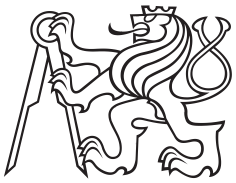
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computers

Machine learning privacy: analysis and implementation of model extraction attacks

Bc. Vít Karafiát

Supervisor: Ing. Maria Rigaki
Field of study: Open Informatics
Subfield: Cybernetic security
May 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Karafiát** Jméno: **Vít** Osobní číslo: **457000**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Kybernetická bezpečnost**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Soukromí strojového učení: analýza a implementace model-extrahujících útoků

Název diplomové práce anglicky:

Machine learning privacy: analysis and implementation of model extraction attacks

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. (2016). Stealing machine learning models via prediction apis. In25th USENIX Security Symposium (USENIXSecurity 16), pages 601–618, Austin, TX. USENIX Association
[2] Oh, S. J., Schiele, B., and Fritz, M. (2019). Towards reverse-engineering black-box neural networks. InExplainable AI: Interpreting, Explaining and Visualizing Deep Learning,pages 121–144. Springer.
[3] J. R. Correia-Silva, R. F. Berriel, C. Badue, A. F. de Souza and T. Oliveira-Santos, "Copycat CNN: Stealing Knowledge by Persuading Confession with Random Non-Labeled Data," 2018 International Joint Conference on Neural Networks (IJCNN), Rio de Janeiro, 2018, pp. 1-8, doi: 10.1109/IJCNN.2018.8489592.
[4] Orekondy, T., Schiele, B. and Fritz, M., 2019. Knockoff nets: Stealing functionality of black-box models. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4954-4963).
[5] Pal, Soham, Yash Gupta, Aditya Shukla, Aditya Kanade, Shirish Shevade, and Vinod Ganapathy. "ACTIVETHIEF: Model Extraction Using Active Learning and Unannotated Public Data." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 01, pp. 865-872. 2020.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Maria Rigaki, katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **07.09.2020** Termín odevzdání diplomové práce: **05.01.2021**

Platnost zadání diplomové práce: **19.02.2022**

Ing. Maria Rigaki podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

I want to thank my family and my wonderful supervisor for all the support and help during my work on this thesis.

Declaration

I hereby declare I have written this master thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, May 2021

Abstract

The rise in popularity and the large amount of improvements done to Machine Learning (ML) resulted in the emergence of a new type of attack called model extraction attack. Model extraction attacks are privacy attacks, which aim to extract information about a victim model or even steal its functionality. These types of attacks are being heavily researched, however, it is very hard to perform comparisons between the proposed papers. In this work, we present MET, which implements state-of-the-art model extraction attacks on arbitrary ML models and datasets. Using the tool, we performed a comprehensive comparison between the implemented attacks to see how they perform under different settings. Our results show that in the case of black-box scenarios, the attacks perform similarly. Based on the results, we propose and implement improvements for some of the attacks both in terms of speed and performance.

Keywords: machine learning, model extraction attacks, deep neural networks

Supervisor: Ing. Maria Rigaki
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czech Republic

Abstrakt

Rostoucí popularita a značná vylepšení ve strojovém učení měla za důsledek vytvoření nového typu útoku, zvaných model-extrahující. Model-extrahující útoky jsou útoky na soukromí, jejichž cílem je získání informací o daném modelu nebo dokonce ukradení jeho funckionality. Tyto typy útoků jsou četně zkoumány, je nicméně velice obtížné provést srovnání mezi konkrétními výzkumy. V této práci představujeme MET, který implementuje model-extrahující útoky a umožňuje jejich testování a experimentaci s libovlnými modely a datasety. Za použití tohoto nástroje jsme provedli komplexní srovnání implementovaných útoků s cílem vypozorovat, jak fungují v ruzných scénářích. Naše výsledky ukazují, že u scénářů, kde je pouze black-box přístup k modelu, útoky fungují podobně. Na základě těchto výsledků jsme navrhuli a implementaovali vylepšení některých z těchto útoků, co se týče jejich rychlosti i výkonu.

Klíčová slova: strojové učení, model-extrahující útoky, hluboké neuronové sítě

Překlad názvu: Soukromí ve strojovém učení: analýza a implementace modelu extrahujících útoků

Contents

Figures

Tables

Acronyms

AL Active Learning. 6–9, 17, 25, 71

API Application Programming Interface. 2, 16, 33, 35, 37

ART Adversial Robustness Toolbox. 46

ATLAS Active Transfer Learning for Adaptive Sampling. 71

CLI Command Line Interface. x, 40, 44, 48

CNN Convolutional Neural Network. ix, 10, 12, 13, 24, 27, 30, 48, 50

CPU Central Processing Unit. 34

DFAL DeepFool Active Learning. 25, 27, 62, 63, 70, 71, 76

DNN Deep Neural Network. ix, x, 8–12, 20, 22, 23, 25, 34, 36, 48, 50

DT Decision Tree. 23

ERM Emprirical Risk Minimization. 11

FGSM Fast Gradient Sign Method. 22

GAN Generative Adversial Network. 21, 30, 35, 51, 66, 76

GPU Graphical Processing Unit. 34, 70

LR Logistic Regression. 23

MET Model Extraction Tool. vi, ix, x, 2, 3, 33–37, 40, 41, 46–48, 50, 52, 53, 55, 57, 69–71, 74, 75, 77

ML Machine Learning. vi, 1–3, 5, 6, 8, 10, 15, 20, 21, 23, 34, 35, 46, 58

MLaaS Machine Learning as a Service. 1, 17, 18

MSE Mean Squared Error. 11

NN Neural Network. 20, 46

NPD Non-Problem Domain. 2, 65, 74, 75

NPDN Non-Problem Domain Natural. 24, 25, 27, 29, 57, 60, 72

NPDS Non-Problem Domain Synthetic. 57

OS Operating System. 34

PD Problem Domain. 2, 23–25, 27, 30, 50, 65, 74, 75

PRADA Protecting against DNN Model Stealing Attacks. 49

RAM Random Access Memory. 70

SGD Stochastic Gradient Descent. 12

SSL Semi-Supervised Learning. 8, 9, 74, 76

SVM Support Vector Machine. 23

TL Tranfer Learning. 8, 71

UML Unified Modeling Language. x, 42–45

VAE Variantional Auto Encoder. 35

VRAM Video Random Access Memory. 70

Chapter 1

Introduction

In recent years, a new branch of privacy attacks against Machine Learning (ML) models emerged called model extraction attacks or model stealing. These attacks focus on extracting information and copying the victim models while having limited information about the model architecture, parameters, and training datasets. The creation of the ML models is often very costly due to both the time and money required to create and train them [4, 5]. It is thus crucial that the information about a model and a model itself stays secret. This is even more important in recent years with the rise of Machine Learning as a Service (MLaaS) and since more and more companies use ML as part of their business model [6].

Since the original paper by Tramèr in 2016 [7] showed that model extraction attacks are indeed possible against public models, many more possible attacks were introduced. Model extraction attacks are still in their exploratory stage; the papers are using different datasets and model types for testing of the attacks [2], which makes the comparison between them hard. Not all papers have their source codes released. Those who have use different frameworks and libraries. Some also use specific dataset splits for the attacks, which are not described in the code or in the publication, making the reproducibility of attacks much harder. Furthermore, because the code was primarily created for that paper's experiments, it is not easily used with different models and datasets. Currently, there exists no comprehensive comparison of the attacks found in the literature and there is no easy way to test these attacks on arbitrary ML models and arbitrary datasets.

Available tools such as [8, 9] are offering security and privacy attack implementations for testing, but have only basic and limited support for model extraction attacks. Currently, there is no easy way to compare and test individual model extrac-

tion attacks. This complicates the research of model extraction attacks, since any researchers that want to test new model extraction attacks, propose improvements to them and compare their new attacks with others must implement the existing attacks from scratch. In this thesis, we aim to rectify these problems by reviewing the state-of-the-art model extraction attacks and developing a tool that would allow to easily test different ML models against these privacy attacks.

The Model Extraction Tool (MET) that is one of the outputs of this thesis has at the time of writing, support for the largest number of model extraction attacks. It offers an intuitive Application Programming Interface (API) and support for arbitrary ML models and datasets. The tool also contains basic building blocks to implement new model extractions attacks easily. Additionally, several popular ML models and datasets are built in. As part of the tool, we also provide scripts reproducing some of the experiments from the original attack papers. We hope that by providing all the necessary tools for implementing and testing new model extraction attacks, the speed and reproducibility of research can improve. At the same time, the tool will allow end users to test their models against state-of-the-art model extraction attacks and potentially improve their defences.

Using MET, we performed a comprehensive comparison of the current state-of-the-art model extraction attacks, which is currently not available in the research literature. We perform our comparison in a challenging scenario, where an adversary has only black-box access to the victim model, limited budget, and access only to Non-Problem Domain (NPD) data for all of the attacks. Here, NPD data is publicly available data of the same type of content as the victim model's input, e.g., image data for image models and text data for text models. Unlike Problem Domain (PD) data, it is not drawn from the same data distribution as the one used for the training of the victim model e.g., medical image data to extract models trained on medical images. [10]. All of our experiments were performed in the image classification domain, where currently the majority of research in model extraction attacks is performed [2]. However, MET is not limited to image classification and can work with arbitrary datasets. In chapter 4 we show example on how to use the tool with cybersecurity dataset.

The results of our comparison show that the difference in the performance of the individual attacks is minimal when the adversary has only black-box access to the victim model and NPD data, while the runtime performance shows a much more significant variance between individual attacks. We also tested the basic defense against model extraction attacks by limiting the granularity of the victim model's output. The results show that less granular output has a negative impact on the performance of all attacks, however for some of the attacks the impact is only marginal. From all implemented attacks, the largest impact was seen on ActiveThief attack.

Finally, we improved both the speed of the attacks and the performance of some of the attacks. Our improved implementation of the k-center method in the ActiveThief attack, called k-center fast, is over 1000× faster that the original implementation. We also significantly improved the query efficiency of the BlackBox ripper attack, maintaining the same performance as the original version of the attack, while using over 6× smaller budget.

The contributions of this thesis are following:

1. A tool that currently implements the largest number of model extraction attacks compared to other available tools. The tool supports the ability to test model extraction attacks against different models and datasets. We call this tool Model Extraction Tool (MET).
2. The most comphrehesive comparative study of the different attacks to date in terms of efficiency, applicability, and scalability.
3. Improvements for some of the attacks, based on the findings from our experiments.

Thesis's structure. This thesis is organized as follows. Chapter 2 gives a theoretical overview of Machine Learning (ML). Chapter 3 introduces privacy attacks against machine learning, with the main focus on model extraction attacks. Chapter 4 describes the implemented Model Extraction Tool (MET), created to test model extraction attacks. Chapter 5 presents the comparison between the attacks' implementation in MET and the original papers. In Chapter 6 describes the experiments and the results of the comparison of the implemented attacks in the same setting. Lastly, chapter 7 discusses the improvements on some of the attacks and the possible future direction of research.

Chapter 2

Background

This chapter explains ML related topics which are used in different model extraction attacks and mentioned throughout the thesis. We provide only a brief description of each relevant topic and refer the reader to the literature for more information.

2.1 Machine Learning

ML is a subset of artificial intelligence, which creates models that can perform tasks that would be too complex to program or that they are adaptive in their nature [11]. The goal is to create learning algorithms that work automatically without human intervention. Rather than programming the solution for the task, the algorithm comes up with its own program based on the provided input data, this approach is called *data-driven approach*, since it relies on the accumulation of a training dataset first [12].

In general, it is possible to say that there are mainly three categories of ML, called *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*. The first subsection briefly describes these three categories, and the following subsections are dedicated to different types of learning. We follow the descriptions from [13, 14] and refer readers to them for more details. Together with these three main categories, we also describe the categories beyond supervised learning relevant to model extraction attacks. These are Active Learning, Semi-supervised Learning, and Transfer Learning.

2.1.1 Main Types of Learning in ML

Supervised Learning. In supervised learning, for each training data point $\mathbf{x}$ the corresponding output $\mathbf{y}$ is known [13]. A variable $\mathbf{x}$ is a vector of vectors, where each vector represents attributes or features. $\mathbf{y}$ is a vector corresponding to a label for each data point, which may have different dimension depending on the learning task. The goal is to learn a model f with parameters θ that will map inputs to outputs $\mathbf{y} = f(\mathbf{x}; \theta)$ using the training set $\mathcal{D}_{train} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{m}$ with m input-output pairs [13]. Examples of supervised learning are regression and classification. In regression, the aim is to learn a model that predicts a real value output for each data point. In classification tasks, the goal is to predict the class label for each data point.

Unsupervised Learning. In case of Unsupervised Learning, the training set contains only inputs $\mathcal{D}_{train} = \{\mathbf{x}_i\}_{i=1}^{m}$ with m samples. Its goal is to find hidden patterns and groupings in the data. In this case, the problem is typically to partition the training set into some appropriate subsets. The main application of Unsupervised Learning is in taxonomic problems in which we try to classify data into meaningful categories [13].

Reinforcement Learning. In reinforcement learning, an agent learns by interacting with the environment in which it exists. It operates by taking, at each time step t, the action a_t from the action space $\mathcal{A}$. After each action is taken, the environment returns a state s_{t+1} and a rewards r_t as feedback to measure the success or failure of the agent's action. The goal of the agent is to maximize the cumulative rewards signal [15].

2.1.2 Active Learning

Supervised learning models usually improve their accuracy with more labeled data available during the training phase. However, often in supervised ML tasks, the size of the labeled data is much smaller compared to the size of the unlabeled data. It is usually costly to annotate the unlabeled data by an oracle (human or some other source). This is where Active Learning (AL) comes in. It focuses on deciding which data points need to get annotated to minimize the cost. It is a special case of ML, where it is possible to query the oracle during training for labels.

As stated in [1] there are three main scenarios for AL in which the active learner

queries data samples to the oracle to obtain the labels. The Figure 2.1 depicts each of these scenarios:

The first scenario is *Query synthesis*. In this scenario, it is assumed that the learner knows the distribution of the input space and can synthesize new data samples that are then queried to the oracle [1].

