in Table 7.9. *FrontierStitching* for a large trigger set size tend to lose watermark accuracy on some architectures. The models reach only around 45% and 65% watermark accuracy on the LeNets when trained with 600 trigger images and also on LeNet-5 when trained with only 120 trigger images. All of the other models have at least 99% watermark accuracy, which is very much acceptable.

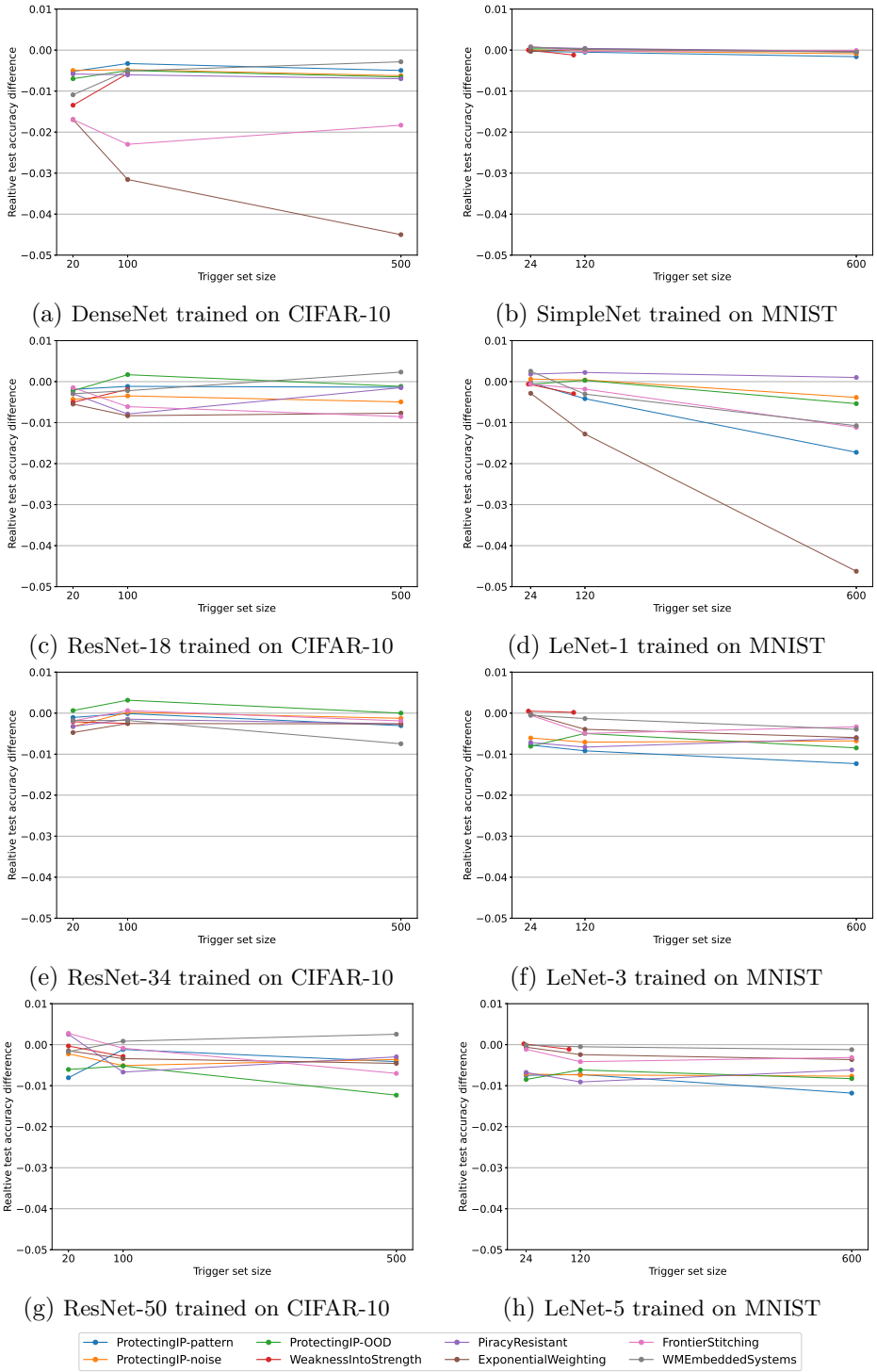
7.2.5 Fidelity

Figure 7.8 shows the relative test accuracy difference as a function of the trigger set size for all watermarking methods for each architecture. From the figure, we can see that there exists a downward trend for LeNet-1, i.e. the larger the trigger set the more we lose on test accuracy.

Interestingly, all methods on SimpleNet perform exceptionally well (cf. Figure 7.8b). We explain this by the complexity of SimpleNet and the large number of "degrees of freedom". SimpleNet is a very complex model for classifying only gray-scaled images, it consists of 51 *times* more trainable parameters than LeNet-5, the largest one of the LeNets, and almost 762 times more than LeNet-1, which shows the strongest drop in fidelity (cf. Table 6.3).

DenseNet performs rather poorly when watermarked with *ExponentialWeighting* and *FrontierStitching* (cf. Figure 7.8a). This could be also explained by the (lack of) complexity of the model: DenseNet has only 30% of the amount of trainable parameters compared to a ResNet-18 (cf. Table 6.3). We can conclude that *ExponentialWeighting*, an in-distribution method, fails on fidelity when used with a smaller model, as we can see it on DenseNet and LeNet-1 (cf. Figures 7.8a and 7.8d).

Figure 7.9 shows the results for fidelity but for each method. This figure confirms the conclusions from above. All methods, except of *ExponentialWeighting* and *Frontier-Stitching*, perform similarly well, i.e. the test accuracy drops for all architectures and trigger set sizes at most for around 1% compared to the benchmark test accuracy. The worst performing models for *ExponentialWeighting* are DenseNet and LeNet-1, with a big influence from the trigger set size, e.g. LeNet-1's test accuracy drops under 0.5% when trained with only 24 trigger images, but over 4% when trained with 600 trigger images. For *FrontierStitching*, only DenseNet performs worse compared to the others with a test accuracy drop of around 2%.



(a) DenseNet trained on CIFAR-10

(b) SimpleNet trained on MNIST

(c) ResNet-18 trained on CIFAR-10

(d) LeNet-1 trained on MNIST

(e) ResNet-34 trained on CIFAR-10

(f) LeNet-3 trained on MNIST

(g) ResNet-50 trained on CIFAR-10

(h) LeNet-5 trained on MNIST

Figure 7.8: Influence of the trigger set size on **fidelity**. Each plot corresponds to one architecture and shows the results for all watermarking methods, on the left models trained on **CIFAR-10** and on the right those trained on **MNIST**. We plot the relative difference between the test accuracy of the watermarked and non-watermarked model.

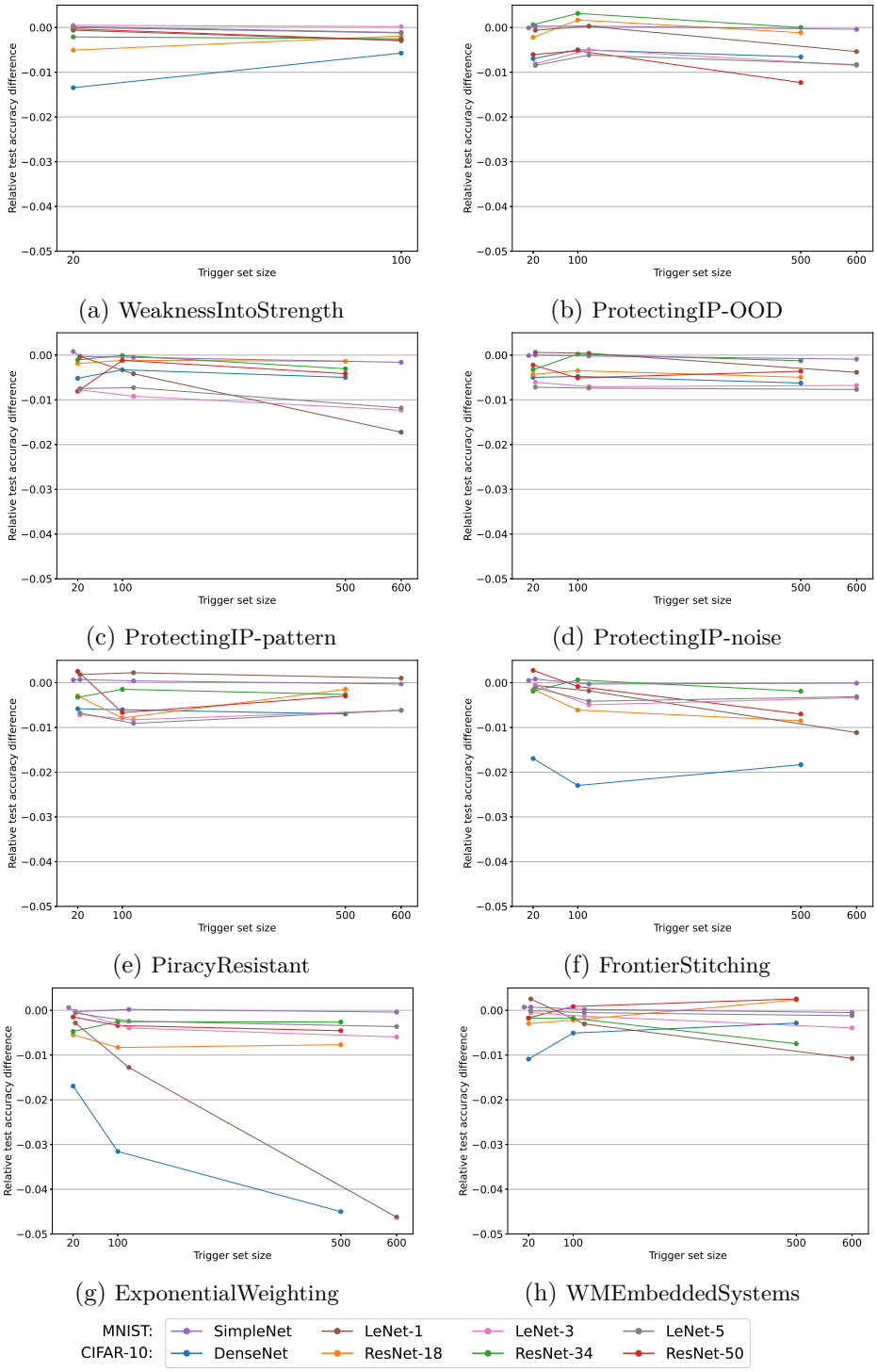


Figure 7.9: Influence of the trigger set size on **fidelity**. Each plot corresponds to one method and shows the results for all architectures. We plot the relative difference between the test accuracy of the watermarked and non-watermarked model.