The second scenario is *Stream-based selective sampling*, the key assumption is that obtaining an unlabeled data sample is free or inexpensive, which means it is possible to take sample data from the real distribution. The learner then decides whether to query the data sample or discard it. The decision is usually based on the sample's informativeness, which is measured by the AL strategy [1].

The third scenario is *Pool-Based Sampling*. For many tasks, it is possible to gather a large collection of unlabeled data at once. This is the motivation of *pool-based sampling* where a large pool of unlabeled data U is available, from which an AL strategy selects samples for querying. Unlike stream-based AL, which makes each query decision on individual data samples, the pool-based AL performs the decision using all of the unlabeled data points and, after ranking them, selects the best query. Most of the work in AL is focused on pool-based scenario [1].

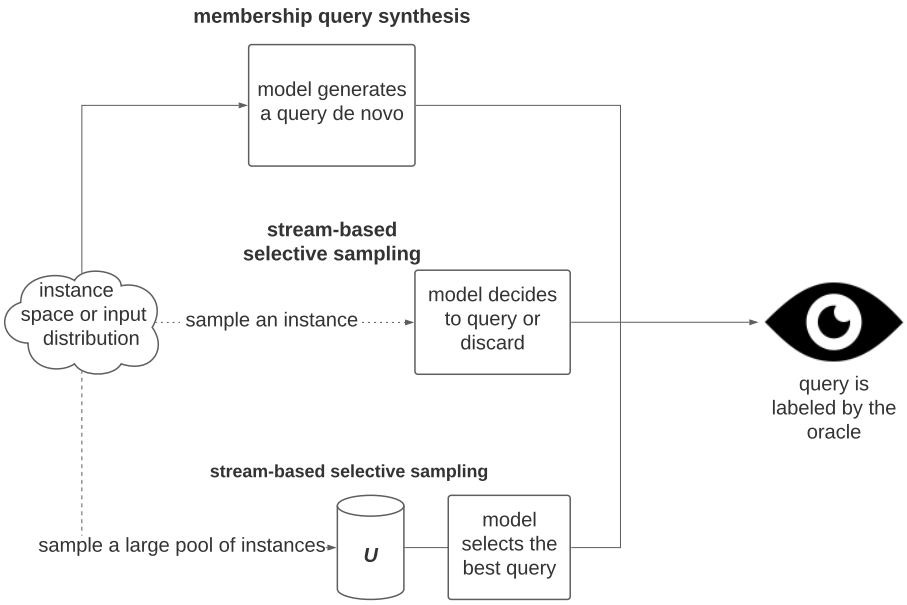


Figure 2.1: Visualization of active learning scenarios. The image is based on the original research [1].

2.1.3 Transfer Learning

One problem with the training of more complex Deep Neural Networks (DNNs) is that they require large amounts of training data to perform well and prevent over-fitting. However, it is frequently costly to acquire enough labeled training data for the target task. Tranfer Learning (TL) tries to solve the problem by transferring knowledge from one task to a related target task, where the latter has fewer training data [16]. From the definition of TL in [17]:

> "Given a source domain D_S and learning task T_S, and a target domain D_T and learning task T_T, TL aims to help improve the learning of the target predictive function $f(\cdot)$ in D_T using the knowledge in D_S and T_S, where $D_S \neq D_T$, or $T_S \neq T_T$."

TL assumes that the features learned by the model that explain the variation in task T_S can help with explanation of variation in target task T_T. Therefore, for example, if in the source task T_S we try to classify dogs and cats and in the target task T_T we want to classify cows and sheep, we can leverage the knowledge that the model learned in the T_S, where we have enough training data and improve the generalization from the few examples in the T_T. Instead of randomly initializing the model parameters, we take the parameters learned on T_S and fine-tune the model on the task T_T during training. In the case of computer vision tasks, this works since many visual categories share low-level notions of edges and visual shapes, the effects of geometric changes, changes in lighting, and so on [18].

TL found, especially in computer vision, substantial success thanks to the availability of many pre-trained models on various tasks such as ResNet [19], or VGGNet [20], where many of them are available in popular ML frameworks and online. This availability of pretrained models allowed a rise in classification accuracy for visual tasks with increasing computational complexity [21].

2.1.4 Semi-Supervised Learning

Semi-Supervised Learning (SSL) falls between supervised and unsupervised learning. The goal of SSL is to leverage the unlabeled data in such a way to create a model, which performs better than a model trained only on labeled samples. It tries to solve the same problem as Active Learning (AL), meaning how to effectively use the unlabeled data since the discrepancy in labeled and unlabeled data is often

significant in real-world problems. However, unlike in AL setting, it is impossible to query the oracle for labels interactively. The SSL however, has some prerequisites that the sample distribution must have for the SSL to be usable [22, 23]. These prerequisites are the following assumptions about the input data:

- *Smoothness*: Two data points $\mathbf{x}$ and $\mathbf{x}'$ that are close in the input space should also have the corresponding labels $\mathbf{y}$ and $\mathbf{y}'$ close.
- *Low-density*: The classifier's decision boundary should preferably pass through low-density regions in the input space: The low-density region is an area where few data points are observed.
- *Manifold assumption*: If two data point $\mathbf{x}$ and $\mathbf{x}'$ are located in a local neighborhood in the low-dimensional substructure, they should have similar class labels. These substructures are called a manifold, which is a topological space that is locally Euclidean.

Since in our work we focus primarily on DNN we will discuss the SSL methods used for DNN. Following the taxonomy presented in [22], the SSL methods can be separated into five categories, i.e., generative methods, consistency regularization methods, graph-based methods, pseudo-labeling methods, and hybrid methods. In our work, we primarily researched the pseudo-labeling methods and thus only provide a brief description for them. The reader can refer to [22, 23] for more details about the other categories.

Pseudo-Labeling Methods. Pseudo-labeling methods can be separated into two main types: disagreement-based models and self-training models. The main idea in disagreement-based SSL is to train multiple models for the target task and use the disagreement between the models during the training process to find the best candidates from the unlabeled samples. The methods usually create two or three models, which then are used to label the unlabeled samples for each other.

In the self-training scenario, there is only one model. The model is first trained on the labeled samples and afterward uses its confident prediction to create pseudo labels for the unlabeled data. This means that the model can create more training data by using its existing labeled data to create labels for unlabeled data. Thanks to their simplicity and generality, self-training SSL methods are successfully applied in different tasks with good performance [22].

2.2 Deep Neural Network

In this section, we describe the general DNNs architecture and their training and testing phase, since the *model extraction attacks* we implemented focused primarily on DNNss. We also introduce a subcategory of DNNs called Convolutional Neural Networks (CNNs), which in recent years achieved tremendous performance in image classification tasks [19, 20] and are used in most of the attack papers and in our experiments. Our description is based on [12, 18, 24]. The reader can refer to them for more details.

The Deep Neural Network (DNN) is composed of L different functions f, connected in a chain to form a hierarchical structure. We refer to these functions f as layers, where each layer in the network is represented by neurons (or sometimes called units). Typically the first layer is called the input layer, the last layer is called the output layer, and layers in between are called hidden layers. What makes DNN deeps is that it has multiple hidden layers, i.e., two or more. Neurons are basic computing units that apply an *activation function* (e.g., sigmoid, RELU, leaky RELU) to the previous layer output weighted by parameters θ. Each layer i for $i \in 0 \dots L$ is representing a function f_i and is parametrized with a vector of vectors $\theta^{(i)}$, where each vector represents parameters (e.g., weights, bias) for each neuron in the layer. These parameters impact the activation of each neuron and are the "knowledge" of the DNN model f. Following this definition, the DNN is defined and computed as follows:

$$\mathcal{M}(\theta_{\mathcal{M}}; \mathbf{x}) = f_L(\theta^{(L)}; f_{L-1}(\theta^{(L-1)}; \dots f_2(\theta^{(2)}; f_1(\theta^{(1)}; \mathbf{x})))) \tag{2.1}$$

where $\theta_{\mathcal{M}} = \{\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(L)}\}$ are parameters of the model $\mathcal{M}$ and vector $\mathbf{x}$ represents input data. The illustration of basic neuron and general DNN can be seen in Figures 2.2 and 2.3. To make our terminology consistent with other works we will use f to symbolize a ML model from now on in the thesis.

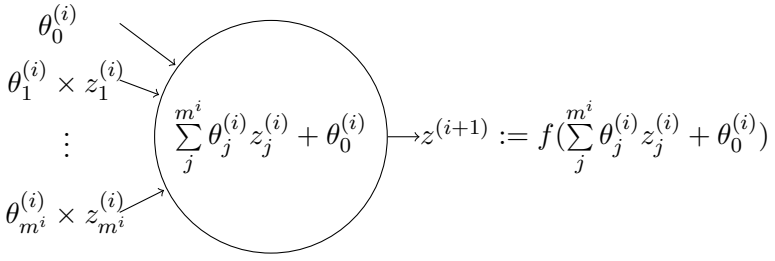


Figure 2.2: Illustration of a single neuron in a DNN. Parameters $\theta_1^{(i)} \dots \theta_n^{(i)}$ represents weights, $\theta_0^{(i)}$ represents bias term and f represents activation function.

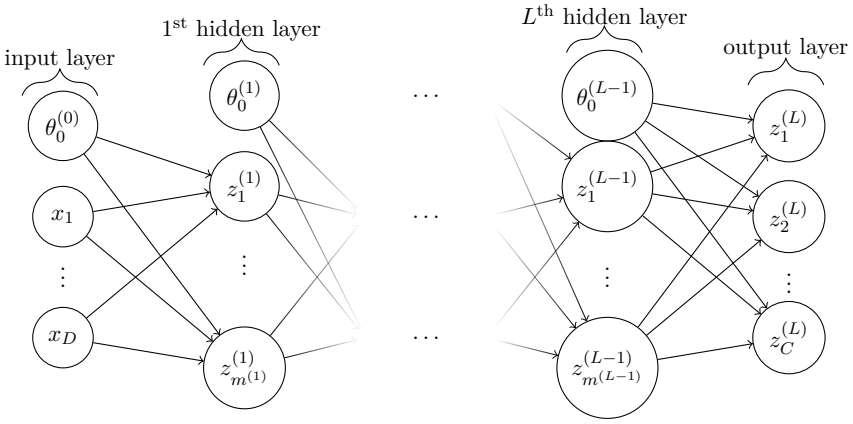


Figure 2.3: General DNN architecture with L layers, where each layer is fully-connected. D represents the input size, C represents the number of output classes and m^i for $i \in 0 \dots L$ represents the number of neurons in the ith layer.

In this work, we primarily focus on image classification tasks, where the objective is to assign a label to an input corresponding to a single class c from a predefined set of classes C and thus we describe the training and testing of a supervised DNN model.

2.2.1 Training Phase

As stated in [2, 18] *supervised learning* usually follows the Emprirical Risk Minimization (ERM), the idea is that we do not know the true data distribution that the model will work on, instead we use the training set to estimate the performance of the model since it is a subset drawn from this distribution. So for the model $\mathcal{M}$ ERM algorithm tries to find the values of its parameters θ that minimize the *objective function* calculated as an average over the training set:

$$\mathcal{J}(\mathcal{D}_{train}; \theta) = \frac{1}{m} \sum_{i=1}^{m} l(f(\mathbf{x}_i; \theta), \mathbf{y}_i) \tag{2.2}$$

where $l(\cdot)$ is a loss function (e.g., cross entropy, MSE) that measures the error between the prediction of the model f and the correct output $\mathbf{y}$. The m is the number of samples in training set $\mathcal{D}_{train}$. Often times regularization $\lambda \times \mathcal{R}(\theta)$ term is also added to the loss function to prevent over-fitting of the model and reduce its complexity, the λ is used to balance between these two.

The minimization of the objective function is usually done with iterative opti-

mization algorithms such as gradient descent that follow the objective function's gradients' path. DNN's usually require a large dataset to learn. To improve speed instead of taking a single gradient step after each sample, we can make batches of multiple samples on which we calculate the gradient. This type of gradient descent is called mini-batch Gradient Descent but is often called Stochastic Gradient Descent (SGD) in literature even though SGD refers to using a single example at a time to evaluate the gradient [12]. Hyper-parameters such as learning rate, momentum, etc, are also an important part of optimization algorithms, which can greatly influence the performance of the final model.

The parameters $\theta_{\mathcal{M}}$ themselves are adjusted in DNN with a back-propagation algorithm after each forward pass through the network that propagates error gradients with respect to the parameters from the output layer to the input layer. The description of the back-propagation algorithm is out of the scope of this thesis and we refer to [12, 18] for more details.

2.2.2 Test Phase

Once the training phase is completed, the parameters of the model, θ, are fixed. A DNN model can then be deployed and is used to make predictions on the test data. Since we are primarily considering classifiers, the DNN usually has a softmax function applied to its output to create a probability vector, which encodes its confidence of input $\mathbf{x}$ belongning in each of the classes. Ideally, the model is able to generalize well and make accurate predictions for inputs outside of the training set.

2.2.3 Convolutional Neural Network

Convolutional Neural Network (CNN) models are a category of DNN models they are similar to ordinary models in that each layer of the network is fully connected to the previous layer. The problem with fully connected networks is that they do not scale well for full images, which generally have width, height, and depth. For example, $224 \times 224 \times 3$ would lead to neurons that have $224 \times 224 \times 3 = 150,528$ weights, and the model would require multiple such neurons, which would increase the number of parameters dramatically. CNN work with the assumption that the input is an image, which allows them to encode properties in their architecture, which makes the forward pass through them more efficient and lowers the number of necessary parameters [12]. At the heart of a basic CNN are three main layer types: the *convolution layer*, the *pooling layer*, and the *fully-connected layer*. A schematic representation of a CNN can be seen in Figure 2.4. We will focus only on

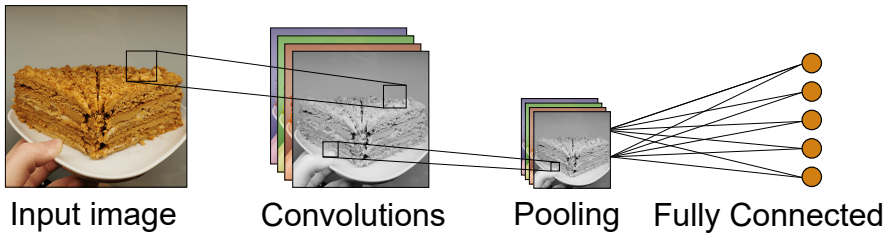


Figure 2.4: Example of CNN showing both convolution layer and pooling layer.

the description of the convolution and pooling layers since they are specific to the CNN.