7.2.6 Robustness

Parameter Pruning

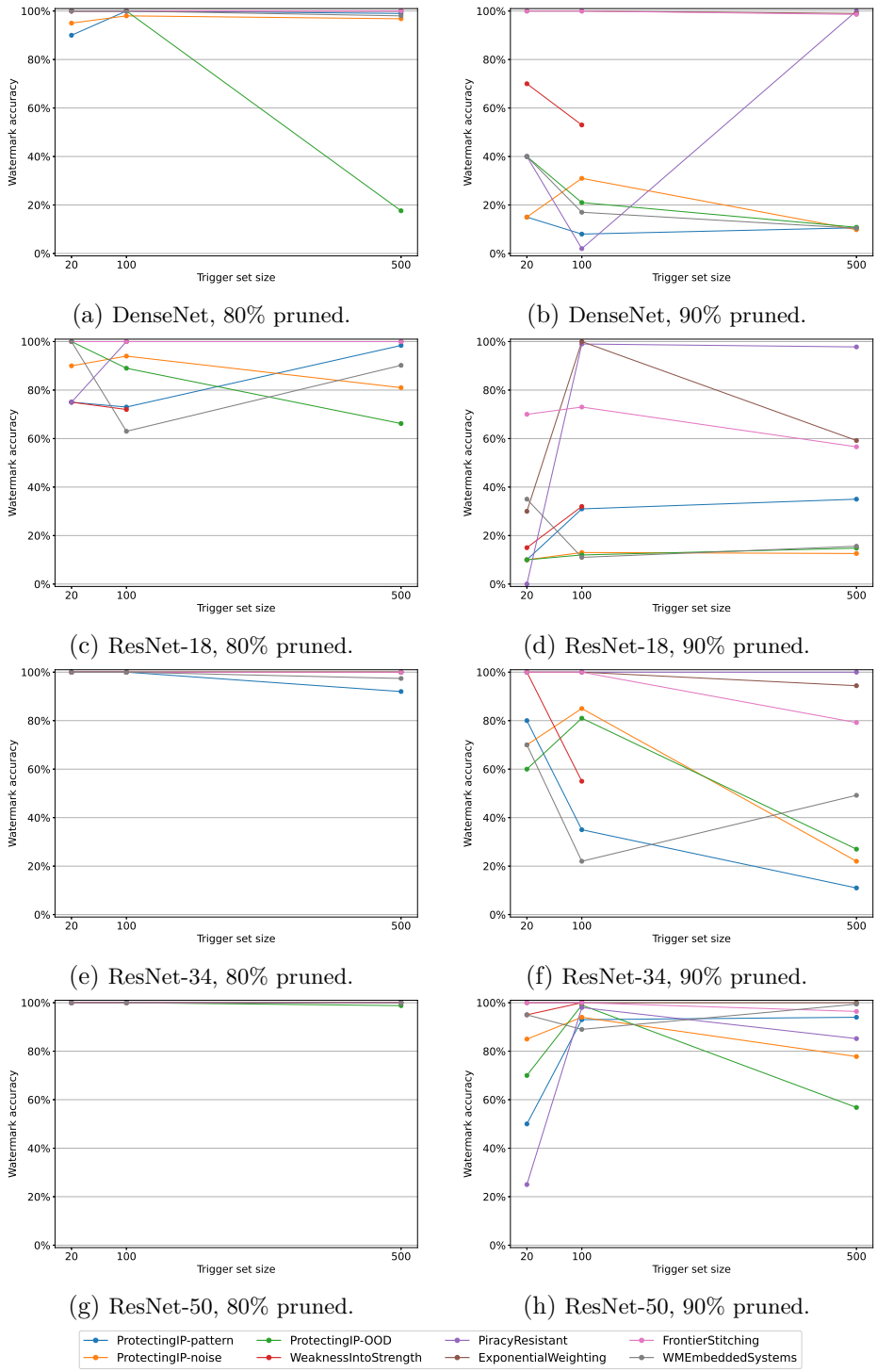
The results for parameter pruning with pruning rates 80% and 90% for CIFAR-10 are shown in Figure 7.10 and for MNIST in Figure 7.11.

The plots show the robustness against pruning for the different sizes of trigger sets for all watermarking methods and each architecture. We clearly see that watermarks in more complex models rather resist a pruning attack with a smaller pruning rate than watermarks embedded into smaller models, as, e.g., ResNet-34 compared with ResNet-18 (cf. Figures 7.10c and 7.10e) or LeNet-3 compared with LeNet-1 (cf. Figures 7.11c and 7.11e).

Also, in some models, e.g. the LeNets, we can see a downward trend for almost all watermarking methods, i.e. a watermark with a smaller trigger set size is more robust (cf. Figures 7.11d, 7.11f and 7.11h). For CIFAR-10 models we do not see such a clear trend – the plots across the watermarking methods do not seem to follow a pattern. Only ResNet-34 shows, for all but one watermarking method, such a trend (cf. Figure 7.10f). This observation is especially interesting, since we did not expect to see a downward trend but rather an upward trend, i.e. more trigger images lead to more robustness, as we would have expected that more trigger images in the training set would embed the watermark stronger into the model. The downward trend could stem from the fact that most of the methods use either a random label for the trigger images, or the next label in the class vector, which means that the model does not learn to predict *one specific* label for a trigger pattern, but rather has to learn combinations of features to predict the right label. For instance, for *ProtectingIP-pattern* a car with the text "TEST" is labelled as "airplane", thus the model has to learn the combination of a car and the pattern in order to predict the right label. Learning such combinations might force the model parameters to adapt to a lot of small parameters instead of a few large ones. A larger trigger set size could then lead to even more of these small parameters, which then, during a pruning attack, are easily set to zero. This is an assumption and could be confirmed by further experiments in future work.

It is worth noting that the *PiracyResistant* method on ResNet-34, SimpleNet and LeNet-3/5 resist even pruning with 90% pruning rate, i.e. the watermark can still be detected with 100% accuracy, for all trigger set sizes. This is probably related to the fact that it is the only method that uses only *one* target label for the trigger images.

Since we do not assume that an attacker would perform a pruning attack regardless of how much the model loses in test accuracy (as the 'value' of the model would then also decrease), we test the watermark accuracy of a model at the maximal plausible pruning rate, as discussed in Section 7.1.2, and show the results in Table 7.10. The values below 50% watermark accuracy are marked in red, and we consider these models as not robust against pruning. However, it depends on the watermark accuracy threshold the model owner sets. Those models without a value in the table are models that performed already



(a) DenseNet, 80% pruned.

(b) DenseNet, 90% pruned.

(c) ResNet-18, 80% pruned.

(d) ResNet-18, 90% pruned.

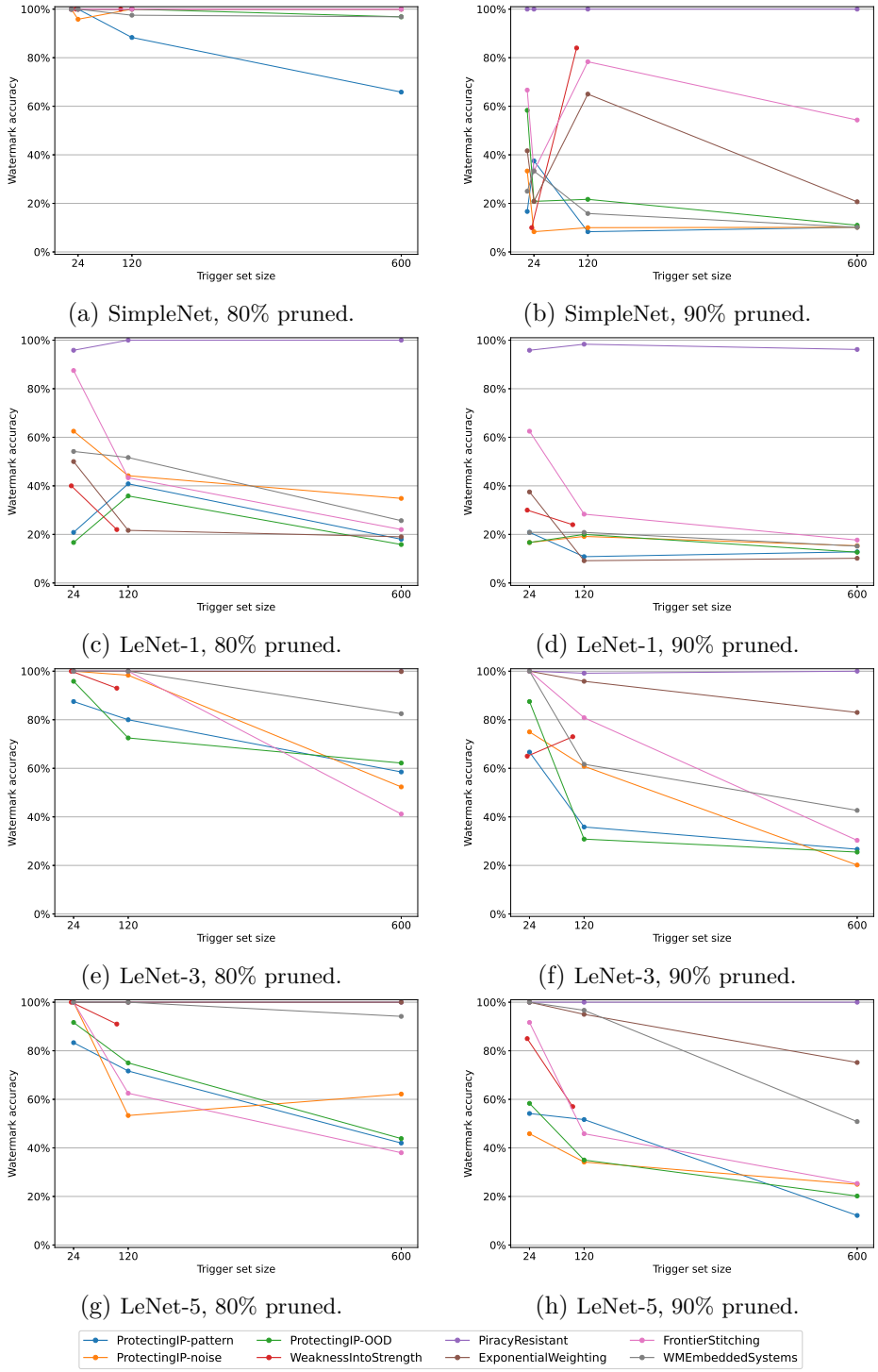
(e) ResNet-34, 80% pruned.

(f) ResNet-34, 90% pruned.

(g) ResNet-50, 80% pruned.

(h) ResNet-50, 90% pruned.

Figure 7.10: Influence of the trigger set size on robustness against pruning on **CIFAR-10** models. Each plot on the left corresponds pruning with 80% and each plot on the right to corresponds pruning with 90%. Each plot shows the results for all watermarking methods.



(a) SimpleNet, 80% pruned.

(b) SimpleNet, 90% pruned.

(c) LeNet-1, 80% pruned.

(d) LeNet-1, 90% pruned.

(e) LeNet-3, 80% pruned.

(f) LeNet-3, 90% pruned.

(g) LeNet-5, 80% pruned.

(h) LeNet-5, 90% pruned.

Figure 7.11: Influence of the trigger set size on robustness against pruning on **MNIST** models. Each plot on the left corresponds pruning with 80% and each plot on the right to corresponds pruning with 90%. Each plot shows the results for all watermarking methods.

Table 7.10: Influence of the trigger set size on robustness against pruning with the maximal plausible pruning rate. The values are the watermark accuracy after an pruning attack, the value in the parenthesis is the maximal plausible pruning rate. A checkmark ✓indicates 100% watermark accuracy.

Method	Type	Trg set size	DenseNet	ResNet-18	ResNet-34	ResNet-50	SimpleNet	LeNet-1	LeNet-3	LeNet-5
ProtectingIP	pattern	20/24	90.0% (0.8)	✓(0.7)	✓(0.8)	✓(0.8)	✓(0.8)	87.5% (0.5)	87.5% (0.8)	83.3% (0.8)
		100/120	✓(0.8)	✓(0.6)	✓(0.8)	✓(0.8)	✓(0.7)	69.2% (0.5)	95.0% (0.7)	89.2% (0.7)
		500/600	99.0% (0.8)	✓(0.6)	92.0% (0.8)	✓(0.8)	✓(0.7)	95.3% (0.2)	76.8% (0.7)	90.0% (0.6)
	noise	20/24	95.0% (0.8)	✓(0.7)	✓(0.8)	✓(0.8)	✓(0.7)	✓(0.6)	✓(0.8)	✓(0.8)
		100/120	98.0% (0.8)	✓(0.7)	✓(0.8)	✓(0.8)	✓(0.8)	63.3% (0.7)	98.3% (0.8)	91.7% (0.7)
		500/600	96.8% (0.8)	✓(0.5)	✓(0.8)	✓(0.8)	✓(0.8)	53.5% (0.6)	72.3% (0.7)	84.0% (0.7)
	OOD	20/24	✓(0.8)	✓(0.8)	✓(0.8)	✓(0.8)	✓(0.8)	79.2% (0.5)	95.8% (0.8)	91.7% (0.8)
		100/120	✓(0.8)	✓(0.7)	✓(0.8)	99.0% (0.9)	✓(0.8)	86.7% (0.5)	90.0% (0.7)	75.0% (0.8)
		500/600	17.6% (0.8)	✓(0.6)	✓(0.8)	98.8% (0.8)	✓(0.7)	41.7% (0.5)	88.7% (0.7)	70.8% (0.7)
Weakness IntoStrength	OOD	20	✓(0.8)	✓(0.7)	✓(0.8)	✓(0.8)	✓(0.8)	45.0% (0.6)	65.0% (0.9)	85.0% (0.9)
		100	✓(0.8)	✓(0.7)	✓(0.8)	✓(0.9)	✓(0.8)	56.0% (0.5)	73.0% (0.9)	57.0% (0.9)
Piracy Resistant	pattern	20/24	✓(0.8)	75.0% (0.8)	✓(0.9)	25.0% (0.9)	✓(0.7)	✓(0.5)	✓(0.8)	✓(0.7)
		100/120	✓(0.8)	✓(0.7)	✓(0.8)	98.0% (0.9)	✓(0.7)	✓(0.6)	99.2% (0.9)	✓(0.7)
		500/600	✓(0.8)	✓(0.8)	✓(0.8)	85.2% (0.9)	✓(0.7)	✓(0.4)	✓(0.8)	✓(0.6)
Exponential Weighting	in-distrib.	20/24	✓(0.9)	✓(0.8)	✓(0.9)	✓(0.9)	✓(0.8)	87.5% (0.5)	✓(0.9)	✓(0.9)
		100/120	✓(0.9)	✓(0.8)	✓(0.9)	✓(0.9)	✓(0.8)	94.2% (0.4)	95.8% (0.9)	95.0% (0.9)
		500/600	-	✓(0.8)	94.4% (0.9)	✓(0.9)	✓(0.8)	-	83.0% (0.9)	✓(0.8)
Frontier Stitching	perturb.	20/24	✓(0.9)	✓(0.8)	✓(0.9)	✓(0.9)	✓(0.8)	87.5% (0.6)	✓(0.9)	91.7% (0.9)
		100/120	✓(0.9)	✓(0.8)	✓(0.9)	✓(0.9)	✓(0.8)	85.0% (0.5)	80.8% (0.9)	45.8% (0.9)
		500/600	98.6% (0.9)	✓(0.8)	79.2% (0.9)	96.4% (0.9)	✓(0.8)	42.5% (0.6)	30.3% (0.9)	25.3% (0.9)
WMEmbedded Systems	pattern	20/24	✓(0.8)	✓(0.7)	✓(0.8)	95.0% (0.9)	✓(0.8)	79.2% (0.7)	✓(0.9)	✓(0.9)
		100/120	✓(0.8)	✓(0.6)	✓(0.8)	✓(0.8)	97.5% (0.8)	83.3% (0.7)	61.7% (0.9)	96.7% (0.9)
		500/600	98.0% (0.8)	✓(0.6)	✓(0.7)	✓(0.8)	✓(0.6)	82.17% (0.5)	82.5% (0.8)	94.2% (0.8)

badly on fidelity, i.e. having more than 3.5% test accuracy drop. Models reaching a 100% watermark accuracy after 90% of the weights are pruned are marked green.

From this table, we conclude that a watermark in a smaller model like LeNet-1 is less robust against pruning than in a more complex model like SimpleNet or ResNet-18. Note that we performed pruning with a step size of 10 percentage points and these results could change when tested with, e.g., a step size of 1 percentage point. We see that those models that fail on pruning are trained with an OOD or perturbation based method, and also ResNet-50 with PiracyResistant and trigger set size 20. However, those perturbation based models that seem to perform badly in this table are also those models that reach only 45-65 % watermark accuracy after the embedding. Therefore, instead of declaring them as not robust against pruning, we rather declare them as primarily not effective.

We can see that the watermarking methods most robust against pruning are *ProtectingIP-pattern*, *ProtectingIP-noise* and *ExponentialWeighting*. For those, the watermark accuracy, after a pruning attack with the maximal plausible pruning rate, stays above 50% for all architectures and all trigger set sizes. For *ExponentialWeighting*, even the worst-performing model has a watermark accuracy of 83%, but two models, DenseNet with the trigger set size 500 and LeNet-1 with the trigger set size 600, did not even qualify for pruning since they already fail the fidelity requirement.

Fine-Tuning

The results for fine-tuning for CIFAR-10 are shown in Figure 7.12 and for MNIST in Figure 7.13.

For fine-tuning with a smaller learning rate, we can see a downward trend for most of the watermarking methods trained on models classifying CIFAR-10 (cf. Figure 7.12). For MNIST models we cannot detect such a trend, neither downward nor upward (cf. Figure 7.13). For fine-tuning with a larger learning rate, all watermarking methods on all architectures for all trigger set sizes fail to resist, the watermark accuracies are mostly below 20%. Again, as discussed above, we did not expect a downward trend regarding robustness but rather an upward trend, i.e. more trigger images lead to more robustness. We discuss this observation in more detail after we analysed the fine-tuning plots per method.

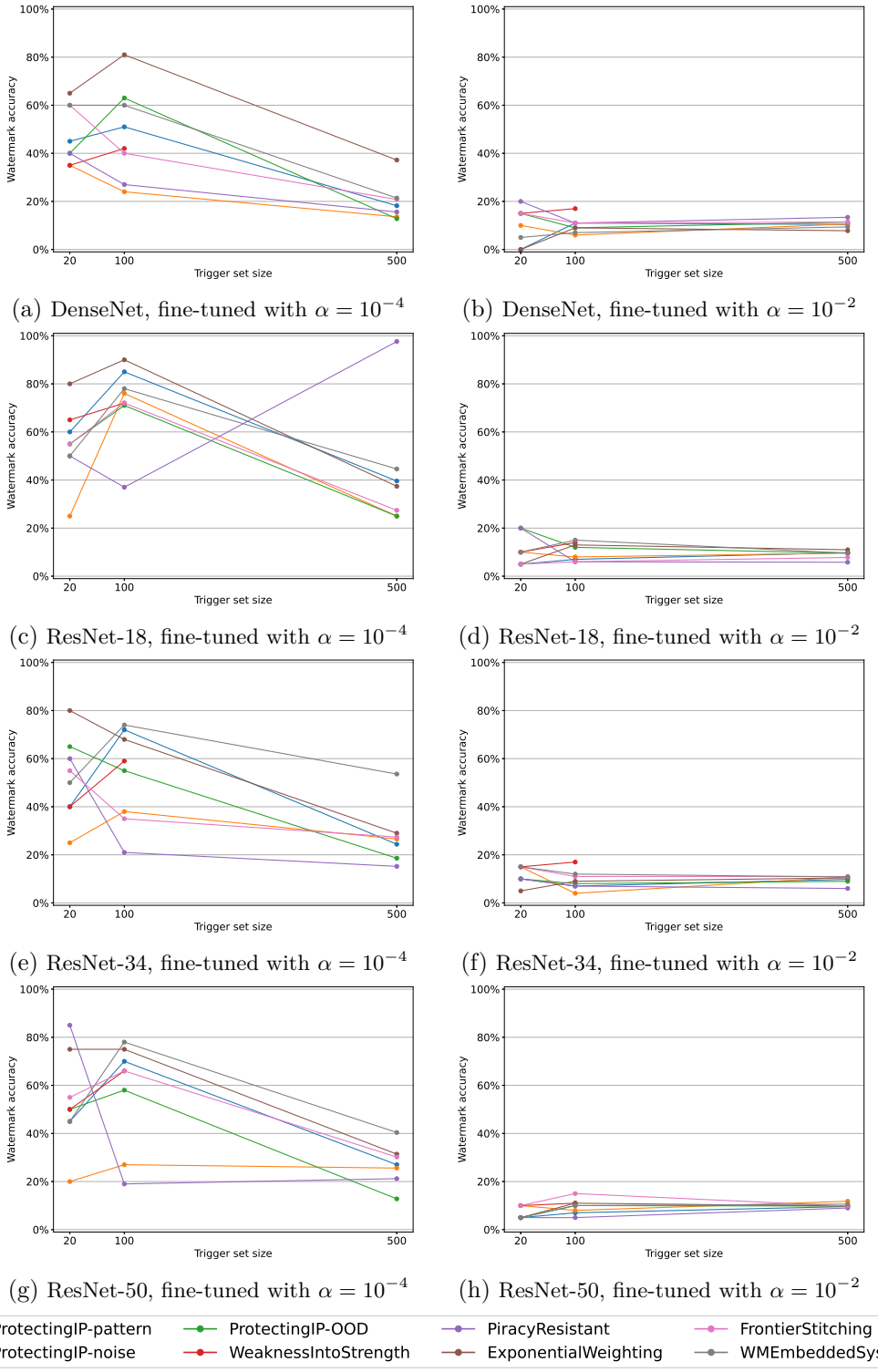
Also, Figure 7.14 shows that, during a fine-tuning attack, the model trained with a larger trigger set size (500) loses watermark accuracy more quickly.

Also here, we can see that SimpleNet performs best for both fine-tuning settings (cf. Figures 7.13a and 7.13b). Overall, the highest watermark accuracies are seen on SimpleNet. Interestingly, for SimpleNet a *larger* trigger set size is beneficial, since models with smaller trigger set sizes lose more watermark accuracy after fine-tuning with a smaller learning rate.

Concluding this section, Figure 7.15 shows the results for fine-tuning with a smaller learning rate ($\alpha = 10^{-4}$ for CIFAR-10 and $\alpha = 10^{-5}$ for MNIST) for all architectures for each watermarking method. Comparing each watermarking method, we can see an upward trend in *WeaknessIntoStrength* (cf. Figure 7.15a) for almost all architectures, but we do see this also for other methods comparing trigger set size 20 and 100. A downward trend can be detected for *ExponentialWeighting*, *FrontierStitching*, *ProtectingIP-pattern* and *ProtectingIP-OOD* (cf. Figures 7.15b, 7.15c, 7.15f and 7.15g). The reason for the downward trend might be the same as for pruning. These methods do use more than one label for the trigger images and therefore the model does not learn the specific trigger "pattern", but overfits on every single trigger image, which the model then easily "forgets" during a fine-tuning attack.

Note that *WeaknessIntoStrength* was trained only with 20 and 100 trigger images and therefore we do not have the same amount of information compared to the other ones. Although *ExponentialWeighting* shows a downward trend, the watermark accuracy for the models trained with 100/120 trigger images are among the highest compared to other methods. We observe the same for *WMEmbeddedSystems*, as the watermark accuracy for those trigger set sizes is above 60%.