In the *convolution layer*, the model is learning a set of filters. Each filter is taking only a small section of the image but across the input volume's full depth. As an example, we can take the typical first convolution layer of CNN, which can have dimensions of 7x7x3 (i.e., 7 pixels width and height and depth of 3 since pictures usually have three color channels). The filter slides across an image during a forward pass. At each position, a dot product is calculated between the filter and the input from the image. The result is a 2-D activation (feature) map for each filter, which shows the filter's response at each position of the input. Through the activation maps of each filter, the model can learn interesting visual features of the input, such as edges and colors in earlier layers and more complicated patterns in later layers. These maps are then stacked and are the output of the convolution layer.

Pooling layers are used in CNNs to reduce the height and width of the feature maps in the network but keep the depth, which results in the decrease of parameters and makes the computation faster. This also protects the network from overfitting. Similarly to the convolution layer, the pooling layer uses a filter, which slides over the input and applies an operation (e.g., MAX, AVG) at each position of the input.

Chapter 3

Model Extraction Attacks

Model extraction is a type of privacy attack, where an adversary that aims to obtain information about a victim model or to extract the model itself. The goal of the attacker is often the creation of a substitute model that approximates or even matches the victim model. To achieve this, the adversary might try to obtain the values of the victim models' parameters [7, 3], hyper-parameters [25] and architecture [26]. This chapter is dedicated to the introduction of the problematic of model extraction attacks and the current state-of-the-art research in the area. We shall also present the attacks selected to be implemented in the model extraction tool and the reasoning behind the selection.

3.1 Threat Model

Before describing the individual model extraction attacks, we must introduce the threat model in which these attacks work. This means describing the individual actors, their capabilities, and their goals. Our threat model is based on the threat model introduced in [2] for privacy attacks.

3.1.1 Actors

Regarding the actors, we have the *data owner*, the *model owner*, the *end-user* and the *adversary*, where each has a different relationship to the ML model.

1. The **data owner**'s primary concern is that both the information about their dataset and the samples themselves are not leaked.

2. The **model owner** depending on the setting, might share some information about their models, such as parameters or the architecture. Their main concern is that the information that is kept private stays private. The data owner can also be the model owner and vice versa.

3. The **end-user** is interested in using the model provided by the model owner to get predictions on their data. Usually, they can access the model either via an API or through an application to which they use to query their data.

4. The **adversary** depending on the settings, might have the same access to the model as a end-user, or they might have access to more information about the model.

The object of interest for the adversary changes depending on the the privacy attacks. For model inference, reconstructing attacks and property inference attacks, the focus is acquiring information about the training dataset D_v, while in the case of a model extraction attack, the focus is the model. In this thesis, we are focused on model extraction attacks, we consider an adversary, whose primary interest is the information related to the model provided by the model owner. This machine learning model for model extraction attacks is often called the *victim model*, which we symbolize as f_v for the remaining of this document. In figure 3.1 we depict the actors, assets, and the relationships between them.

3.1.2 Adversary's Capabilities

The vital thing to consider is the adversary's capabilities, which are usually concerning the adversary's knowledge about the victim model. The case where the adversary knows everything about the model (e.g., architecture, training data, parameters, hyper-parameters) is called a *white-box* attack. The opposite case where the adversary does not know anything about the model is called a *black-box* attack. Between these two extremes, we can find adversaries with varying knowledge about the victim model. These attacks are called *partial white-box* as mentioned in [2].

Another thing to consider is the knowledge the adversary has about the problem domain. This means whether or not the adversary knows for which task the victim model f_v was trained. It is reasonable to assume that in most cases the adversary will know as much about the problem domain as the model owner.

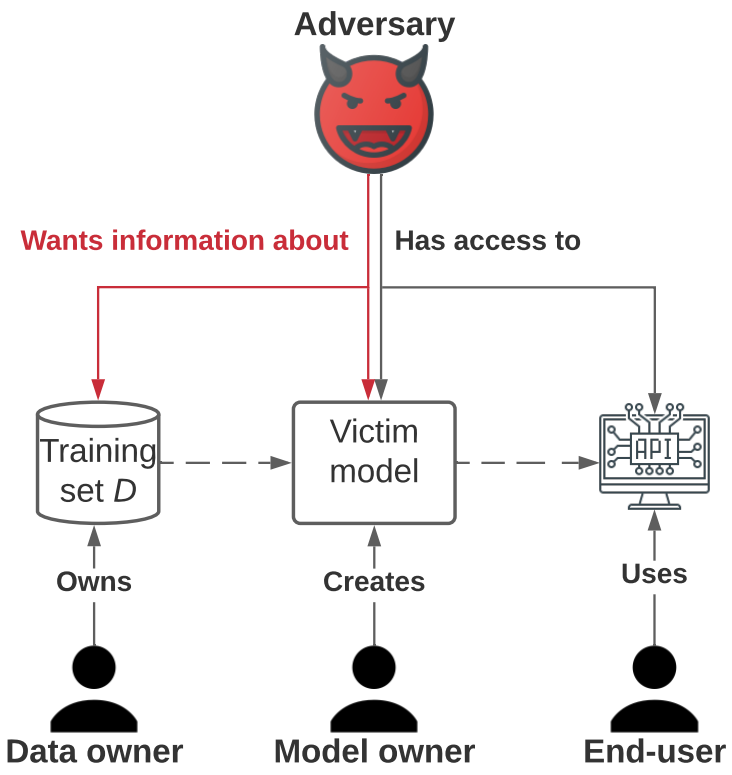


Figure 3.1: Threat model of privacy attacks depicting the individual actors and their capabilities. The dashed lines represent the flow of information. This image is based on the original from [2].

In black-box attacks, the only information the adversary can get is through the victim model's query responses. However, these responses can differ in the degree of informativeness, either in the best case, returning the probabilities for each class or returning responses that might be one-hot encoded, i.e. returning only information about the most probable class. Chandrasekaran et al. [4] described a parallel that can be drawn between Active Learning Active Learning (AL) and model extraction attacks in the context of MLaaS, where the oracle is not human as is commonly the case but the victim model f_v.

3.1.3 Adversary's Motivation

There are two main reasons an adversary would be interested in extracting information about the victim model f_v. The first type of attacker we can consider is an attacker motivated by monetary gain. The training of the victim model is usually

expensive both in terms of time and money. The model owner is thus interested in getting a return on their investment by providing the model behind a paywall, which is usually the case for MLaaS that charges end-users for queries. By creating a substitute model f_s, the adversary can circumvent the paywall to get annotations for their data or monetize it the same way as the original model.

The second motivation is reconnaissance, where the victim model's information can help the adversary perform additional attacks against the victim model. This mainly refers to security attacks, specifically evasion attacks. By obtaining a substitute model, which behaves similarly to the victim model, the adversary suddenly works in a white-box scenario when crafting the adversarial inputs to fool f_v. This can also be leveraged in membership inference attacks, i.e., privacy attacks.

3.1.4 Adversary's Goals

We divide the adversary goals according to the precision of the extraction of the substitute model as described in [3]. The precision here represents the amount of agreement between the victim model f_v and the substitute model f_s. Starting from the most precise extraction to the least precise, the adversary can have the following goals *functionally equivalent extraction*, *fidelity extraction* and *task accuracy extraction*. We depict each of these goals graphically in figure 3.2.

Functionally Equivalent Extraction

The adversary's goal is to construct a substitute model such that $\forall \mathbf{x} \in \mathcal{X}, f_s(\mathbf{x}) = f_v(\mathbf{x})$. Meaning that for any possible input $\mathbf{x}$ the substitute model f_s has the same predictions as the victim model f_v. As mentioned in [3] this is the hardest possible task in model extraction when working with only input-output pairs. The final substitute model f_s of funtionally equivalent extraction is depicted in Figure 3.2 as a blue line, which also represents f_v.

Fidelity Extraction

Instead of focusing on agreement over the whole input space $\mathcal{X}$ fidelity extraction focuses on maximizing similarity of predictions between the substitute model f_s and the victim model f_v on some target distribution $\mathcal{D}_f$ over $\mathcal{X}$. The similarity is

measured via some similarity function S. This means that the adversary is interested in that the substitute model f_s has the same correct and incorrect predictions as the victim model on the D_f. This goal is much easier for the adversary to achieve than the functionally equivalent extraction, where the goal is to achieve perfect similarity on every target distribution and every similarity function. The substitute model f_s created by the attacker, whose goal is fidelity extraction, is depicted as the green line in Figure 3.2.

Task Accuracy Extraction

Here, the adversary tries to achieve the best possible performance on some task distribution D_t from $\mathcal{X} \times Y$. This makes the problem easier as f_s no longer has to copy the wrong predictions of f_v. Instead, the adversary is only interested in that f_s achieves similar or better performance than f_v on D_t. The main difference between the fidelity extraction and task accuracy extraction is that in the former, the adversary controls D_f and does not necessarily care about generalization in the underlying task, while in the latter the D_t is the true task distribution. The orange line in Figure 3.2 represents the substitute model f_s created by an attacker who focuses on task accuracy extraction.

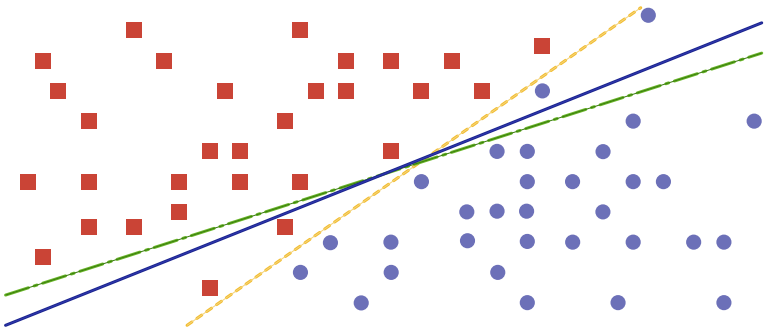


Figure 3.2: Visualization of the adversary goals. The blue line represents the decision boundary, represents both the victim model and adversary, whose focus is functionally equivalent extraction. The green line shows the final decision boundary created by an adversary that focuses on fidelity extraction. Finally, the decision boundary created by the adversary focusing on task accuracy extraction is represented by the orange line. The figure is based on the original from [3].

3.2 Attack Requirements

Following the above introduction of the threat model, we would like to dedicate this section to describe what attributes we were looking for when researching the current

state-of-the-art model extraction attacks. The four main points on which we focused during the selection process are:

- The attack threat model should assume that the adversary has only limited knowledge about f_v. This means we are primarily focused on attacks which are close to or being black-box.
- The attack focuses primarily on stealing NN, specifically DNN but should be able to work with an arbitrary model architecture for both the substitute and victim model.
- If possible, the attack should not be limited to the image classification task, where most of the research in model extraction attacks is done.
- The primary adversary's goals of interest were fidelity and task accuracy extraction, since both of them are much easier to achieve than functionally equivalent extraction.

Additionally, we tried to find attacks which employ different ML approaches to perform model extraction. Another aspect that influenced the attack selection was the availability of information about the test setups for individual attacks as well as if the authors provided code for their experiments. The last two reasons were not the most important factors, but they were also considered, since they allow us to easily compare the correctness of our implementation with the original.

3.3 Attack Metrics

Since we are interested in fidelity extraction and task accuracy extraction in the attacks and perform the experiments in the image classification domain, we used the following two metrics to compare the performance of the different attacks. The first one is the accuracy of the final substitute model f_s at the end of the attack on the test set D_t, which tells us how good was the attack in task accuracy extraction. The accuracy of classification model f is calculated with the following formula:

$$Accuracy(f) = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{3.1}$$

where the correct prediction is the prediction, where the predicted label corresponds to the true label of the sample. The second metric we looked at is the agreement

between the victim model f_v and the substitute model f_s on the test set D_t, which is calculated as:

$$Agreement(f_s, f_v) = \frac{1}{\|D_t\|} \sum_{x \in D_t} \mathbf{I}(f_s(x) = f_v(x)) \tag{3.2}$$

where $\mathbf{I}$ is the indicator function which is 1 if there is agreement between the models and 0 otherwise. This metric tells us how good was the attack in fidelity extraction and was used in both [7] and [10] to measure the performance of the attacks.

3.4 Researched and Implemented Attacks

In table 3.1 a non-exhaustive list of papers on model extraction attacks, which were researched and considered to be implemented in the testing tool. From the researched attacks according to our criteria, which we mentioned in 3.2 we chose to implement in our tool CopyCat [27], BlackBox [24], ActiveThief [10], KnockOff-Nets [5] and BlackBox ripper [6] attacks. Each of these attacks is based on different techniques: active learning (i.e., ActiveThief), reinforcement learning (i.e., KnockOff-Nets), learning using adversarial samples (i.e., BlackBox), and learning with the help of GAN (i.e., BlackBox ripper). We believe that this selection can show the diversity of possible model extraction attacks nicely. We also want to clarify the selection of the BlackBox attack since it is limited to image classification, which does not satisfy one of our criteria. We chose to implement the attack since it is well documented and relatively easy to implement. We were also interested in the performance of the attack.

For each implemented attack, we provide a brief explanation of the attack, the assumptions about the adversary's capabilities, the ML algorithms used, and the information about the used datasets. Since we are primarily focusing on image classification tasks in this thesis, we will mainly mention only those datasets related to image classification with their name. We leave it to the reader to check the corresponding papers for more details

Paper	Limited knowledge about f_v	Works with arbitrary model architecture	Not limited to image classification	Fidelity extraction	Task accuracy extraction
Tramèr et al. [7]			✓	✓	✓
Binghui et al. [25]			✓	✓	
Jagielski et al. [3]			✓	✓	✓
Honggang et al. [28]	✓	✓			✓
Oh et al. [26]	✓	✓			
Papernot et al. [24]	✓	✓		✓	
Silva et al. [27]	✓	✓	✓		✓
Pal et al. [10]	✓	✓	✓	✓	
Orekondy et al. [5]	✓	✓	✓		✓
Barbalau et al. [6]	✓	✓	✓		✓

Table 3.1: Researched model extraction attacks. The task fidelity and task accuracy extraction column represent what type of extraction was the focus in the corresponding paper.

3.4.1 BlackBox Attack

The BlackBox attack from Papernot et al. [24], focused on showing an attack against a DNNs that would create a substitute model using a synthetic dataset, which could be then used to craft adversarial samples that the victim model would missclassify. The proposed attack uses the so called *Jacobian-based dataset augmentation* to generate new samples of the synthetic dataset. The dataset augmentation works by adding a small perturbation to the original sample $\mathbf{x}$ in the direction of the gradient with respect to the label given to the sample by the victim model $f_v(\mathbf{x})$. The perturbation is defined as follows:

$$\acute{\mathbf{x}} = \mathbf{x} + \lambda \times sgn(J_F(\mathbf{x})[f_v(\mathbf{x})]) \tag{3.3}$$

where λ represents a step size and it is a hyperparameter that is specified before starting the attack. J_F represents the Jacobian matrix and $f_v(\mathbf{x})$ represents the prediction of the victim model on the input sample $\mathbf{x}$. As mentioned in [29] the Jacobian augmentation is the same as the calculation of Fast Gradient Sign Method (FGSM) adversarial attack [30].

The attack starts with an initial adversary's dataset D_a consisting of seed samples, which are classified by the victim model f_v and create an initial training set $\tilde{D}_1$ used

in the trainining of the substitute model f_s. Following the training, new synthetic samples $\hat{\mathbf{x}}$ are generated using the Jacobian-based augmentation. The augmented samples $D_{aug,i}$ are used to queriy the victim and $(\hat{\mathbf{x}}, f_v(\hat{\mathbf{x}}))$ are added to $\tilde{D}_{i+1}$. The substitute model is then retrained from scratch with the updated dataset $\tilde{D}_{i+1}$ and the whole process repeats for ρ iterations, where the authors call each of these iterations *substitute training epoch*. The whole attack process is shown in Figure 3.3.

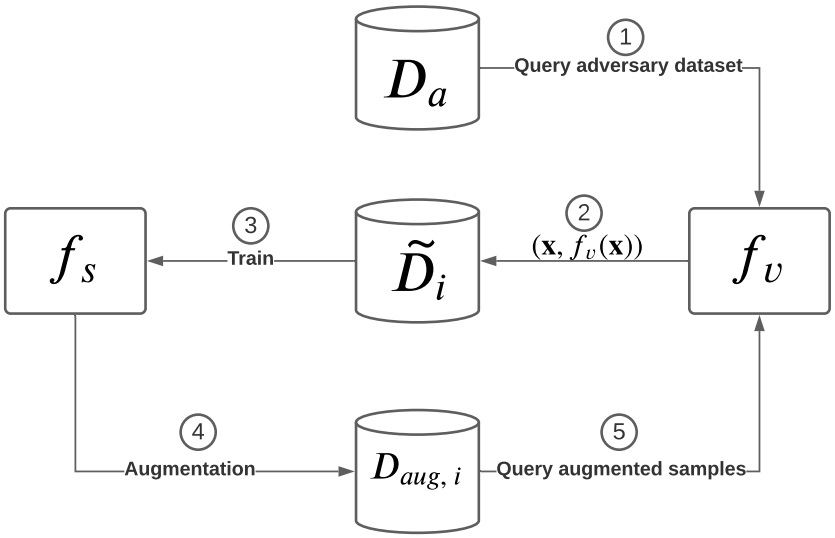


Figure 3.3: Flowchart depicting the steps of BlackBox attack.

The main focus of the paper was to show that adversarial examples created on the substitute model can be used to fool the victim model, thus the attack is not focused on the accuracy of the final substitute model but that it approximates the decision boundaries of the victim model well. The attack expects that the adversary has limited knowledge of the victim model and has only black-box access to the victim model. The adversary is also assumed to have access to the data samples from the Problem Domain (PD) (e.g. images of digits in case the victim is digit classifier), which are used as the seed samples for the attack. The attack was tested against a DNNs trained on MNIST [31] and GTSRB [32] datasets and the results showed both good transferability of the adversarial images created with the substitute model and also accuracy of the substitute model. The authors additionally also showed that their attack was general enough and could also be used against other ML algorithms such as LR, SVM and DT.

The vanilla version of the attack has, however, a problem with query efficiency. The attack budget corresponds to $n \times 2^{\rho}$, where n represents the number of seed samples. The authors proposed *reservoir sampling* as a solution to this problem. Instead of querying all newly generated synthetic samples, k random samples are instead chosen at each iteration starting from iteration $\rho > \sigma$, where the first σ iterations run normally. This change makes the new attack budget $n \times 2^{\sigma} + k \times (\rho - \sigma)$.

3.4.2 CopyCat Attack

Silva et al. [27] proposed the so-called *CopyCat attack*. They showed that it is possible to steal the victim CNN model by querying it with random unlabeled natural samples. The idea of the attack is based on the hypothesis that the CNNs does not require training samples from the problem domain to operate in it. So for their attack they used a combination of PD and Non-Problem Domain Natural (NPDN) samples to extract the victim model f_v.

The attack is separated into two steps. In the first step, the adversary generates the so-called fake dataset $\tilde{D}$. The fake dataset D_a is created from the unlabeled samples that the attacker prepares before the attack, which are labeled by the victim model f_v. The authors call the fake dataset labels created by f_v as *stolen labels*. This fake dataset $\{\mathbf{x}, f_v(\mathbf{x})\} \in \tilde{D}$ should capture the victim model's feature space f_v, allowing a different network to be trained and achieve similar performance as the original. After the attacker successfully creates the fake dataset $\tilde{D}$, the substitute model f_s is trained from scratch with a standard training approach. Figure 3.4 shows the flowchart of the attack.

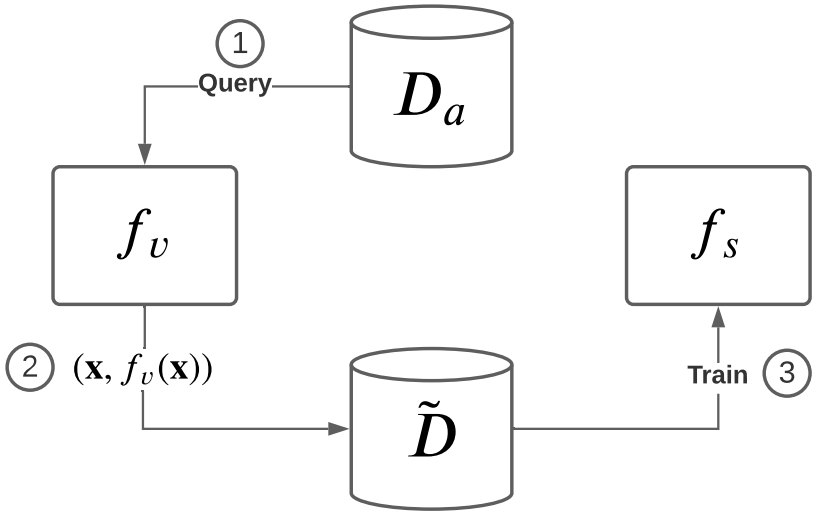


Figure 3.4: Flowchart of CopyCat attack.

The attack was tested on both victim and substitute models using the VGG16 [20] architecture. The tests were performed over three distinct problem domains: *Facial Expression Recognition (FER)*, *General Object Classification (GOC)*, and *Satellite Crosswalk Classification (SCC)*. For GOC CIFAR10 [33], STL10 [34] datasets were used. In the case of FER and SCC, multiple different datasets were used, but we refer to [27] for the specific list of datasets for brevity. For all the problem domains the whole ImageNet dataset [35] was used for NPDN samples. The authors tested

multiple types of D_a in their paper (e.g., only PD samples, PD and NPDN samples). The results showed that the substitute models could achieve good accuracy similar to that of the victim model in all cases but the combination of PD and NPDN samples showed the best results.

The authors also compared the substitute models' performance on the fake dataset with that of substitute models trained only on PD samples with the original labels. The substitute models trained using a fake dataset achieved better accuracy than the models with PD samples with original labels. This result shows that to copy a victim model, it is more important to have its predictions (together with its mistakes) than to have a smaller version of the problem domain dataset (with correct labels).

This work was the first that showed the possibility of copying a victim DNNs model with random unlabeled data taken from the Internet. The attack's main problem is that it does not optimize its query selection; it simply queries all of the available samples of the adversary's dataset to the victim model and lets it annotate it. The authors' threat model assumes only black-box access to the model and requires only labels as the victim model's output.

3.4.3 ActiveThief Attack

The *ActiveThief* framework proposed by Pal et al. [10], leverages Active Learning techniques to perform model extraction attacks. The paper focused on using non-problem domain unlabeled datasets to extract the victim model. The framework uses four different pool-based active learning sampling strategies to perform the extraction. Their proposed attack follows the typical pattern of active learning. The adversary initially selects k random samples $\mathbf{x}_j \in D_a$, $j \in \{1, k\}$ for the query set Q_1 and these samples are labeled by the victim model f_v. These labeled samples $(\mathbf{x}_j,\ f_v(\mathbf{x}_j))$ are added to the labeled dataset $\tilde{D}_1$. Afterwards the attack runs for σ iterations, where the substitute model f_s is retrained in each iteration from scratch with the current labeled dataset $\tilde{D}_i$, $i \in \{1 \dots \sigma\}$. After the training of f_s, the newly learned substitute gives predictions for unlabeled samples from D_a. The authors call these predictions *approximate labels*. Afterwards, an AL technique is used to score the samples with the approximate labels and selects the k best ones for the query set Q_{i+1} that is used to query f_v. The samples from Q_{i+1} and the retrieved labels are then added to the labeled dataset $\tilde{D}_{i+1}$. The whole process is depicted in Figure 3.5.

The four AL techniques used in the paper were *entropy*, *k-center*, *DFAL* and *DFAL+k-center*. The *entropy strategy* is based on the uncertainty strategy which calculates the entropy of f_s prediction on each unlabeled sample in D_a:

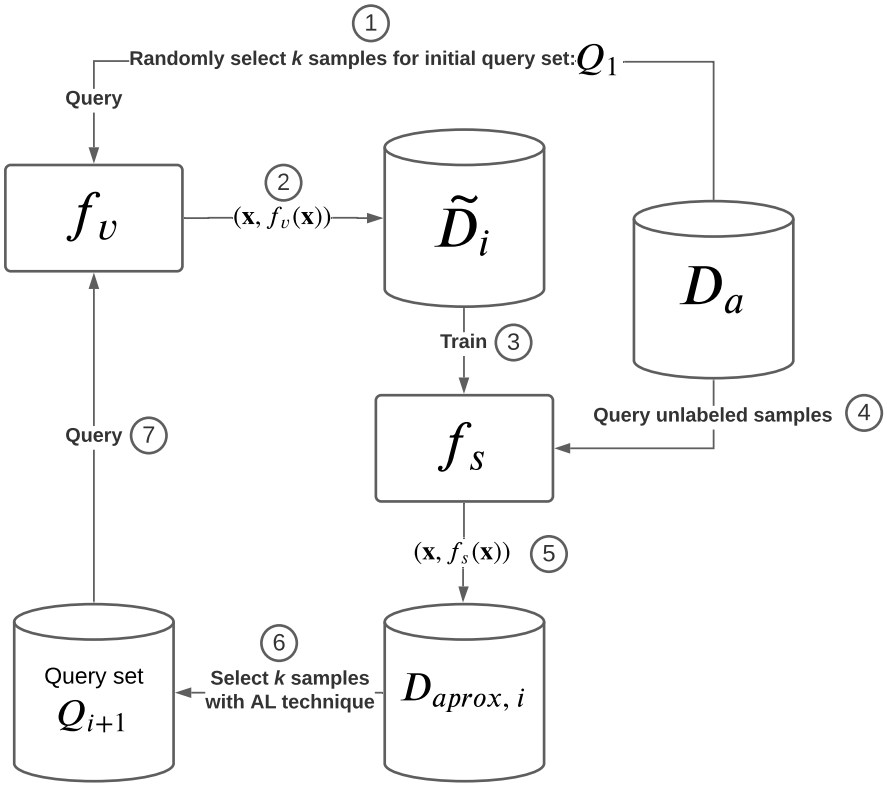


Figure 3.5: ActiveThief attack flowchart

$$H_n = -\sum_{j=0}^{C} f_s(\mathbf{x}_n)_j \log(f_s(\mathbf{x}_n)_j) \tag{3.4}$$

where C is number of classes in the classification problem and x_n represents the nth unlabeled sample in D_a. After calculating the entropy, k samples with the highest entropy values i.e., the samples on which the substitute model is most uncertain, are selected for the query set Q_{i+1}.

The *k-center strategy* uses the greedy k-center technique [36] that tries to select the most distinct unlabeled samples from D_a. The samples in the query set $\tilde{D}_i$ are used as the initial cluster centers for the algorithm, with labels from f_s. Afterwards, in each iteration, the algorithm selects the most distant unlabeled sample from D_a from all current centers:

$$(\mathbf{x}^*, f_s(\mathbf{x})^*) = arg \max_{(\mathbf{x}_n, f_s(\mathbf{x}_n)) \in \tilde{D}_a} \min_{(\mathbf{x}_m, f_s(\mathbf{x}_m)) \in \tilde{D}_i} \|f_s(\mathbf{x}_n) - f_s(\mathbf{x}_m)\|_2^2 \tag{3.5}$$

where x_n represents the nth unlabeled sample from D_a and x_m represents the mth sample from labeled dataset $\tilde{D}_i$. the selected sample $(\mathbf{x}^*, f_s(\mathbf{x})^*)$ is then added to the $\tilde{D}_i$ as new center. This process is repeated k times to find the other samples for query set Q_{i+1}.

The *DeepFool Active Learning (DFAL)* strategy uses the DeepFool adversarial attack [37] to perturb each unlabeled sample $\mathbf{x} \in D_a$ until f_s misclassifies it. Then the peturbation on the sample is calculated $\alpha_n = \|\mathbf{x}_n - \hat{\mathbf{x}}\|_2^2$ and the k samples with the smallest peturbation α_n are selected to the $\tilde{D}_{i+1}$. The idea behind DFAL technique is that the samples with the smallest perturbation are closest to the decision boundary of f_s.