Interestingly, MNIST models watermarked with *PiracyResistant* have a watermark accuracy of 100% even after a fine-tuning attack with a small learning rate. This, however, is not the case for CIFAR-10 models, as the watermark accuracy for those models is between 20% and 40% after the fine-tuning attack. As above, this high



(a) DenseNet, fine-tuned with $\alpha = 10^{-4}$

(b) DenseNet, fine-tuned with $\alpha = 10^{-2}$

(c) ResNet-18, fine-tuned with $\alpha = 10^{-4}$

(d) ResNet-18, fine-tuned with $\alpha = 10^{-2}$

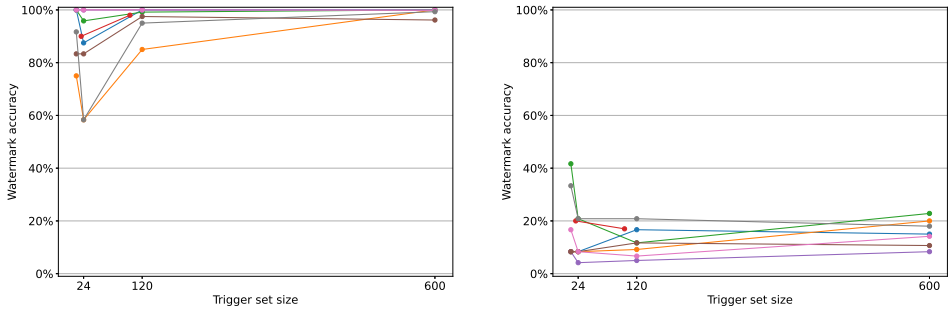
(e) ResNet-34, fine-tuned with $\alpha = 10^{-4}$

(f) ResNet-34, fine-tuned with $\alpha = 10^{-2}$

(g) ResNet-50, fine-tuned with $\alpha = 10^{-4}$

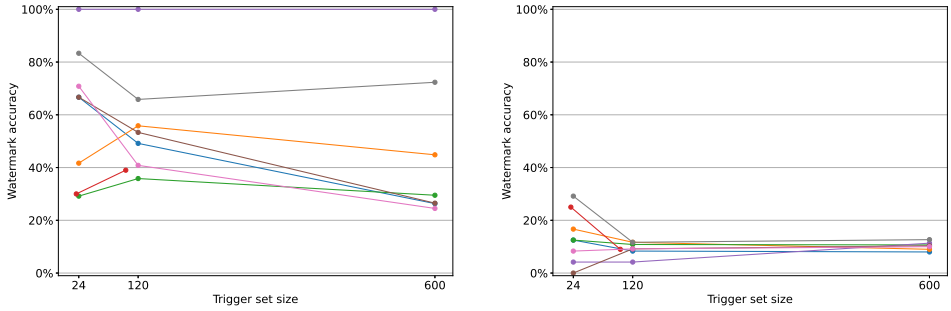
(h) ResNet-50, fine-tuned with $\alpha = 10^{-2}$

Figure 7.12: Influence of the trigger set size on robustness against fine-tuning on **CIFAR-10** models. Each plot on the right corresponds fine-tuning with a small learning rate and each plot on the left to fine-tuning with a large learning rate, all of them show the results for all watermarking methods.



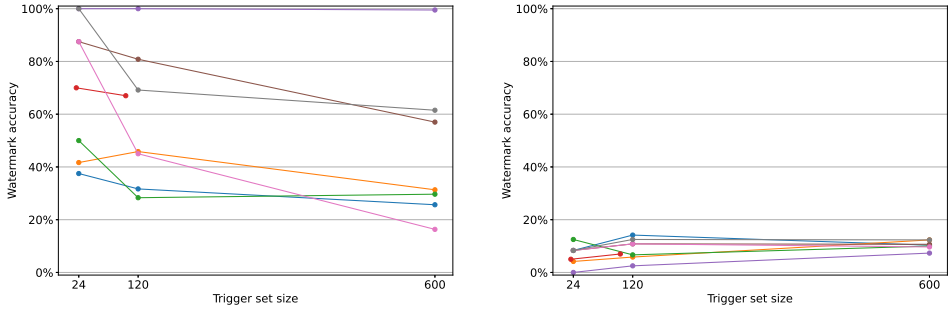
(a) SimpleNet, fine-tuned with $\alpha = 10^{-5}$

(b) SimpleNet, fine-tuned with $\alpha = 10^{-2}$



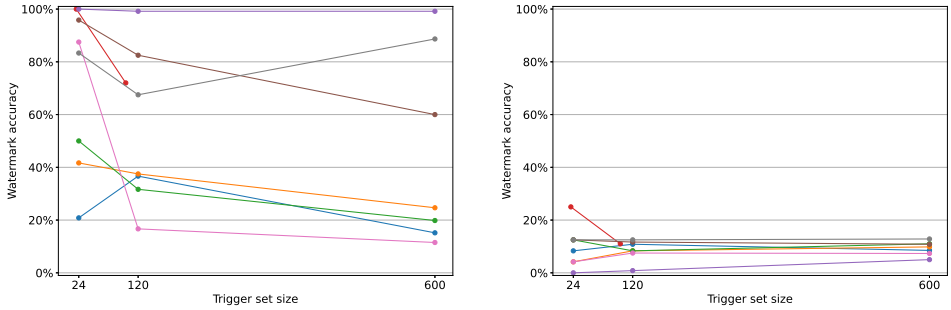
(c) LeNet-1, fine-tuned with $\alpha = 10^{-5}$

(d) LeNet-1, fine-tuned with $\alpha = 10^{-2}$



(e) LeNet-3, fine-tuned with $\alpha = 10^{-5}$

(f) LeNet-3, fine-tuned with $\alpha = 10^{-2}$



(g) LeNet-5, fine-tuned with $\alpha = 10^{-5}$

(h) LeNet-5, fine-tuned with $\alpha = 10^{-2}$

Figure 7.13: Influence of the trigger set size on robustness against fine-tuning on MNIST models. Each plot on the left corresponds fine-tuning with a small learning rate and each plot on the right to fine-tuning with a large learning rate, all of them show the results for all watermarking methods.

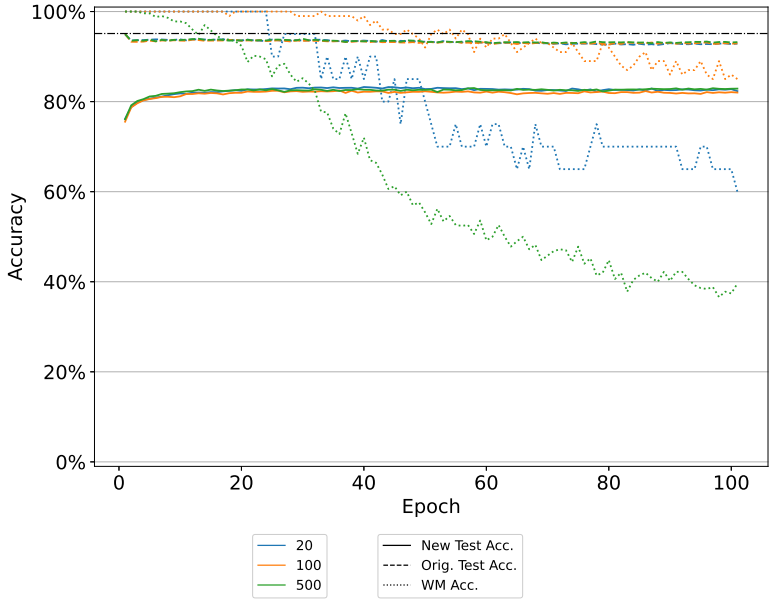
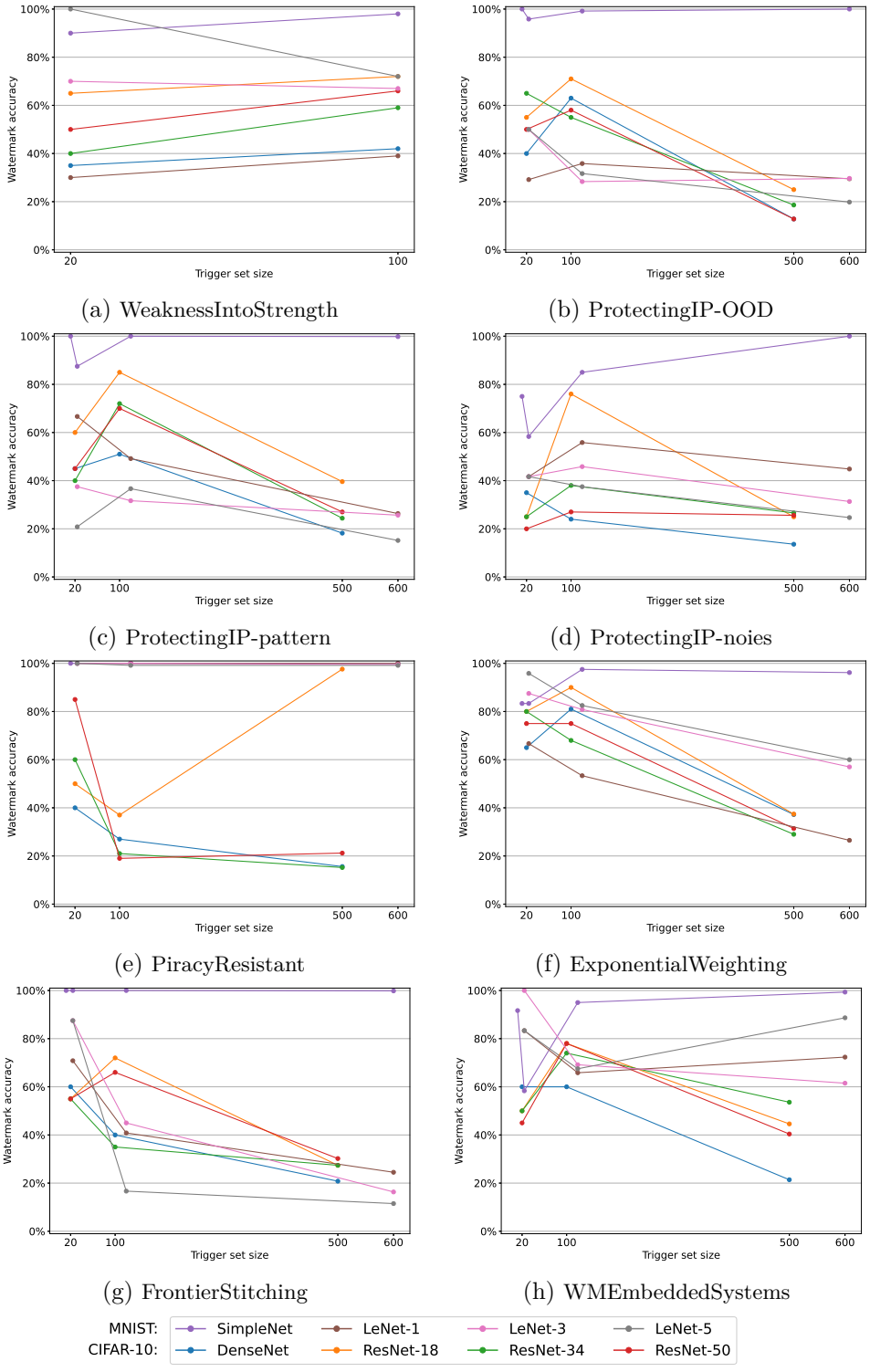


Figure 7.14: Behaviour of ResNet-18 watermarked with *ProtectingIP-pattern* during a fine-tuning attack with a small learning rate $\alpha = 10^{-4}$. The colors indicate the trigger set size, with which the model was watermarked. The black dash-dotted line corresponds to the benchmark test accuracy of the non-watermarked model.

robustness of MNIST models might be related to the fact that *PiracyResistant* uses one label for all trigger images.



(a) WeaknessIntoStrength

(b) ProtectingIP-OOD

(c) ProtectingIP-pattern

(d) ProtectingIP-noies

(e) PiracyResistant

(f) ExponentialWeighting

(g) FrontierStitching

(h) WMEmbeddedSystems

Figure 7.15: Influence of the trigger set size on robustness against fine-tuning with a small learning rate, 10^{-5} for MNIST and 10^{-4} for CIFAR-10. Each plot corresponds to one watermarking method and shows the results for all architectures.

CHAPTER 8

Conclusions and future work

In this thesis, we systematised findings on IP protection (IPP) in machine learning (ML). We developed a comprehensive threat model for IP in ML, and categorised attacks and defences within a unified and consolidated taxonomy.

The focus of this thesis was then an in-depth analysis of backdoor-based watermarking approaches for DNNs for image classification. We defined a common study setting, implemented eight methods, and trained 191 models. The methods were analysed regarding effectiveness, fidelity and robustness based on several experiments with well-known Deep Learning (DL) architectures on widely used benchmark datasets.

We address the research questions from Chapter 6, starting with the subquestions:

To what extend is a more complex model able to hold more watermark information (a bigger trigger set) without compromising test accuracy? We tested fidelity for all watermarking methods and architectures and showed the results in Figure 7.8. We can clearly say, at least for MNIST, a more complex model is able to hold more watermark information without compromising test accuracy, as we see on SimpleNet in Figure 7.8b compared to the other architectures. SimpleNet is a very complex model for classifying grayscaled images (cf. Table 6.3). But also comparing LeNet-1 and LeNet-3/5, we can see that a more complex model is less affected by a bigger trigger set. For models trained on CIFAR-10, however, we do not see a clear trend. The test accuracy difference on ResNet-18/34/50 across the trigger set sizes is more or less the same. Only perturbation based and in-distribution methods perform worse on DenseNet, which could be related to DenseNet's small size. For future work, testing even larger trigger set sizes for CIFAR-10 models could show similar results as for MNIST models, as the trigger set size 500 might be too small to be representative for such a trend.

To what extend does the trigger set size influence the effectiveness, fidelity and robustness of a watermarking method? Regarding effectiveness, we could not

detect any correlation between the trigger set size and effectiveness. All watermarking methods reach 100%, or almost 100%, watermark accuracy, except for the perturbation based method on the LeNets with the highest trigger set size and LeNet-5 with the midsize trigger set, which only have a watermark accuracy between 45% and 65%. However, we would have rather expected larger trigger sizes to have a positive effect on the effectiveness, as the model is trained on more data regarding the watermark. As we observe this odd behaviour only on *FrontierStitching*, this could be related to the perturbation based trigger images and the chosen parameter ϵ. Recall that we tested several values for ϵ and the value that was chosen by the authors ($\epsilon = 0.25$) was the only one that did not lead to 100% effectiveness on our test group (all architectures trained with 100 trigger images). As perturbation based trigger images are very much dependent on the model, one should test several values for ϵ before deciding on one.

Regarding fidelity, as mentioned above, only smaller models trained on MNIST are influenced by the trigger set size.

Regarding robustness against pruning, we can confirm that the trigger set size has some influence on robustness against pruning. For most of the architectures and most of the methods, a bigger trigger set size is less robust. This is something we did not expect. We would have assumed that a model watermarked with a bigger trigger set would be more robust, since the model is trained on more of these trigger images and therefore increases the weights of the neurons classifying those. However, since this is not the case, we believe that a larger trigger set size increases the weights responsible for the original task in order to still be able to classify correctly on the original task, i.e. to cope with the influence of the trigger images.

For fine-tuning with a larger learning rate, no watermarking method could survive this type of attack. The models have a watermark accuracy mostly below 20% after the fine-tuning attack with a larger learning rate. With a smaller learning rate, however, we can again say, as with pruning, that a larger trigger set size leads to a higher watermark accuracy drop. The exception of this trend is SimpleNet, where a higher trigger set size leads to more robustness against fine-tuning, which could be related to SimpleNet's high complexity compared to the classification task.

To what extend does the complexity of the model influence, the effectiveness, fidelity and robustness of the watermarking method? As discussed above, we can confirm that complexity has a (positive) influence on fidelity and robustness, as we see that SimpleNet performs exceptionally well compared to the other MNIST models and also CIFAR-10 models. For CIFAR-10 models, we see that DenseNet, which is the smallest model, performs worse on fidelity compared to the other CIFAR-10 models (cf. Figure 7.8), but only for two methods, namely *FrontierStitching* and *ExponentialWeighting*.

With the insights and answers found in the analysis, we now address the overall research question:

How can we define the most fitting watermarking method depending on the ML setting? It depends on the model owners requirements, regarding effectiveness,

fidelity and robustness, but it seems very likely that a model owner should consider building a more complex model than needed for the original task, in order to get the best results for watermarking, as we have seen it on SimpleNet.

Comparing the different watermarking methods, we now summarise the findings on the performance regarding effectiveness, fidelity and robustness against pruning and fine-tuning:

- **Effectiveness:** We cannot make any recommendation based on the complexity of a model, as almost all models reach 100% watermark accuracy after watermark embedding (cf. Section 7.2.4). Only *FrontierStitching* fails on some models for MNIST.
- **Fidelity:** All models, except of *ExponentialWeighting* on DenseNet and LeNet-1 and *FrontierStitching* on DenseNet, did pass the fidelity requirement, i.e. the test accuracy drops not more than around 1% of the benchmark test accuracy (cf. Section 7.2.5).
- **Robustness against pruning:** The OOD methods, *WeaknessIntoStrength* and *ProtectingIP-OOD*, perform worst and the methods *ProtectingIP-pattern*, *ProtectingIP-noise* and *ExponentialWeighting* perform best (cf. Section 7.2.6). *PiracyResistant* resists a pruning attack exceptionally well on ResNet-34, SimpleNet and LeNet-3/5.
- **Robustness against fine-tuning:** The best performing method over all architectures is *ExponentialWeighting* and *WMEmbeddedSystems*, especially with a trigger set size of 100/120 (cf. Section 7.2.6). The best robustness against pruning for MNIST models did reach *PiracyResistant* with 100% watermark accuracy after the attack.

Depending on the model owner's requirements, one could now choose a fitting watermarking method based on this analysis' summary. Furthermore, we would like to point out three important differences: the time of embedding, the overhead regarding the implementation and the efficiency. As we saw in Section 7.2.3, *WeaknessIntoStrength* performed better when the watermark was embedded from scratch and we believe this holds true also for the other methods. All of the methods use embedding from scratch, except of *FrontierStitching*, for which the watermark generation relies on an already trained model to generate the fitting adversarial examples used as trigger images, and *ExponentialWeighting*, which first has to learn the original task and then, as a second step, gets the watermarks embedded.

Regarding the implementation and training overhead, a simple pattern based or noise based method such as *ProtectingIP-pattern* or *ProtectingIP-noise* would be the simplest method to implement, as it does not utilise any further security measures and the watermark generation is straightforward. More advanced methods would be *WMEmbeddedSystems* or *PiracyResistant*, which use a unique signature for the watermark generation.

For *ExponentialWeighting* the layers of the model need to be changed, as this method applies a special transformation on the weights. Watermarking with *FrontierStitching* likely requires extra effort for experiments before choosing the best value for the parameter ϵ. Also, the OOD methods *WeaknessIntoStrength* and *ProtectingIP-OOD* need another set of data which could lead to additional effort when searching for and preparing the OOD data.

The most efficient watermarking methods are those that embed the watermark from scratch, since embedding and model training is done in the same step. *FrontierStitching* and *ExponentialWeighting* need an already trained model, and therefore are less efficient. *FrontierStitching* needs even more computation, as the creation of adversarial images is quite time-consuming.

For those model owners that are interested in fingerprinting, we believe that the most fitting methods would be *ProtectingIP-pattern*, *WMEmbeddedSystems* and *PiracyResistant*, as the watermark generation can be easily customised, in order to embed unique watermarks for multiple users.

To enable more comparable results, it would be very helpful if authors in future research would evaluate their newly proposed IPP method or attack in a more uniform manner. The usage of state-of-the-art DL architectures and datasets should be common practice, as the results of different methods could be compared directly. When authors use custom DNNs instead of well-known ones, it raises concerns whether this method or attack is transferable to another setting or if the algorithm performed only exceptionally well on the chosen custom DNN. Moreover, results that are only shown in plots are difficult to compare, as the exact values are often not possible to identify from a plot. Therefore, we suggest in future research to always provide results both in a table and plot.

For future work, we would wish to explore more attacks on backdoor-based watermarking methods, e.g. fine-pruning or watermark overwriting, and compare the methods on a larger scale of settings, implementing more architectures and use larger datasets, in order to give even clearer recommendations. Also, a comparison with larger datasets would answer the question of whether the trigger set size should be chosen as a ratio of the original dataset, or as some absolute number. Moreover, a thorough analysis of a watermarked model during a pruning or fine-tuning attack could give clearer answers why in some cases a larger trigger set size leads to less robustness.

Further black-box watermarking methods need to be explored, e.g. methods focusing on trigger labelling. Future work could find out if trigger labelling is not only a useful but necessary feature.

As watermarking methods are already quite well studied on image classification with DNNs, it would be useful to also develop concepts for other ML techniques and especially real-world problems. This, however, would need a comprehensive study of different industries and their use cases in order to follow the needs and wishes of real-world problems.

APPENDIX A

Appendix

A.1 Dependencies

A complete list of python packages used for the implementation and experiments:

- Babel version 2.9.1
- kmeans_pytorch version 0.3
- matplotlib version 3.3.4
- numpy version 1.20.2
- pandas version 1.2.4
- Pillow version 8.2.0
- rsa version 4.7.2
- torch version 1.8.1
- torchvision version 0.9.1