The last strategy *DFAL+k-center* first selects l unlabeled samples from D_a using the DFAL strategy and from these, k samples are selected with the k-center strategy. The reasoning behind this combination is that while k-center selects the most distinct samples, it does not necessarily select the most informative ones. On the other hand, the DFAL selects the most informative, but it might not select the most diverse samples. With the combination of these strategies, the deficiencies should disappear, and the samples are expected to have both good diversity and be informative at the same time.

The authors tested their framework against CNN models using the MNIST [31], the CIFAR10 [33] and the GTSRB [32] datasets. The architectures for victim and substitute models were simple custom CNNss. A subset of ImageNet [35] was used as D_a . The authors also conducted tests on text classification datasets. In both cases, their results showed that all active learning techniques outperformed the simple random strategy and achieved good fidelity extraction of the victim model. The authors also tested the transferability of their adversarial samples crafted by their substitute models. The threat model of their attack assumes only black box access to the model and probabilities as the output of f_v, however the attack showed good results even with only labels as output.

3.4.4 KnockOff-Nets Attack

The *KnockOff-Nets* attack was proposed by Orekondy et al. [5] and it is based on a reinforcement learning approach. Instead of the adversary trying to find the most informative samples of D_a, the attack leaves this work to the substitute model itself. The authors' attack is based on the reinforcement learning policy of gradient bandit algorithm [14] and their research focused on stealing image classification CNN models. Before the attack the adversary needs to obtain a labeled dataset D_a, which may consist of both PD and NPDN samples. The labels of the dataset are

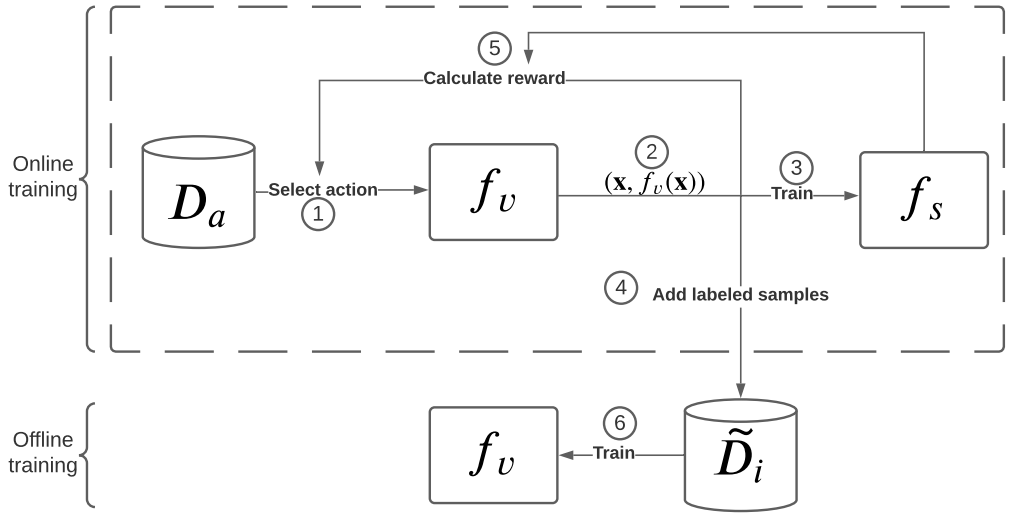


Figure 3.6: KnockOff-Nets attack flowchart

used to represent the actions a, which the model can take. The attack itself can be separated into two main steps according to the training of the substitute model f_s. The *online training* part and the *offline training* part. During the online training part the attack samples the D_a over σ iterations. In each iteration t action $a_t \in A$ is selected and k samples $\mathbf{x}_t$ corresponding to the action are queried to the victim model f_v and labeled. The f_s is then trained on these new samples and a reward is calculated afterwards for each of them. In 3.6 we depict the attack's flowchart.

The proposed rewards for the attack are *certainty*, *diversity* and *loss*. The certainty reward uses a margin-based certainty measure, which tries to promote samples that the f_v is most certain about. The reward is calculated for each selected sample in the iteration $\mathbf{x}_t$ as the difference between the class with the highest probability and the second highest probability:

$$R^{cert}(f_v(\mathbf{x}_t) = P(f_v(\mathbf{x}_t)_1|\mathbf{x}_t) - P(f_v(\mathbf{x}_t)_2)|\mathbf{x_t}) \tag{3.6}$$

The diversity reward tries to encourage the attack to explore different actions a to prevent the repeated selection of the same label and is calculated as:

$$R^{div}(f_v(\mathbf{x}_{1:t})) = \sum_{c=0}^{C} \max(0, avg(f_v(\mathbf{x}_t)_c) - avg(f_v(\mathbf{x}_{t-\Delta})_c)) \tag{3.7}$$

where $avg(f_v(\mathbf{x}_t)_c)$ represents the mean probability of class c in the current iteration and $avg(f_v(\mathbf{x}_{t-\Delta})_c))$ represents the mean calculated over the past Δ iterations.

Lastly, the loss reward encourages samples on which the prediction of f_s does not correspond to the prediction of f_v:

$$R^{\mathcal{L}}(f_s(\mathbf{x}_t), f_v(\mathbf{x}_t)) = \mathcal{L}(f_s(\mathbf{x}_t), f_v(\mathbf{x}_t)) \tag{3.8}$$

After calculating the reward r_t, the attack updates the potentials of each action $a \in A$ with the gradient bandit algorithm [14]:

$$H_{t+1}(a) = \begin{cases} H_t(a) + \alpha(r_t - avg(r_{t-\Delta}))(1 - P_t(a)) & a = a_t \\ H_t(a) - \alpha(r_t - avg(r_{t-\Delta}))P_t(a) & a \neq a_t \end{cases} \tag{3.9}$$

where $\alpha > 0$ represents a step-size parameter and $avg(r_{t-\Delta})$ is the average of all rewards up to $t-1$. The action preferences $H(a)$ are initially all set to zero, which means that each action has same probability of being selected. The $avg(r_{t-\Delta})$ is used as a baseline which measures if the current reward is higher or lower than the average reward. This influences whether the potential should be increased or lowered for a_t. The potentials for the rest of the actions a move in the opposite direction.

The probabilities for each action $P_t(a)$ in iteration t are calculated by the softmax function applied to the potentials:

$$P_t(a) = \frac{e^{H_t(a)}}{\sum_{i=0}^{m} e^{H_t(a_i)}} \tag{3.10}$$

, where m is the number of possible actions. This is repeated for σ iterations, until the attack's budget B is exhausted. Following this, the offline part of the attack begins, where f_s is trained from scratch on all selected samples and the final substitute model is created.

The authors tested the attack using ResNet34 [19] as both the victim and substitute model. The tests were conducted over multiple image classification datasets, which included Indoor67 [38] and Caltech256 [39]. Random NPDN samples were taken from

the ImageNet [35] and OpenImages [40] datasets as D_a. In some of the experiments they were also used in combination with PD samples from the tested datasets. Their tests were focused primarily on the task accuracy of f_s. The results showed that the substitute model f_s, was able to achieve very good accuracy, in some cases even higher than f_v. One of the interesting results of their experiments was that the f_s trained with the same samples as the victim f_v but with labels from f_v can achieve better accuracy than the victim model itself. Their threat model assumes that the adversary has only black box access to the victim model and is able to only query f_v and read its outputs. The victim model's output in their experiments is assumed to be a full probability vector.

3.4.5 BlackBox Ripper Attack

BlackBox ripper attack was proposed in [6] and its main focus was attacks against CNN models used for image classification. The attack uses a different approach when constructing the adversary dataset D_a. All of the previously mentioned attacks required that the adversary has a dataset D_a, which they prepare prior to the attack and they use it during the attack. BlackBox ripper tries a different approach with the use of Generative Adversial Network (GAN) models. The adversary first trains a GAN model, which does not necessarily have to generate samples corresponding to the classes that the victim model f_v was trained to predict. Then through an evolutionary algorithm, the GAN creates samples that the f_v predicts with high confidence for a specific class $c \in C$. The substitute model f_s is then trained on these labeled synthetic samples. The whole process is visualized in Figure 3.7.

The evolutionary algorithm traverses the latent space to find the optimal samples for each class of f_v. The algorithm runs over multiple iterations, selecting latent vectors with which the GAN model generates images with the highest probability for class c. The targeted class c is selected randomly before starting the algorithm. The search stops when one of the samples achieves a loss smaller than a threshold value t, where the loss is calculated as the mean squared error between the desired prediction of f_v and the prediction for the sample by f_v. The desired prediction is the one-hot encoded output with the target class c being 1.

The authors tested the attack on multiple image classification datasets, including CIFAR10 [33]. In their experiments, they used a Generative Adversial Networks (GANs) that they trained themselves and also used pre-trained ones available online. The victim and substitute models used multiple different popular CNN architectures. The focus of the tests was on the task accuracy of the substitute model and only black-box access to the victim model. They were able to achieve good accuracy for the substitute model. They even achieved higher accuracy than the KnockOff-Nets attack [5], which they used to compare the different approaches to model extraction

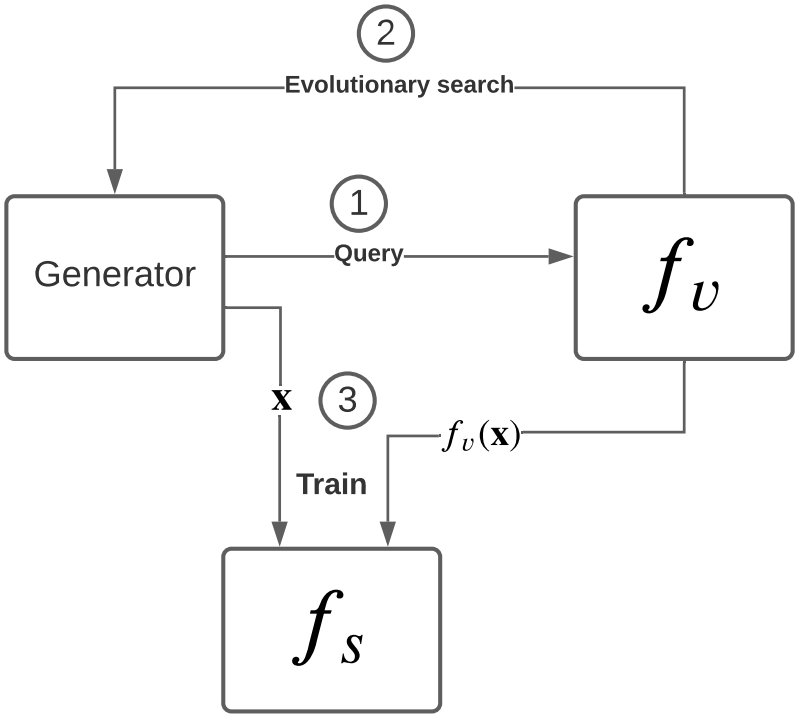


Figure 3.7: Flowchart of BlackBox ripper attack

attacks. There is, however, a problem with the query efficiency of the attack, which the authors mention in their paper.

Chapter 4

Model Extraction Tool

The following chapter is dedicated to the detailed description of Model Extraction Tool (MET). MET through its easy-to-use API and built-in attacks, datasets, and models, allows easy experimentation and testing of existing model extraction attacks. It also has all the necessary building blocks for easy creation and prototyping of new attacks. The goal of MET is to make it easier to try state-of-the-art model extraction attacks and compare them in different settings. We also try to provide the necessary building blocks to create new attacks. The tool can be used not only by researchers that want to compare attacks under similar settings, but also by anyone that wishes to test their model and see how susceptible it is to model extraction attacks. The MET is available on GitHub [41].

As mentioned in Chapter 1, model extraction attacks are in their exploratory stage [2], and there is still no consensus on what test datasets should be used, nor which model architectures are the best for the experiments. These issues result in complications when comparing the results from the previous work. The situation is made even more challenging since not all papers release source code with the original implementations of the attacks. MET aims to fix these issues.

4.1 Design Requirements of MET

Since the goal is to make it easy for anyone to try and experiment with model extraction attacks, the following requirements were followed during the tool's design. The tool:

1. Must allow the user to configure individual attacks easily.
2. Must contain examples of the attacks.
3. Should work on any major Operating System (OS).
4. It should be possible to use a Graphical Processing Unit (GPU) if it is present in the system.
5. Should contain models and datasets, which can be used to bootstrap user experiments.
6. Must allow the use of custom datasets.
7. Must allow the use of custom model architectures.
8. Must allow testing attacks on user-supplied machine learning models.
9. Should allow testing models regardless of the ML framework used.

MET meets all the above requirements, except for 8. We support arbitrary ML frameworks but only in the case of the victim model. For the substitute model, the support is limited only to models implemented with PyTorch [42] at the moment. The main reason for this limitation was that the primary focus of this work was on model extraction attacks applied to DNNs and PyTorch was the framework of choice for implementing them.

4.2 Used Libraries

The two primary libraries around which the framework is built are PyTorch [42] and Pytorch-lightning [43]. PyTorch is an optimized machine learning library for deep learning using Graphical Processing Units (GPUs) and Central Processing Units (CPUs). We chose PyTorch because the syntax is easy to read, and it is more pythonic than Tensorflow. Moreover, we considered the introductory tutorials of PyTorch to be well written, which makes the whole library more accessible. Another reason for this choice was the increasing popularity of the PyTorch framework in recent years. The reason we use Pytorch-lightning is that it is primarily used to remove a lot of the boilerplate code required by Pytorch for the training and testing of the models. It also allows to easily add support for multiple GPUs, different visualization and logging libraries, and training and testing to be run deterministically.

MET's last big library is Torchvision [44], which is the Pytorch library dedicated to computer vision tasks. The library contains multiple pretrained models for vision

tasks and several popular datasets. It also allows for easy manipulation with custom vision datasets through its API.

Other important libraries that are used in MET are FoolBox [45, 46] and Mimicry [47]. The FoolBox library implements multiple evasion attacks, and it has a good performance in terms of speed and creating adversarial samples. We used the FoolBox library for some of the attacks that use adversarial training based on evasion attacks (e.g., BlackBox attack, ActiveThief attack), and implementing these attacks is out of the scope for this thesis.

The mimicry library is used because the BlackBox ripper attack requires a GAN or a VAE to function. MET provides support for pretrained GANs models from the Mimicry library, which aims to provide easy reproducibility of GAN research and has several pretrained ones.

4.3 Example Usage of MET

Before describing the implementation and structure of MET in more detail, we present a step-by-step example of how to use the tool. We go over how to import and initialize the tool, configure the attacks, make arbitrary models and datasets compatible with the tool, and finally, how to run the attack.

The basic steps of using the tool are the following:

1. Import the desired attack and VictimModel, TrainableModel wrapper classes.
2. Make the datasets and models compatible with the PyTorch API.
3. Wrap the victim model into the VictimModel class and the substitute model into the TrainableModel class.
4. Initialize and configure the attack.
5. Run the attack.

As an example, we shall use the Ember dataset [48] and the model that accompanies the dataset. Ember is a dataset for training ML models to detect malicious Windows portable executable files. It has $800,000$ training samples ($300,000$ malicious,

300,000 benign, 300,000 unlabeled) and 200,000 test samples (half malicious, half benign) [48].

As a victim model f_v we use the LightGBM [49] model that is provided as a baseline for Ember and is trained on the labeled samples of the dataset. For the substitute model f_s we shall use a DNN implemented in PyTorch.

Finally, the attack in our example is ActiveThief with the k-center method. For the adversary dataset D_a, we shall use the unlabeled samples from the Ember dataset. To test the performance of the substitute and victim models, we use the test samples from Ember. The example is also provided as part of the tool and can be found under the `experiments` folder in `gbm_ember2018.py` script.

4.3.1 Necessary Preparations Before Using MET

To use MET the user first has to import one of the available attacks. Additionally, they need to import the *VictimModel* and *TrainableModel* classes, which are wrapper classes used to make the victim and substitute models compatible with MET. The code showing the imports is depicted in Figure 4.1.

```
from met.attacks import ActiveThief
from met.utils.pytorch.lightning.module import (
    TrainableModel,
    VictimModel,
)
```

Figure 4.1: Example of basic imports for MET.

Afterwards, the user must prepare the test set D_t and the thief dataset D_s. The only requirement to create a custom dataset compatible with the tool is to make the dataset compatible with PyTorch. There are multiple ways of how this can be done. The user may create a custom dataset class from scratch that is a subclass of `torch.utils.data.Dataset`. If the user's dataset contains images, the user can also use one of the generic data loader classes from Torchvision `torchvision.datasets.ImageFolder` and `torchvision.datasets.DatasetFolder`.

In the case of the Ember dataset, which we load using NumPy, we can use the built-in *NumpyDataset* class in MET, which offers mapping from a NumPy dataset to a PyTorch dataset. The *NumpyDataset* class can be found in `met.utils.pytorch.datasets`.

The whole conversion of the Ember dataset into a compatible dataset with MET is shown in Figure 4.2.

```
# Load training set
X_train_path = Path(data_dir).joinpath("X_train.dat")
y_train_path = Path(data_dir).joinpath("y_train.dat")
y_train = np.memmap(y_train_path, dtype=np.float32, mode="r")
N = y_train.shape[0]
X_train = np.memmap(
            X_train_path,
            dtype=np.float32,
            mode="readwrite",
            shape=(N, 2381),
        )
# Read unlabeled samples
train_rows = y_train == -1
X_train = X_train[train_rows]
y_train = y_train[train_rows]

# Load test set
X_test_path = Path(data_dir).joinpath("X_test.dat")
y_test_path = Path(data_dir).joinpath("y_test.dat")
y_test = np.memmap(
            y_test_path,
            dtype=np.float32,
            mode="readwrite",
        ).astype(np.int32)
N = y_test.shape[0]
X_test = np.memmap(
            X_test_path,
            dtype=np.float32,
            mode="readwrite",
            shape=(N, 2381),
        )

adversary_dataset = NumpyDataset(X_train, y_train)
test_set = NumpyDataset(X_test, y_test)
```

Figure 4.2: Conversion of Ember dataset into compatible dataset for MET.

The next step is to prepare the victim and substitute models. For a victim model to be compatible with the tool, the user only needs to make sure that it is compatible with PyTorch's API. What this means is that the model should be wrapped in the class which inherits from `torch.nn.Module` and implements the `forward` method, which takes a PyTorch tensor as input and returns a PyTorch tensor as output. An

example of using an example of the conversion of a LightGBM victim model into a PyTorch model is presented in Figure 4.3. As for the substitute model, it must be a PyTorch model and return the raw logit output. An example of PyTorch model that we are using in the example is in Figure 4.4.

```
class Ember2018(nn.Module):
    def __init__(self, model_dir: str, seed: int):
        super().__init__()
        model_file = str(Path(model_dir)
                    .joinpath("ember_model_2018.txt"))
        self.ember = lgb.Booster(
                        params={"seed": seed},
                        model_file=model_file,
                        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        input_ = x.detach().cpu().numpy()
        y_preds = self.ember.predict(input_)
        y_preds = y_preds.astype(np.float32)
        return torch.from_numpy(y_preds).to(x.device)
```

Figure 4.3: Example transformation of LightGBM Ember model into PyTorch model.

```
class EmberSubsitute(nn.Module):
    def __init__(self, scaler, return_hidden=False):
        super().__init__()
        self._layer1 = /***/
        /***/
        self._final = /***/
        self._scaler = scaler

    def forward(self, x):
        x_scaled = self._scaler.transform(
                x.cpu().numpy()).astype(np.float32)

        /***/

        logits = self._final(hidden)
        return logits
```

Figure 4.4: Substitute model used in Ember example. We are showing only abbreviaton of the model.

Once the model classes have been defined, both the victim and substitute models

can be wrapped into the VictimModel and TrainableModel classes, respectively. The VictimModel class supports multiple different output types specified through a parameter, namely: *one-hot*, *logits*, *labels*, *probability*, and *rounded*. The TrainableModel class requires that the loss function and the optimizer are passed as arguments for the training of the substitute model. Figure 4.5 shows the initialization of the wrapper classes.

```
ember_model = Ember2018(args.ember2018_model_dir, args.seed)
victim_model = VictimModel(ember_model, NUM_CLASSES, "raw")

ember_substitute = EmberSubsitute(scaler)
substitute_model = TrainableModel(
    ember_substitute,
    NUM_CLASSES,
    torch.optim.Adam,
    torch.nn.BCEWithLogitsLoss(),
)
```

Figure 4.5: Use of the MET wrapper classes for both victim model and substiute model. The "raw" parameter in VictimModel class is the option to return the unmodified victim model's output.

4.3.2 Attack Initialization and Execution

After the models and datasets are in a compatible format, the only thing left to do is to call the attacks themselves. All attacks are located inside `mef.attacks`. Each attack requires for its initialization the victim and substitute models wrapped in `VictimModel` and `SubstituteModel` classes, respectively. The rest of the parameters depend on the type of attack.

The parameters of the attacks also contain settings related to the training of the substitute model and settings for the tool itself. These settings and attack settings can also be accessed through the settings classes (`AttackSettings`, `BaseSettings`, `TrainerSettings`), which are accessible as attributes of each attack.

Finally, the attack can be called using the `__call__` method, which expects datasets as input in the following order: the adversary dataset, followed by the test set. Optionally, the validation set can also be passed as a third argument. In Figure 4.6 we show the initialization and configuration for the ActiveThief attack together with a call to start the attack.

While running, the attacks create and update a log file. The log file contains both

```
attack = ActiveThief(
            victim_model,
            substitute_model,
            args.selection_strategy,
            args.iterations,
            args.budget,
        )

/***/

# Accesssing settings through attack's attributes
# Trainer settings
attack.trainer_settings.training_epochs = args.training_epochs
attack.trainer_settings.precision = args.precision
attack.trainer_settings.use_accuracy = args.accuracy

# Start the attack
attack(adversary_dataset, test_set)
```

Figure 4.6: Initialization of ActiveThief attack. In this example we show both ways of accesing the settings of the MET and the attack, i.e., through attributes of the attack class or as arguments during initialization.

details about the attack and the final results of the attack. The log is saved to the specified save location, which by default is `./cache`.

The final substitute model created by the attack can be found in the `substitute` folder in the `final_substitute_model.pt` file. The file contains the state dictionary of the substitute model, allowing its easy distribution and deployment.

Finally, every attack also comes with built-in support for python's argparse module, which can be accessed through the `get_attack_args` method. This allows easy creation of scripts with all of the possible settings of the attack accessible through the Command Line Interface (CLI). The example call from the CLI for the `gbm_ember2018.py` script with the ActiveThief attack is shown in Figure 4.7.

```
python gbm_ember2018.py \\
--selection_strategy k-center \\
--gpu 0 --save_loc "./cache/ember2018/" \\
--batch_size 256 --budget 20000 \\
--iterations 10 --num_workers 16 ...
```

Figure 4.7: Example of accesing the ActiveThief arguments through CLI. Every attack has builtin support for Python's argparse module allowing easy access to all the settings through CLI.

4.4 Design

In the following section, we present a high-level overview of the MET's structure and its most essential parts in detail. The design of the tool focuses primarily on model extraction attacks. We originally planned for MET to automatically do all the necessary compatibility conversions for both models and datasets design. However, this proved to be too unsustainable. It forced us to focus more on automation than on the implementation of the attacks. As such, our final design is much more minimalistic and straightforward. It contains only the essential parts needed to run and test model extraction attacks. It also offers some additional utility classes and functions to make running of the user's experiments easier e.g., classes for popular datasets and popular model architectures. It requires that the user does the necessary preprocessing of both models and datasets to make it compatible with MET.

4.4.1 Structure

The tool is separated into multiple parts, which are visualized in the high-level tree view of the MET folder structure in Figure 4.8.

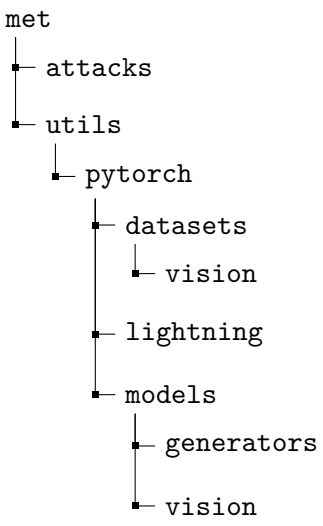


Figure 4.8: MET folder structure

The code is structured in an intuitive way and is easy to follow and use. A brief description of each part of the tool is as follows:

- `met.attack` - Main part of the whole module containing all model extraction

attacks.

- `met.utils` - Folder containing utility functions and classes, which make it easier to perform experiments and to make the models compatible with the tool.
- `met.utils.pytorch` - Utility functions and classes dedicated to PyTorch framework.
- `met.utils.pytorch.datasets` - Provides PyTorch compatible datasets, which can be used for experiments. Currently, only computer vision datasets are provided.
- `met.utils.pytorch.lightning` - Contains classes, which are based on Pytorch-lightning library.
- `met.utils.pytorch.models` - Model zoo currently containing some popular models and generators related to computer vision.

In the following sections we provide detailed descriptions of the most important parts in the current version of the tool that are located in the `met.attacks` and `met.utils.pytorch.lightning` modules. We will focus specifically on the `MetModel` and `AttackBase` classes which form the core of the whole tool.

4.4.2 AttackBase Class

All implemented attacks inherit from a class called `AttackBase` and that is intended to be used as basis for all attacks. It is an abstract class, which contains some already implemented methods that are useful in all different attacks (e.g., substitute model training, model testing, calculating the performance of the attack). This way, the attack class needs only to contain code specific for that attack. The class itself abstracts from the attacks most of the utility classes and methods provided by the tool, which means it is not necessary to know them in detail to implement a new attack. The `AttackBase` class Unified Modeling Language (UML)[1] diagram is depicted in Figure 4.9. Since the whole UML class diagram of the tool is too big to fit into the document, we will be presenting only parts of it.

[1] https://www.uml-diagrams.org/class-diagrams-overview.html

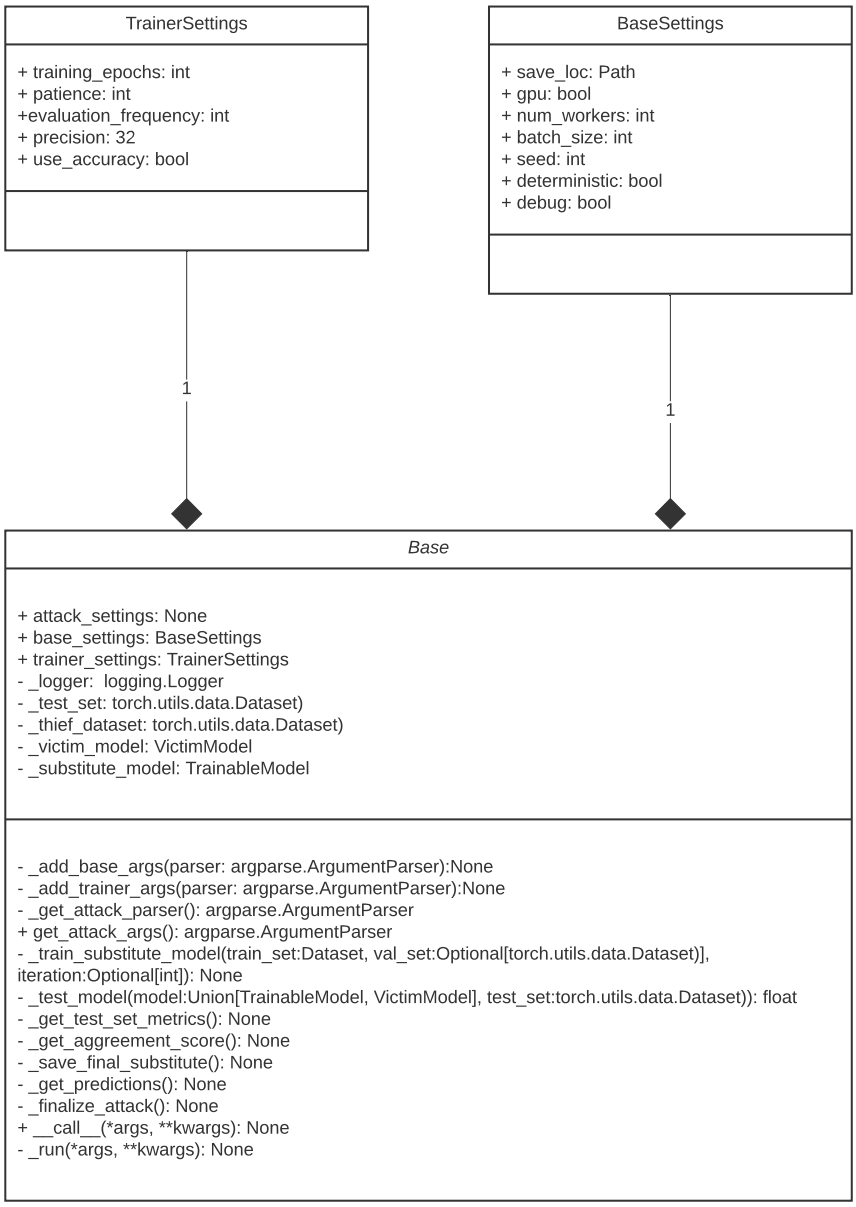


Figure 4.9: UML representation of `AttackBase` class

From the class diagram, we can see that most of the methods and attributes are quite self-explanatory from their given names. The ones that need a more detailed explanation are `_thief_dataset`, `base_settings` and `trainer_settings` attributes. The `_thief_dataset` represents the D_s used by the adversary for model extraction attacks. As for the other two, they contain instances of the `BaseSettings` and `TrainerSettings` classes respectively, which encapsulate the general settings of the tool and the settings related to substitute model training.