A.2 Additional figures

A.2.1 Experiments for fine-tuning

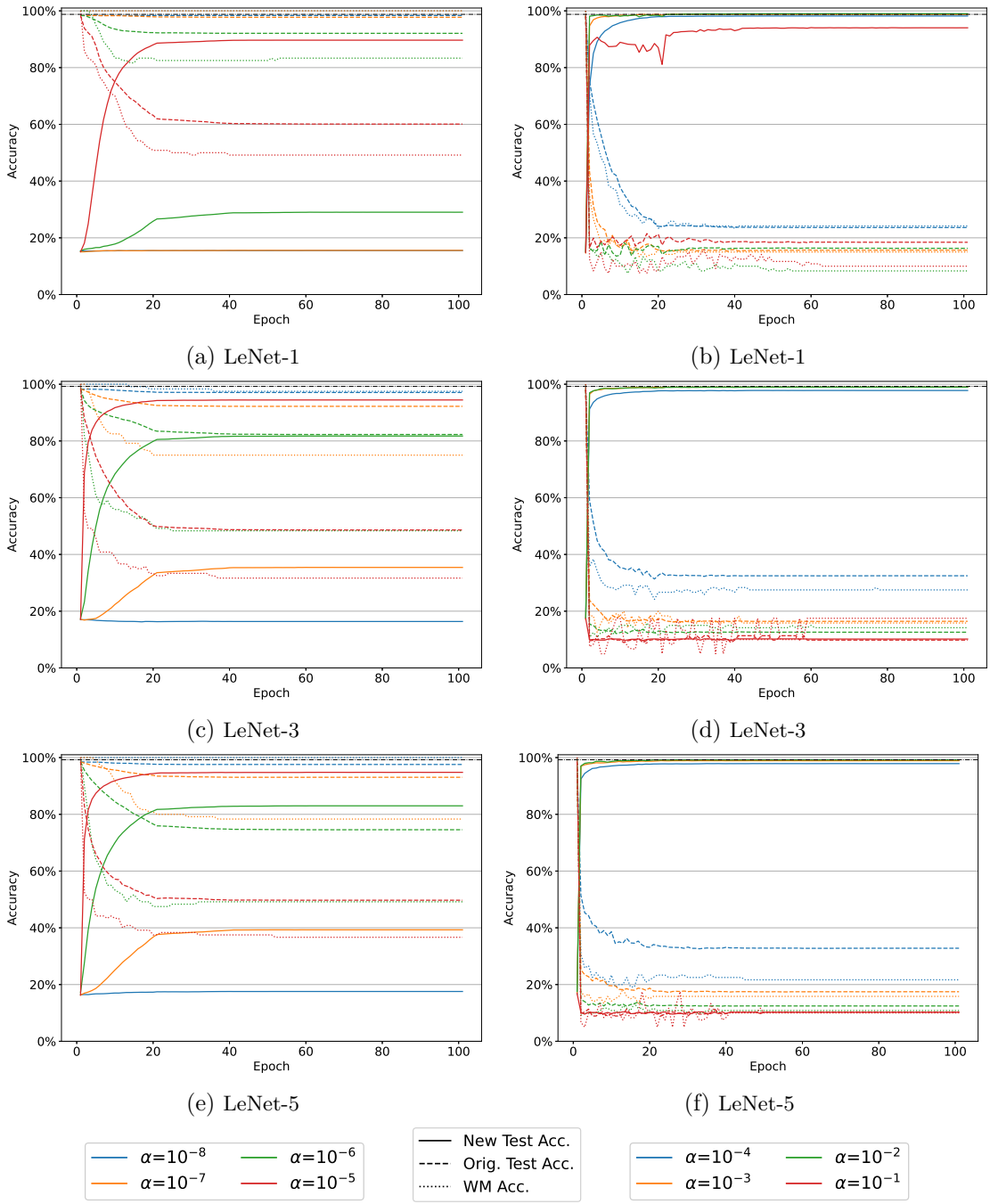
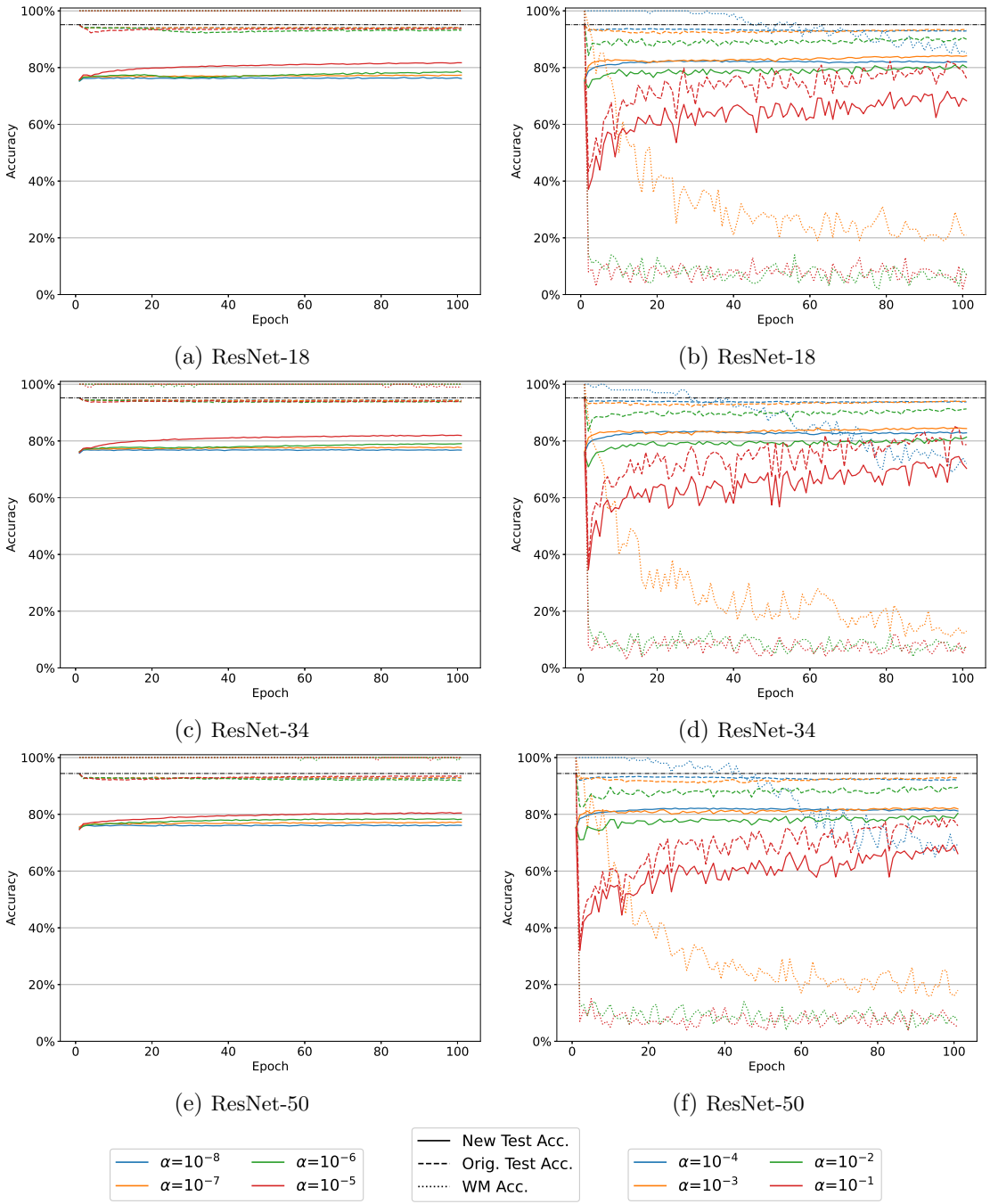


Figure A.1: Fine-tuning on **MNIST** models, watermarked with *ProtectingIP-pattern*. The plots on the left side correspond to fine-tuning with smaller learning rates and the ones on the right side to fine-tuning with larger learning rates. The black dash-dotted line corresponds to the benchmark test accuracy of the non-watermarked model.

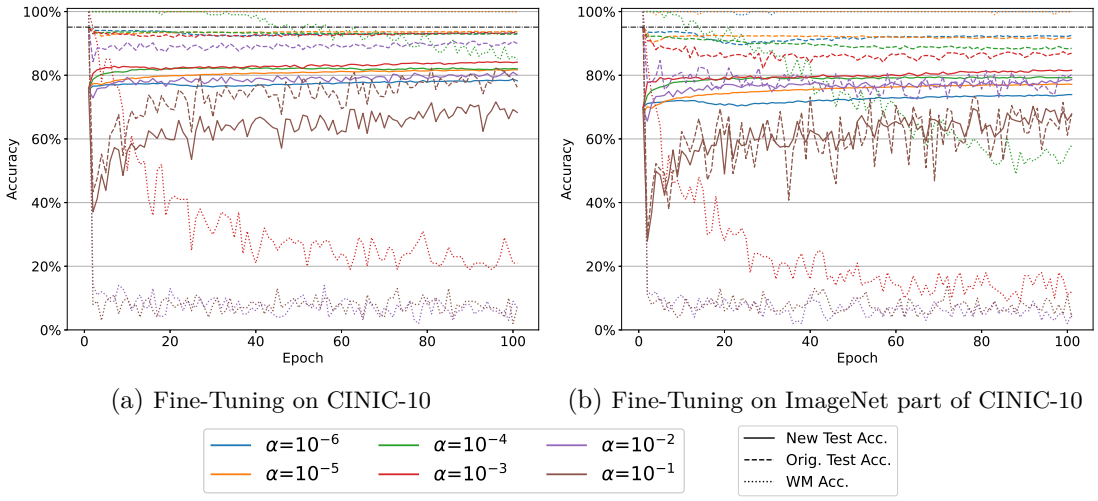


(a) ResNet-18 (b) ResNet-18

(c) ResNet-34 (d) ResNet-34

(e) ResNet-50 (f) ResNet-50

Figure A.2: Fine-tuning on **CIFAR-10** models, watermarked with *ProtectingIP-pattern*. The plots on the left side correspond to fine-tuning with smaller learning rates and the ones on the right side to fine-tuning with larger learning rates. The black dash-dotted line corresponds to the benchmark test accuracy of the non-watermarked model.



(a) Fine-Tuning on CINIC-10 (b) Fine-Tuning on ImageNet part of CINIC-10

Figure A.3: Fine-Tuning on both, CINIC-10 and only on the ImageNet part of CINIC-10. Original line plots to Figure 7.2.

List of Figures

List of Tables

Bibliography

[1] GreyNet. http://www.greynet.org/greysourceindex/documenttypes.html. Accessed: 2020-11-20.

[2] Model Zoo. https://modelzoo.co/. Accessed: 2020-10-16.

[3] Sahar Abdelnabi and Mario Fritz. Adversarial Watermarking Transformer: Towards Tracing Text Provenance with Data Hiding, September 2020. arXiv: 2009.03015. URL: http://arxiv.org/abs/2009.03015.

[4] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In *27th USENIX Security Symposium*, SEC'18, pages 1615–1631, Baltimore, USA, August 2018. USENIX Association.

[5] William Aiken, Hyoungshick Kim, and Simon Woo. Neural Network Laundering: Removing Black-Box Backdoor Watermarks from Deep Neural Networks, April 2020. arXiv: 2004.11368. URL: http://arxiv.org/abs/2004.11368.

[6] Manaar Alam, Sayandeep Saha, Debdeep Mukhopadhyay, and Sandip Kundu. Deep-Lock: Secure Authorization for Deep Neural Networks, August 2020. arXiv: 2008.05966. URL: http://arxiv.org/abs/2008.05966.

[7] Khaled Almezhghwi and Sertan Serte. Improved classification of white blood cells with the generative adversarial network and deep convolutional neural network. *Computational Intelligence and Neuroscience*, 2020, July 2020. doi:10.1155/2020/6490479.

[8] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[9] MaungMaung AprilPyone and Hitoshi Kiya. Training DNN Model with Secret Key for Model Protection, August 2020. arXiv: 2008.02450. URL: http://arxiv.org/abs/2008.02450.

[10] Buse Gul Atli, Yuxi Xia, Samuel Marchal, and N. Asokan. WAFFLE: Watermarking in Federated Learning, August 2020. arXiv: 2008.07298. URL: http://arxiv.org/abs/2008.07298.

[11] Vahid Behzadan and William Hsu. Sequential Triggers for Watermarking of Deep Reinforcement Learning Policies, June 2019. arXiv: 1906.01126. URL: http://arxiv.org/abs/1906.01126.

[12] Franziska Boenisch. A Survey on Model Watermarking Neural Networks, September 2020. arXiv: 2009.12153. URL: http://arxiv.org/abs/2009.12153.

[13] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[14] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. IPGuard: Protecting Intellectual Property of Deep Neural Networks via Fingerprinting the Classification Boundary, April 2020. arXiv: 1910.12903. URL: http://arxiv.org/abs/1910.12903.

[15] Abhishek Chakraborty, Ankit Mondal, and Ankur Srivastava. Hardware-Assisted Intellectual Property Protection of Deep Learning Models. In *57th ACM/IEEE Annual Design Automation Conference 2020*, pages 1–6. IEEE, June 2020. URL: https://eprint.iacr.org/2020/1016.pdf.

[16] Huili Chen, Bita Darvish, and Farinaz Koushanfar. SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems. In *Interspeech 2020*, pages 2312–2316. ISCA, October 2020. doi:10.21437/Interspeech.2020-2787.

[17] Huili Chen, Bita Darvish Rouhani, Xinwei Fan, Osman Cihan Kilinc, and Farinaz Koushanfar. Performance Comparison of Contemporary DNN Watermarking Techniques, November 2018. arXiv: 1811.03713. URL: http://arxiv.org/abs/1811.03713.

[18] Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models. In *International Conference on Multimedia Retrieval*, ICMR '19, pages 105–113, Ottawa, Canada, 2019. ACM. doi:10.1145/3323873.3325042.

[19] Huili Chen, Bita Darvish Rouhani, and Farinaz Koushanfar. BlackMarks: Blackbox Multibit Watermarking for Deep Neural Networks, March 2019. arXiv: 1904.00344. URL: http://arxiv.org/abs/1904.00344.

[20] Mingliang Chen and Min Wu. Protect Your Deep Neural Networks from Piracy. In *IEEE International Workshop on Information Forensics and Security*, WIFS '18, pages 1–7, Hong Kong, Hong Kong, 2018. IEEE. doi:10.1109/WIFS.2018.8630791.

[21] Xinyun Chen, Wenxiao Wang, Chris Bender, Yiming Ding, Ruoxi Jia, Bo Li, and Dawn Song. REFIT: a Unified Watermark Removal Framework for Deep

Learning Systems with Limited Data, January 2020. arXiv: 1911.07205. URL: http://arxiv.org/abs/1911.07205.

[22] Xinyun Chen, Wenxiao Wang, Yiming Ding, Chris Bender, Ruoxi Jia, Bo Li, and Dawn Song. Leveraging Unlabeled Data for Watermark Removal of Deep Neural Networks. In *ICML Workshop on Security and Privacy of Machine Learning*, June 2019.

[23] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, September 2014. arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[24] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters, March 2017. arXiv: 1702.05373. URL: http://arxiv.org/abs/1702.05373.

[25] Luke N. Darlow, Elliot J. Crowley, Antreas Antoniou, and Amos J. Storkey. CINIC-10 is not ImageNet or CIFAR-10, October 2018. arXiv: 1810.03505. URL: http://arxiv.org/abs/1810.03505.

[26] Peter Eckersley. How Unique Is Your Web Browser? In *Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer Berlin Heidelberg, 2010. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-642-14527-8_1, doi:10.1007/978-3-642-14527-8_1.

[27] Lixin Fan, Kam Woh Ng, and Chee Seng Chan. Rethinking Deep Neural Network Ownership Verification: Embedding Passports to Defeat Ambiguity Attacks. In *33rd International Conference on Neural Information Processing Systems*, volume 32, pages 4714–4723. Neural information processing systems foundation, December 2019.

[28] Le Feng and Xinpeng Zhang. Watermarking Neural Network with Compensation Mechanism. In Gang Li, Heng Tao Shen, Ye Yuan, Xiaoyang Wang, Huawen Liu, and Xiang Zhao, editors, *Knowledge Science, Engineering and Management*, volume 12275, pages 363–375. Springer International Publishing, August 2020. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-030-55393-7_33, doi:10.1007/978-3-030-55393-7_33.

[29] Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, 1999. doi:10.1023/A:1007662407062.

[30] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations. In *ACM SIGSAC Conference on Computer and Communications*

Security, CCS'18, pages 619–633, Toronto, Canada, October 2018. ACM. doi:10.1145/3243734.3243834.

[31] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, page 580–587, USA, 2014. IEEE Computer Society. doi:10.1109/CVPR.2014.81.

[32] Laurent Gomez, Marcus Wilhelm, José Márquez, and Patrick Duverger. Security for Distributed Deep Neural Networks: Towards Data Confidentiality & Intellectual Property Protection. In *16th International Joint Conference on e-Business and Telecommunications*, pages 439–447, Prague, Czech Republic, July 2019. SCITEPRESS - Science and Technology Publications. doi:10.5220/0007922404390447.

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[34] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *27th International Conference on Neural Information Processing Systems*, volume 2, pages 2671–2680, Montreal, Canada, 2014. MIT Press.

[35] Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks, March 2015. arXiv: 1312.6211. URL: http://arxiv.org/abs/1312.6211.

[36] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples, 2015. arXiv: 1412.6572. URL: http://arxiv.org/abs/1412.6572.

[37] Patrick Grother. NIST special database 19. NIST handprinted forms and characters database. 1970.

[38] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Evaluating Backdooring Attacks on Deep Neural Networks. *IEEE Access*, 7:47230–47244, 2019. doi:10.1109/ACCESS.2019.2909068.

[39] Xiquan Guan, Huamin Feng, Weiming Zhang, Hang Zhou, Jie Zhang, and Nenghai Yu. Reversible Watermarking in Deep Convolutional Neural Networks for Integrity Authentication. In *28th ACM International Conference on Multimedia*, MM '20, pages 2273–2280, Seattle, USA, October 2020. ACM. doi:10.1145/3394171.3413729.

[40] Jia Guo and Miodrag Potkonjak. Watermarking deep neural networks for embedded systems. In *International Conference on Computer-Aided Design*, ICCAD '18, San Diego, California, November 2018. ACM. doi:10.1145/3240765.3240862.

[41] Jia Guo and Miodrag Potkonjak. Evolutionary Trigger Set Generation for DNN Black-Box Watermarking, June 2019. arXiv: 1906.04411. URL: http://arxiv.org/abs/1906.04411.

[42] Shangwei Guo, Tianwei Zhang, Han Qiu, Yi Zeng, Tao Xiang, and Yang Liu. The Hidden Vulnerability of Watermarking for Deep Neural Networks, September 2020. arXiv: 2009.08697. URL: http://arxiv.org/abs/2009.08697.

[43] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, February 2016. arXiv: 1510.00149. URL: http://arxiv.org/abs/1510.00149.

[44] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks, October 2015. arXiv: 1506.02626. URL: http://arxiv.org/abs/1506.02626.

[45] Seyyed Hossein Hasanpour, Mohammad Rouhani, Mohsen Fayyaz, and Mohammad Sabokrou. Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures, February 2018. arXiv: 1608.06037. URL: http://arxiv.org/abs/1608.06037.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *IEEE International Conference on Computer Vision*, ICCV '15, pages 1026–1034, Santiago, Chile, December 2015. IEEE. doi:10.1109/ICCV.2015.123.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, pages 770–778, Las Vegas, USA, June 2016. IEEE. doi:10.1109/CVPR.2016.90.

[48] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network, March 2015. arXiv: 1503.02531. URL: http://arxiv.org/abs/1503.02531.

[49] Dorjan Hitaj, Briland Hitaj, and Luigi V. Mancini. Evasion Attacks Against Watermarking Techniques found in MLaaS Systems. In *6th International Conference on Software Defined Systems (SDS)*, pages 55–63, Rome, Italy, June 2019. IEEE. doi:10.1109/SDS.2019.8768572.

[50] Dorjan Hitaj and Luigi V. Mancini. Have You Stolen My Model? Evasion Attacks Against Deep Neural Network Watermarking Techniques, September 2018. arXiv: 1809.00615. URL: http://arxiv.org/abs/1809.00615.

[51] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi:10.1162/neco.1997.9.8.1735.

[52] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '17, pages 4700–4708, Honolulu, USA, 2017. IEEE. URL: https://doi.org/10.1109/CVPR.2017.243.

[53] Hengrui Jia, Christopher A. Choquette-Choo, and Nicolas Papernot. Entangled Watermarks as a Defense against Model Extraction, February 2020. arXiv: 2002.12200. URL: http://arxiv.org/abs/2002.12200.

[54] A B Kahng, J Lach, W H Mangione-Smith, S Mantik, I L Markov, M Potkonjak, P Tucker, H Wang, and G Wolfe. Watermarking Techniques for Intellectual Property Protection. In *35th Annual Design Automation Conference*, DAC '98, pages 776–781, San Francisco, USA, 1998. doi:10.1145/277044.277240.

[55] Muhammad Kamran and Muddassar Farooq. A Comprehensive Survey of Watermarking Relational Databases Research, January 2018. arXiv: 1801.08271. URL: https://arxiv.org/abs/1801.08271.

[56] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations*, ICLR '15, San Diego, USA, 2015. arXiv: 1412.6980.

[57] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, March 2017. doi:10.1073/pnas.1611835114.

[58] Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering, 2007.

[59] T. Kohno, A. Broido, and K.C. Claffy. Remote Physical Device Fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, February 2005. doi:10.1109/TDSC.2005.26.

[60] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images, 2009.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84–90, June 2017. doi:10.1145/3065386.

[62] J. Lach, W.H. Mangione-Smith, and M. Potkonjak. FPGA fingerprinting techniques for protecting intellectual property. In *IEEE Custom Integrated Circuits Conference*,

pages 299–302, Santa Clara, USA, 1998. IEEE. URL: http://ieeexplore.ieee.org/document/694986/, doi:10.1109/CICC.1998.694986.

[63] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. doi:10.1109/5.726791.

[64] Huiying Li, Emily Wenger, Ben Y. Zhao, and Haitao Zheng. Piracy Resistant Watermarks for Deep Neural Networks, February 2020. arXiv: 1910.01226. URL: http://arxiv.org/abs/1910.01226.

[65] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to prove your model belongs to you: a blind-watermark based framework to protect intellectual property of DNN. In *35th Annual Computer Security Applications Conference*, pages 126–137, San Juan, Puerto Rico, December 2019. ACM. URL: https://dl.acm.org/doi/10.1145/3359789.3359801, doi:10.1145/3359789.3359801.

[66] Jian Han Lim, Chee Seng Chan, Kam Woh Ng, Lixin Fan, and Qiang Yang. Protect, Show, Attend and Tell: Image Captioning Model with Ownership Protection, August 2020. arXiv: 2008.11009. URL: http://arxiv.org/abs/2008.11009.

[67] Ning Lin, Xiaoming Chen, Hang Lu, and Xiaowei Li. Chaotic Weights: A Novel Approach to Protect Intellectual Property of Deep Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1327–1339, August 2020. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. doi:10.1109/TCAD.2020.3018403.

[68] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks. In *Research in Attacks, Intrusions, and Defenses*, pages 273–294. Springer International Publishing, 2018. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-030-00470-5_13, doi:10.1007/978-3-030-00470-5_13.

[69] Xuankai Liu, Fengting Li, Bihan Wen, and Qi Li. Removing Backdoor-Based Watermarks in Neural Networks with Limited Data, August 2020. arXiv: 2008.00407. URL: http://arxiv.org/abs/2008.00407.

[70] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into Transferable Adversarial Examples and Black-Box Attacks, 2017. arXiv: 1611.02770. URL: http://arxiv.org/abs/1611.02770.

[71] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning Attack on Neural Networks, 2017.

[72] Nils Lukas, Yuxuan Zhang, and Florian Kerschbaum. Deep Neural Network Fingerprinting by Conferrable Adversarial Examples, February 2020. arXiv: 1912.00888. URL: http://arxiv.org/abs/1912.00888.

[73] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *30th International Conference on Machine Learning*, volume 28 of *ICML '13*, Atlanta, USA, 2013. PMLR.

[74] Quenby Mahood, Dwayne Van Eerd, and Emma Irvin. Searching for grey literature for systematic reviews: challenges and benefits. *Research Synthesis Methods*, 5(3):221–234, September 2014. URL: doi.org/10.1002/jrsm.1106, doi:10.1002/jrsm.1106.

[75] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. Adversarial Frontier Stitching for Remote Neural Network Watermarking. *Neural Computing and Applications*, 32(13):9233–9244, August 2019. doi:10.1007/s00521-019-04434-z.