The methods which are required to be implemented by each attack are `_run` and `_get_attack_parser`. The former is called in `__call__` method and should contain the attack's main loop as for the latter method, it is used to initialize the argument parser with the parameters of the attack for use in the CLI.

4.4.3 MetModel Class

The second most important class of the tool is the `MetModel` class. It is a parent class to both `TrainableModel` and `VictimModel`, which are used throughout the tool. They contain the substitute model f_s and the victim model f_v that the user provides together with relevant information about them. The intention with these classes is to have all the necessary information and functionality of the models in one place. The class is based on Pytorch-lightning's `LightningModule`, which is based on PyTorch normal `torch.nn.Module` class with additional methods and options.

The way Pytorch-lightning works is that it abstracts away the writing of the training, testing, and validation loops, which is normally necessary to implement in PyTorch. The only thing that needs to be implemented is the relevant training, test, and validation step in the model class which inherits from `LightningModule`.

The `MetModel` class only implements the general validation step and both the `MetModel` and `TrainableModel` implement that `_shared_step_output` that does the necessary preprocessing of the model output. The class also contains the prepared F1-macro and accuracy metrics for model testing and training.

In the case of the `VictimModel` the class also offers built-in output formatting, which consists of *sigmoid*, *softmax*, *labels*, *one_hot* and for unmodified output *logits* and *raw* options. This gives the user the ability to test the attacks with different victim models output types. As for the `TrainableModel` it contains the implementation of the general supervised learning training step. Thanks to the flexibility of Pytorch-lightning the training step can be easily modified to support an arbitrary training regime. The UML representation of `MetModel` class is represented in Figure 4.10, together with both `TrainableModel` and `VictimModel`.

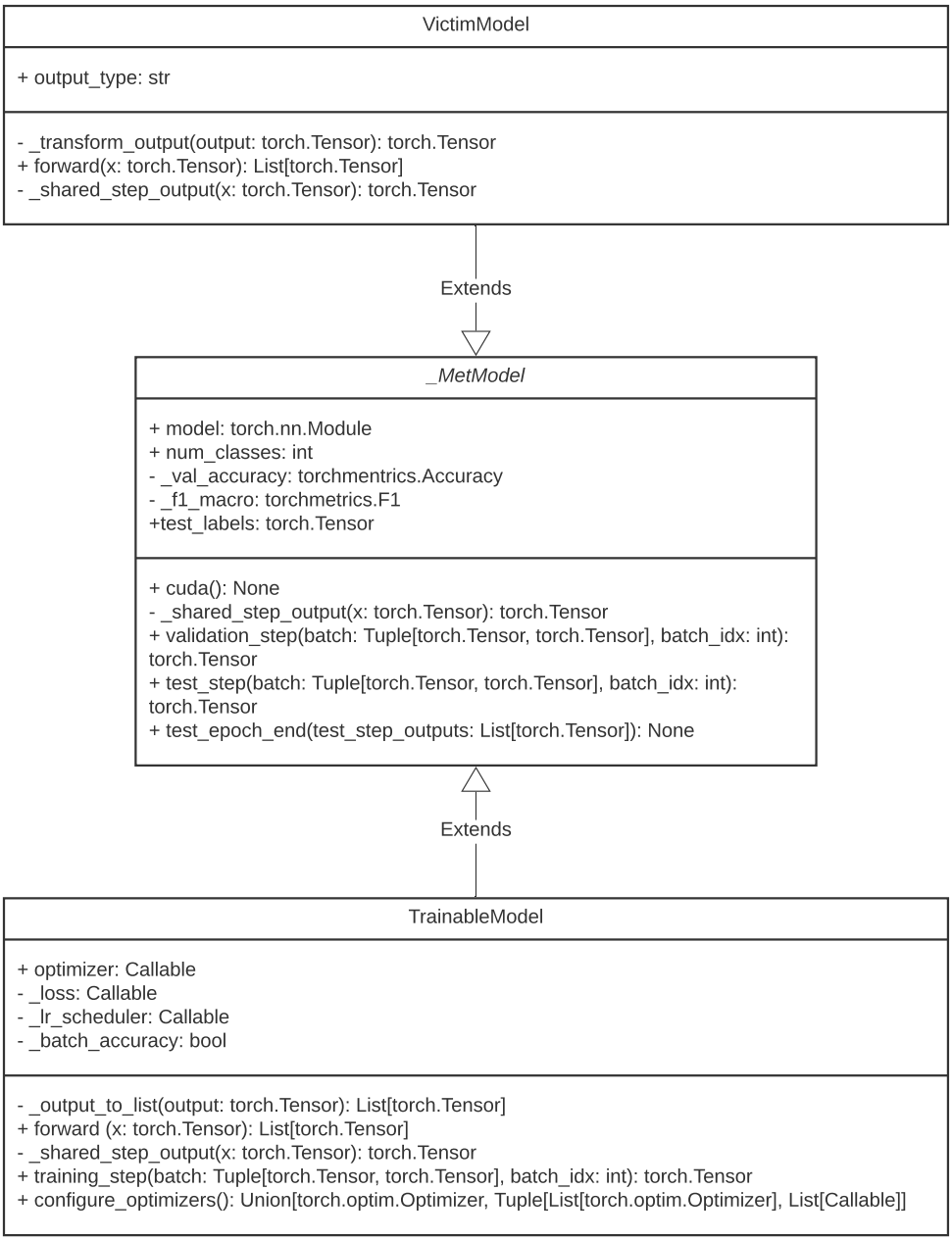


Figure 4.10: UML representation of `MetModel` class

4.5 Comparison with Other Tools

There are a few other libraries online which implement model extraction attacks. This section is dedicated to the comparison of these libraries with MET. The libraries we will discuss are Adversial Robustness Toolbox (ART) [8] and PrivacyRaven [9].

ART is a huge library dedicated to Machine Learning security and privacy. It implements model extraction attacks and other privacy and security attacks such as inference attacks, poisoning attacks, and evasion attacks. It also supports all popular frameworks, dataset types, and Machine Learning tasks. When it comes to model extraction attacks, ART provides implementations of the Copycat attack [27], KnockOff nets [5] and the Functionally Equivalent Extraction [3].

The PrivacyRaven serves a similar purpose as ART, but unlike ART, which implements both privacy and security attacks, PrivacyRaven focuses only on privacy ones. When it comes to model extraction attacks, the currently implemented attacks are the CopyCat attack and partially the CloudLeak attack [28]. The library internally uses ART for evasion attacks and some utility functions.

In terms of implemented attacks, both tools implement the CopyCat attack, which is the basic model extraction attack that can be performed, i.e., the adversary queries D_a to the victim model. While the KnockOff-Nets implementation in MET is based on the implementation available in ART but our implementation is more comprehensive and closer to the original paper. MET supports a variable number of samples during online training when using the adaptive strategy. It also prevents selecting the same samples during online training, and adaptively changes the number of possible actions when there are no more samples for some actions.

The Functionally Equivalent Extraction attack [3] which is only implemented in ART is limited to attacking simple NN architectures and does not meet our requirements as mentioned in Section 3.2. We also considered implementing the CloudLeak attack, but the paper introducing the attack was missing information about the test settings. We contacted the authors to obtain the missing details, but did not get a response from them. Overall, MET implements additional model extraction attacks, which are not available in either of the other libraries and improves existing implementations. To the best of our knowledge, we currently have the largest number of implemented model extraction attacks in one single tool.

Chapter 5

Verification of Implemented Attacks

This chapter compares the MET attacks' results to the original papers set-ups to confirm the implementations' correctness. We performed this comparison to test that the MET is indeed working correctly and that our implementations of the attacks show similar results to the ones reported by the authors. We provide all the necessary details to reproduce each experiment with MET. We ran all experiments on a PC with Intel I9-9900K CPU, an NVIDIA TITAN V GPU, and 32GB of RAM, running Ubuntu 18.04.5 LTS with CUDA Toolkit 11.03 installed.

We selected a single experiment setting from each corresponding paper to compare the performance between our implementation of the attacks and the original implementations. The reason for selecting only one experiment from each paper was time constraints since some attacks take a relatively long time to perform. We also wanted to have an equal number of comparisons for each attack. In the following subsection, we provide information for each comparison.

We omit the comparison for the CopyCat attack. The CopyCat attack is based on querying the victim model f_v with the samples from the adversary dataset D_a and using the predictions as labels for the synthetic dataset. The querying and training parts of the code are used by multiple attacks and are tested thoroughly. In addition, our code is quite similar to the original implementation that was also written in Pytorch. Finally, the experiments in the original paper that would have been used as a baseline for the verification require a large adversary dataset that was very hard to collect and test under the current time constraints. For these reasons, we chose to omit the comparison and extensive verification of the CopyCat attack.

2 × Convolution-3x3-s1-p1-32 MaxPool-2x2
2 × Convolution-3x3-s1-p1-32 MaxPool-2x2
2 × Convolution-3x3-s1-p1-64 MaxPool-2x2
2 × Convolution-3x3-s1-p1-64 MaxPool-2x2
2 × Convolution-3x3-s1-p1-128 MaxPool-2x2
2 × Convolution-3x3-s1-p1-128 MaxPool-2x2
FC-10

Figure 5.1: DNN architectures used in comparison experiments for ActiveThief attack. The Convolution-3x3-s1-p1-32 means a convolution layer with 3×3 kernel, stride of 1, padding of 1 and 32 output channels. FC stands for fully connected layer and the number represents number of output neurons.

5.1 Reproducibility of Results

Each of the experimental settings we mention in the following subsections has a corresponding script located in the `examples_paper` folder of the MET tool, which can be used to replicate our results easily. Each script provides CLI interface to modify the settings of the attack and contains a hardcoded seed for random generators, which is used for deterministic creation of datasets, model training, and attack execution.

5.2 ActiveThief Attack

For comparison, we selected the Cifar10 [33] dataset experiment from the respective work. In the original paper, the authors use a subset of $120,000$ samples from the ImageNet [35] dataset as the adversary dataset D_a. Unfortunately, they do not provide information about the exact subset they used. We chose a subset of the same size as the one reported by the authors in the paper. We created our D_a by taking the 120 samples from each of the 1000 classes of the ImageNet dataset. For the model architecture, they used a custom CNN architecture. The architecture is depicted in Figure 5.1.

Method	Victim Test Acc.		Substitute Test Acc.		Test Agreement	
	MET	Original	MET	Original	MET	Original
Random	80.8%	Unknown	61.9%	Unknown	66.8%	71.38%
Entropy	80.8%	Unknown	61.8%	Unknown	67.0%	72.99%
k-center	80.8%	Unknown	64.3%	Unknown	68.6%	72.97%
DFAL	80.8%	Unknown	62.6%	Unknown	67.6%	71.52%
DFAL+k-center	80.8%	Unknown	62.9%	Unknown	68.2%	73.47%

Table 5.1: Table with results for our implementation of the ActiveThief attack compared to the results reported in the original paper. Unknown represents values which the authors did not report in the original paper.

Both victim and substitute models are trained with the Adam optimizer with the default parameters of 1,000 epochs. Early stopping was used, and it was set to the patience of 100 epochs and a batch size of 150. The attack's initial seed size is 10% of the budget, and the validation set size is 20% of the budget. The budget is set to $20,000$ queries, and each active learning method uses the authors' recommended settings. The attack's performance is tested on the test set of the Cifar10 dataset. The comparison results can be seen in Table 5.1.

The results achieved using our implementations follow a similar pattern to the original baselines. The different attack methods beat simple random strategies and create substitute models with higher test accuracy and test agreement. However, the results achieved by our implementation are slightly lower than those reported in the original papers. We hypothesize that this discrepancy could be caused by the difference in the adversary datasets, initialization of model weights, and slight differences in the model architectures. In addition, to ensure that our implementation of active learning techniques in the ActiveThief attack is correct, we tested them in various active learning scenarios against other implementations available online.

5.3 BlackBox Attack

For the BlackBox attack, we did not use any of the experiment settings from the original paper. Instead, we used the attack settings from the paper Protecting against DNN Model Stealing Attacks (PRADA) [29], where the authors used the BlackBox attack to test the performance of their proposed defense against model extraction attacks. The reason is that the original paper experiments descriptions omitted some of the necessary information for easy reproduction, e.g., victim model architecture in case of the MNIST [31] experiment, dataset modification in case of the GTSRB [32] experiment. In the PRADA paper, the authors performed experiments with the attack on the MNIST dataset. We use their experimental settings and

Convolution-3x3-s1-p1-32 MaxPool-2x2
Convolution-3x3-s1-p1-64 MaxPool-2x2
FC-200
FC-10

Figure 5.2: DNN architectures used in comparison experiments for BlackBox attack. The Convolution-3x3-s1-p1-32 means a convolution layer with 3×3 kernel, stride of 1, padding of 1 and 32 output channels. FC stands for fully connected layer and the number represents number of output neurons.