[76] Tom M Mitchell. *Machine Learning*, volume 45. McGraw-Hill Education Ltd, 1997.

[77] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. Adaptive computation and machine learning. MIT Press, 2 edition, 2018.

[78] Itay Mosafi, Eli Omid David, and Nathan S. Netanyahu. Stealing Knowledge from Protected Deep Neural Networks Using Composite Unlabeled Data. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Budapest, Hungary, July 2019. IEEE. ISSN: 2161-4407. doi:10.1109/IJCNN.2019.8851798.

[79] Yuki Nagai, Yusuke Uchida, Shigeyuki Sakazawa, and Shin'ichi Satoh. Digital watermarking for deep neural networks. *International Journal of Multimedia Information Retrieval*, 7(1):3–16, March 2018. doi:10.1007/s13735-018-0147-1.

[80] Ryota Namba and Jun Sakuma. Robust Watermarking of Neural Network with Exponential Weighting. In *ACM Asia Conference on Computer and Communications Security*, ASIACCS '19, pages 228–240, Auckland, New Zealand, July 2019. ACM. doi:10.1145/3321705.3329808.

[81] Sinno Jialin Pan and Qiang Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010. doi:10.1109/TKDE.2009.191.

[82] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks against Machine Learning. In *ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 506–519, Abu Dhabi, United Arab Emirates, April 2017. ACM. URL: https://dl.acm.org/doi/10.1145/3052973.3053009, doi:10.1145/3052973.3053009.

[83] Michael Phi. Illustrated Guide to LSTM's and GRU's: A step by step explanation. https://towardsdatascience.com/illustrated-guide-to-lstms-

and-gru-s-a-step-by-step-explanation-44e9eb85bf21, September 2018. Accessed: 2021-08-31.

[84] V.M. Potdar, S. Han, and E. Chang. A survey of digital image watermarking techniques. In *3rd IEEE International Conference on Industrial Informatics*, pages 709–716, Perth, Australia, 2005. IEEE. doi:10.1109/INDIN.2005.1560462.

[85] Yuhui Quan, Huan Teng, Yixin Chen, and Hui Ji. Watermarking Deep Neural Networks in Image Processing. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1852–1865, May 2020. doi:10.1109/TNNLS.2020.2991378.

[86] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. DeepSigns: An End-to-End Watermarking Framework for Ownership Protection of Deep Neural Networks. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 485–497, Providence, USA, April 2019. ACM. URL: https://dl.acm.org/doi/10.1145/3297858.3304051, doi:10.1145/3297858.3304051.

[87] David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. doi:10.1038/323533a0.

[88] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, December 2015. doi:10.1007/s11263-015-0816-y.

[89] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. Radioactive data: tracing through training, February 2020. arXiv: 2002.00937. URL: http://arxiv.org/abs/2002.00937.

[90] Masoumeh Shafieinejad, Jiaqi Wang, Nils Lukas, Xinda Li, and Florian Kerschbaum. On the Robustness of the Backdoor-based Watermarking in Deep Neural Networks, November 2019. arXiv: 1906.07745. URL: http://arxiv.org/abs/1906.07745.

[91] Sai Shyam Sharma and V. Chandrasekaran. A robust hybrid digital watermarking technique against a powerful CNN-based adversarial attack. *Multimedia Tools and Applications*, 79(43-44):32769–32790, November 2020. doi:10.1007/s11042-020-09555-5.

[92] Dan Shewan. 10 Companies Using Machine Learning in Cool Ways. https://www.wordstream.com/blog/ws/2017/07/28/machine-learning-applications. Accessed: 2021-07-26.

[93] Vladislav Skripniuk, Ning Yu, Sahar Abdelnabi, and Mario Fritz. Black-Box Watermarking for Generative Adversarial Networks, August 2020. arXiv: 2007.08457. URL: http://arxiv.org/abs/2007.08457.

[94] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine Learning Models that Remember Too Much. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 587–601, Dallas, USA, October 2017. ACM. doi:10.1145/3133956.3134077.

[95] Rainer Storn. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of global optimization*, 11:341–359, 1997.

[96] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations*, ICLR '14, Banff, Canada, April 2014.

[97] Kálmán Szentannai, Jalal Al-Afandi, and András Horváth. Preventing Neural Network Weight Stealing via Network Obfuscation. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *SAI 2020: Intelligent Computing*, volume 1230 of *AISC*, pages 1–11. Springer International Publishing, July 2020. preprint on arXiv: 1907.01650. doi:10.1007/978-3-030-52243-8_1.

[98] Sebastian Szyller, Buse Gul Atli, Samuel Marchal, and N. Asokan. DAWN: Dynamic Adversarial Watermarking of Neural Networks, June 2020. arXiv: 1906.00830. URL: http://arxiv.org/abs/1906.00830.

[99] Ruixiang Tang, Mengnan Du, and Xia Hu. Deep Serial Number: Computational Watermarking for DNN Intellectual Property Protection, November 2020. arXiv: 2011.08960. URL: http://arxiv.org/abs/2011.08960.

[100] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium*, pages 601–618, Austin, USA, August 2016. USENIX Association.

[101] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding Watermarks into Deep Neural Networks. In *ACM on International Conference on Multimedia Retrieval*, ICMR '17, pages 269–277, Bucharest, Romania, June 2017. ACM. doi:10.1145/3078971.3078974.

[102] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks. In *IEEE Symposium on Security and Privacy*, S&P '19, pages 707–723, San Francisco, USA, May 2019. IEEE. doi:10.1109/SP.2019.00031.

[103] Jiangfeng Wang, Hanzhou Wu, Xinpeng Zhang, and Yuwei Yao. Watermarking in Deep Neural Networks via Error Back-propagation. In *IS&T International Symposium on Electronic Imaging 2020*, volume 2020. Society for Imaging Science and Technology, January 2020. doi:10.2352/ISSN.2470-1173.2020.4.MWSF-022.

[104] Tianhao Wang and Florian Kerschbaum. Attacks on Digital Watermarks for Deep Neural Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '19, pages 2622–2626, Brighton, United Kingdom, April 2019. IEEE. doi:10.1109/ICASSP.2019.8682202.

[105] Tianhao Wang and Florian Kerschbaum. RIGA: Covert and Robust White-Box Watermarking of Deep Neural Networks, October 2020. arXiv: 1910.14268 [v3]. URL: http://arxiv.org/abs/1910.14268.

[106] Tianhao Wang and Florian Kerschbaum. Robust and Undetectable White-Box Watermarks for Deep Neural Networks, March 2020. arXiv: 1910.14268 [v2]. URL: http://arxiv.org/abs/1910.14268.

[107] Hanzhou Wu, Gen Liu, Yuwei Yao, and Xinpeng Zhang. Watermarking Neural Networks with Watermarked Images. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 2591–2601, October 2020. doi:10.1109/TCSVT.2020.3030671.

[108] Hui Xu, Yuxin Su, Zirui Zhao, Yangfan Zhou, Michael R. Lyu, and Irwin King. DeepObfuscation: Securing the Structure of Convolutional Neural Networks via Knowledge Distillation, June 2018. arXiv: 1806.10313. URL: http://arxiv.org/abs/1806.10313.

[109] Kelvin Xu, Jimmy Lei, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural Image CaptionGeneration with Visual Attention. In *32nd International Conference on Machine Learning*, volume 37, pages 2048–2057, 2015.

[110] XiangRui Xu, YaQin Li, and Cao Yuan. A novel method for identifying the deep neural network model with the Serial Number, November 2019. arXiv: 1911.08053. URL: http://arxiv.org/abs/1911.08053.

[111] Xiangrui Xu, Yaqin Li, and Cao Yuan. "Identity Bracelets" for Deep Neural Networks. *IEEE Access*, 8:102065–102074, June 2020. doi:10.1109/ACCESS.2020.2998784.

[112] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated Machine Learning: Concept and Applications. *ACM Transactions on Intelligent Systems and Technology*, 10(2):1–19, February 2019. doi:10.1145/3298981.

[113] Ziqi Yang, Hung Dang, and Ee-Chien Chang. Effectiveness of Distillation Attack and Countermeasure on Neural Network Watermarking, June 2019. arXiv: 1906.06046. URL: http://arxiv.org/abs/1906.06046.

[114] Xue Ying. An Overview of Overfitting and its Solutions. *Journal of Physics: Conference Series*, 1168:022022, February 2019. doi:10.1088/1742-6596/1168/2/022022.

[115] Yingjiu Li, V. Swarup, and S. Jajodia. Fingerprinting relational databases: schemes and specialties. *IEEE Transactions on Dependable and Secure Computing*, 2(1):34–45, January 2005. doi:10.1109/TDSC.2005.12.

[116] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph. Stoecklin, Heqing Huang, and Ian Molloy. Protecting Intellectual Property of Deep Neural Networks with Watermarking. In *ACM Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 159–172, Incheon, Republic of Korea, June 2018. ACM. doi:10.1145/3196494.3196550.

[117] Jie Zhang, Dongdong Chen, Jing Liao, Han Fang, Weiming Zhang, Wenbo Zhou, Hao Cui, and Nenghai Yu. Model Watermarking for Image Processing Networks. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 12805–12812, New York, USA, February 2020. AAAI Press. doi:10.1609/aaai.v34i07.6976.

[118] Ying-Qian Zhang, Yi-Ran Jia, Xing-Yuan Wang, Qiong Niu, and Nian-Dong Chen. DeepTrigger: A Watermarking Scheme of Deep Learning Models based on Chaotic Automatic Data Annotation. *IEEE Access*, pages 213296 – 213305, November 2020. doi:10.1109/ACCESS.2020.3039323.

[119] Jingjing Zhao, Qingyue Hu, Gaoyang Liu, Xiaoqiang Ma, Fei Chen, and Mohammad Mehedi Hassan. AFA: Adversarial fingerprinting authentication for deep neural networks. *Computer Communications*, 150:488–497, December 2019. doi:10.1016/j.comcom.2019.12.016.

[120] Xiangyu Zhao, Hanzhou Wu, and Xinpeng Zhang. Watermarking Graph Neural Networks by Random Graphs, November 2020. arXiv: 2011.00512. URL: http://arxiv.org/abs/2011.00512.

[121] Qi Zhong, Leo Yu Zhang, Jun Zhang, Longxiang Gao, and Yong Xiang. Protecting IP of Deep Neural Networks with Watermarking: A New Label Helps. In Hady W. Lauw, Raymond Chi-Wing Wong, Alexandros Ntoulas, Ee-Peng Lim, See-Kiong Ng, and Sinno Jialin Pan, editors, *Advances in Knowledge Discovery and Data Mining*, pages 462–474. Springer International Publishing, May 2020. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-030-47436-2_35, doi:10.1007/978-3-030-47436-2_35.

[122] Xin Zhong, Pei-Chi Huang, Spyridon Mastorakis, and Frank Y. Shih. An Automated and Robust Image Watermarking Scheme Based on Deep Neural Networks. *IEEE*

Transactions on Multimedia, pages 1951–1961, 2020. doi:10.1109/TMM.2020.3006415.

[123] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random Erasing Data Augmentation. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 13001–13008, New York, USA, April 2020. AAAI Press. doi:10.1609/aaai.v34i07.7000.

[124] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, November 2017. arXiv: 1710.01878. URL: http://arxiv.org/abs/1710.01878.

[125] Renjie Zhu, Xinpeng Zhang, Mengte Shi, and Zhenjun Tang. Secure neural network watermarking protocol against forging attack. *EURASIP Journal on Image and Video Processing*, 2020(1), September 2020. doi:10.1186/s13640-020-00527-1.