Method	**Victim Test Acc.**		**Substitute Test Acc.**		**Test Agreement**	
	MET	Original	MET	Original	MET	Original
No synthetic samples	99.2%	±99%	85.1%	Unknown	85.2%	Unknown
Synthetic samples	99.2%	±99%	98.2%	Unknown	98.6%	±96%

Table 5.2: Table with results for our implementation of the BlackBox attack compared to the results reported in the original paper. Unknown represents values which the authors did not report in the original paper. ± is used for values, which are not reported as a number but only displayed as a symbol in a chart in the original papers without value.

results to compare our implementation of the attack. The used model architecture is depicted in Figure 5.2. The attack ran for 10 iterations. In each iteration, the substitute model was trained for 100 epochs with a batch size of 64. The lambda for the Jacobian augmentation was set to 64/255. The adversary dataset D_a was created by taking 10 samples from each class from a subset of the training set of the MNIST dataset, which was not used for the training of the victim model. To see the benefit of using synthetic samples, we also ran an attack with only the seed samples without the Jacobian augmentation. We report the results in Table 5.2.

The BlackBox attack implemented in MET with synthetic samples successfully creates a substitute model which achieves almost 99% test agreement with the victim model and achieves better results than the one reported in [29]. We can also see that the substitute model trained with synthetic samples achieves better results than the model trained with only seed samples. However, one must remember that the attack without synthetic samples uses only 100 samples. Still, the substitute model achieves almost 90% test accuracy and test agreement. This shows that the MNIST dataset is too easy to be used in model extractions experiments since with relatively simple CNN architecture and a small number of PD samples, it is possible to create a substitute model with good performance.

5.4 BlackBox Ripper Attack

For the BlackBox ripper attack, we selected the experiment on the FashionMNIST [50] dataset from the paper as a baseline. The authors use LeNet [51] as the victim model and Half-LeNet as the substitute model. For the generator component, the authors used SNGAN [52] which was trained on the Cifar10 dataset. We did not use their implementation of SNGAN in our experiment. Instead, we used the implementation provided by Torch-Mimicry and their pretrained weights on Cifar10 available for SNGAN. We use the Torch-Mimicry library since their implementations of GAN are thoroughly tested and are available with multiple pretrained weights.

Both the victim and substitute models are trained for 200 epochs with the Adam optimizer with default settings and the batch size set to 64. We want to note one difference between our experiment settings and the original one. Each epoch of substitute training consisted of $1,000$ batches of generated images for both the random strategy and the evolutionary strategy in the paper. In our case, to save time, we used only 100 batches per epoch for the evolutionary method and $1,000$ batches per epoch for the attack using a random strategy for generating images. The performance of the attack is tested on the test set of the FashionMNIST dataset. We report the results of the comparison in Table 5.3.

Method	**Victim Test Acc.**		**Substitute Test Acc.**		**Test Agreement**	
	MET	Original	MET	Original	MET	Original
Random	89.7%	89.9%	81.3%	80%	82.2%	Unknown
Optimized	89.7%	89.9%	Unknown	82.2%	Unknown	Unknown
Random $\frac{1}{10}$th	89.7%	89.9%	73.8%	Unknown	74.2%	Unknown
Optimized $\frac{1}{10}$th	89.7%	89.9%	75.3%	Unknown	76.5%	Unknown

Table 5.3: Table with results for our implementation of the BlackBox ripper attack compared to the results reported in the original paper. Unknown represents values which the authors did not report in the original paper and for which we did not run the experiments. The $\frac{1}{10}$th represents experiments, where only one-tenth of the original budget from the paper was used.

The implementation of BlackBox ripper reports better results for the random strategy than the original. As for the optimized method, we can see that the attack reports a similar trend as in the results in the original paper. The optimized method with an evolutionary algorithm creates a better substitute model than the random method. It is also worth pointing out that the test accuracy of the substitute models created with only $\frac{1}{10}$th of the budget is only worse by $\pm 6\%$ while using $11,520,000$ fewer samples.

5.5 KnockOff-Nets Attack

For the KnockOff-Nets attack, we chose the Caltech256 [39] dataset experiment from the original paper. One thing to note is that the original paper leveraged the hierarchical structure of ImageNet to create the adversary's dataset D_a for their experiments. We do not perform experiments with this structure and instead perform experiments in the attack setting that the authors call `adaptive-flat`. In this setting, the adversary uses the original labels of the ImageNet dataset. In the `adaptive-flat` experiments, we do not use the hierarchical structure since we argue that such a structure, as in ImageNet, cannot be created for every adversary dataset. For their experiments with `adaptive-flat` authors also assume that the adversary has access to all images on the internet. This meant that D_a contained the training samples from each dataset used in different experiments throughout the paper. However, since we performed the experiments only on Caltech256, we use only its training samples in our D_a. Our adversary dataset also does not contain samples from the OpenImages [40] dataset, which the authors also used in their experiments, since it is a large dataset and its downloading is a very time-intensive task.

The victim model and the substitute model use the ResNet34 [19] architecture with pretrained weights on the ImageNet dataset. For our experiments, we used the implementation available in the TorchVision library. The victim model is trained with an SGD optimizer for 200 epochs with a 0.1 learning rate and 0.5 momentum, where the learning rate is decayed by 0.1 every 60 epochs. The substitute model is trained with an SGD optimizer during online training with a learning rate of 0.0005 and batch size of 4, which is the number of new samples selected each iteration in the adaptive strategy. Afterward, during offline training, the substitute model is trained for 100 epochs with an SGD optimizer using a learning rate of 0.01 and momentum of 0.5 and uses the same learning rate decay as the victim model. The batch size is set to 64, and the attack performance is measured on the test set of Caltech256. The optimizer used during the online training uses a small learning rate since it performs training with only the newly selected samples at each iteration. We report the results in Table 5.4.

The reported results by MET show lower test accuracy than the results reported by the authors. The difference is most significant for the random method, which reports 10% lower test accuracy than the original paper. We speculate that this discrepancy is a result of the differences in the adversary dataset D_a. Even with these discrepancies, our results show the same trend that the original paper shows. The adaptive method using the reinforcement learning approach creates a substitute model with better test accuracy and test agreement than the simple random method.

Method	Victim Test Acc.		Substitute Test Acc.		Test Agreement	
	MET	Original	MET	Original	MET	Original
Random	77.8%	78.8%	61.2%	±71%	63.0%	Unknown
Cert	77.8%	78.8%	68.1%	±73%	68.1%	Unknown
Div	77.8%	78.8%	67.3%	±73%	67.8%	Unknown
Loss	77.8%	78.8%	65.9%	±73%	66.3%	Unknown

Table 5.4: Table with results for our implementation of the KnockOff-Nets attack compared to the results reported in the original paper. Unknown represents values which the authors did not report in the original paper and for which we did not run the experiments. ± is used for values, which are not reported as a number but only displayed as a symbol in a chart in the original papers without value.

5.6 Discussion of the Results

Apart from the KnockOff-Nets attack, all other attacks report very similar results to the original papers. KnockOff-Nets results are further apart, but even the random method is much lower. The original paper reports improvements of only 2% compared to the adaptive technique while MET shows a much bigger improvement. Overall the results achieved by MET follow a similar pattern to the results in the original papers, in all implemented attacks. The various techniques create substitute models with better test accuracy and test agreement than a simple random method. As such, we conclude that our implementations of the attacks are correct.

Chapter 6

Comparison of Implemented Attacks

None of the publications or papers on model extraction attacks presents a comprehensive comparison of different attack techniques in the same experimental settings. In this chapter, we hope to rectify this by showing the results of multiple experiments for all implemented attacks in MET tool. We set multiple goals for the experiments that allow us to compare the performance of individual attacks under variable settings. The experiments aimed to examine the influence of different factors on the attacks, such as the victim model output, the adversary dataset, or the target domain of the victim model. To this end, the experiments were divided into multiple parts, where each experiment tests the impact of a single factor:

- Comparison of attacks on the most popular datasets used in various papers.
- Influence of the victim model's output on the performance of the attack.
- Influence of the adversary dataset diversity on the attacks.

In each setting, we aimed to test all possible variations of attacks implemented in MET to make the fairest comparison and to locate possible areas for improvement. As already mentioned, the primary focus is on image classification problems since they are the most prevalent problem task in model extraction papers [2]. However, we also performed experiments in a more cybersecurity-related domain with the Ember2018[48] malware dataset to test the applicability of the attacks in other domains. As mentioned in 3.3 for each attack, we present both test accuracy and test agreement metrics. Additionally, we present the running time of each attack. This metric is often overlooked, but it is crucial since there is a possible scenario

where the adversary needs to copy the victim model as fast as possible before it is updated or changed.

We also considered additional important experiments testing other factors, which we were unable to conduct due to time constraints but plan to do in the future. We list them for completeness here:

- Scaling of the attacks in terms of adversary dataset size.
- Influence of the subset model architecture on the attacks.
- Performance and scaling of the attacks on harder image classification datasets.
- Performance of the attacks on non-image classification tasks.
- Influence of victim model's training regime on the attacks.

6.1 Adversary Capabilities

Following the threat model described in Section 3.1 and the attack requirements presented in Section 3.2, we consider attackers with the following capabilities: the adversary has only black-box access to the victim model f_v, and they do not know the exact training dataset used to train the victim model. The only information they have access to is the predictions returned by f_v. However, we assume that the adversary has knowledge about the target domain and can collect related data samples for the creation of the adversary dataset. For all different types of experiments, we will provide additional details for the adversary dataset D_a.

6.2 Common Experiment Setup

If not stated otherwise, all experiments used the following setting: The victim model is trained for 200 epochs with the SGD optimizer, the learning rate set to 0.1, and with Nesterov momentum set to 0.9. The learning rate decayed by 0.1 every 60 epochs. As for the substitute model, Adam optimizer was used with default parameters for 100 epochs. For both models, the batch size was set to 150 samples. The seeds for all random generators used in the tool and the generation of dataset subsets were $200,916$ and $201,096$. We understand that it would be better to run

the experiments with more seeds, but time constraints did not allow us to perform experiments with more than two seeds, as the number of different methods needed to be tested is quite substantial.

For both the victim model and substitute model, we use SimpleNet [53]. The decision for using SimpleNet is based on its good performance, easy-to-implement architecture with a relatively small number of parameters, and most importantly, because it works well both on high and smaller resolution images. The code used for all experiments is located in the `experiments` folder in the `attack_tests.py` file.

The recommended settings by the authors for all attacks were used as the default parameters in MET except for the k-center strategy in ActiveThief. For the k-center attack, we used five centers per iteration instead of one because otherwise, the strategy was extremely slow. For most of the attacks, the experiments were run with a query budget of $20,000$ samples. This number was chosen both because of each attack's running time and because we consider an adversary which aims to use as few queries as possible to copy the victim model while minimizing the cost of the attack.

The only exception to the query budget was when performing the BlackBox ripper attack. It is difficult to easily report the actual budget of the attack since the evolutionary algorithm performs a variable number of queries for each generated sample. In addition, for the current implementation of the BlackBox, ripper in MET the budget is calculated from the number of epochs, the batch size, and the number of batches per iteration, which the user has to specify. This restricts the attack budget in our settings (i.e., 100 epochs, 150 batch size) to a budget size of either $15,000$ or $30,000$. We chose the former since it leaves room for additional queries made by the evolutionary algorithm.

Additionally, for each experiment, we also performed a CopyCat attack using the whole adversary dataset D_a to show the best possible result the adversary can achieve in the given scenario. Similarly, for each experiment, we also ran the BlackBox ripper with a budget set to the size of the adversary dataset to see how good is the performance when using only generated Non-Problem Domain Synthetic samples compared to Non-Problem Domain Natural samples. These attacks are represented as *BlackBox ripper: Random All* and *BlackBox ripper: Optimized All* respectively, in the results.

6.3 Baseline Experiments on the Most Popular Datasets

For this experiment, we chose the following target datasets: Cifar10 [33], FashionMNIST [50], and GTSRB [32]. Except for FashionMNIST, each of these datasets is repeatedly used in the model extraction papers to test the performance of the attacks. We used FahsionMNIST because it is often used as a direct replacement for MNIST [31] dataset, another very popular dataset in model extraction attack papers. However, we did not want to use the simple version of MNIST since it is too easy. Both classic ML algorithms and convolutional networks can easily achieve excellent results on MNIST, while this is not the case for FashionMNIST. Details for each of these datasets are presented in Table 6.1. The results of this experiment are used as the baseline for the comparisons with other experiments.

Dataset	Classes	Number of training samples	Number of test samples
GTSRB	43	39,209	12,630
Cifar10	10	50,000	10,000
FashionMNIST	10	60,000	10,000

Table 6.1: Details of popular datasets in model extraction attacks.

6.3.1 Experiment Setup

For the adversary dataset D_a, we used a subset of ImageNet dataset similar to the one used in [10] of 120 thousand samples, which is approximately one-tenth of the whole ImageNet dataset. We took 120 samples from each of the $1,000$ classes from the train split to create the dataset. In the case of Cifar10, since ImageNet contains classes that correspond to the Cifar10 classes, we selected 31 samples from 10 of these classes (plane, sports_car, hummingbird, tiger_cat, water_buffalo, appenzeller, bullfrog, sorrel, speedboat, trailer_truck). Both the adversary and training datasets are resized to resolution 32×32 and normalized into range $[-1, 1]$. For the BlackBox ripper attack, we chose SNGAN from Torch-mimicry as the generator, which was trained on ImageNet and has an output resolution of 32×32. Except for Cifar10, we do not perform any data augmentation on the dataset. For Cifar10, we performed a random crop followed by a random horizontal flip on the training set since the test accuracy of the victim model is better with the augmentation. Finally, all victim models in this set of experiments return the full probability vector as output.

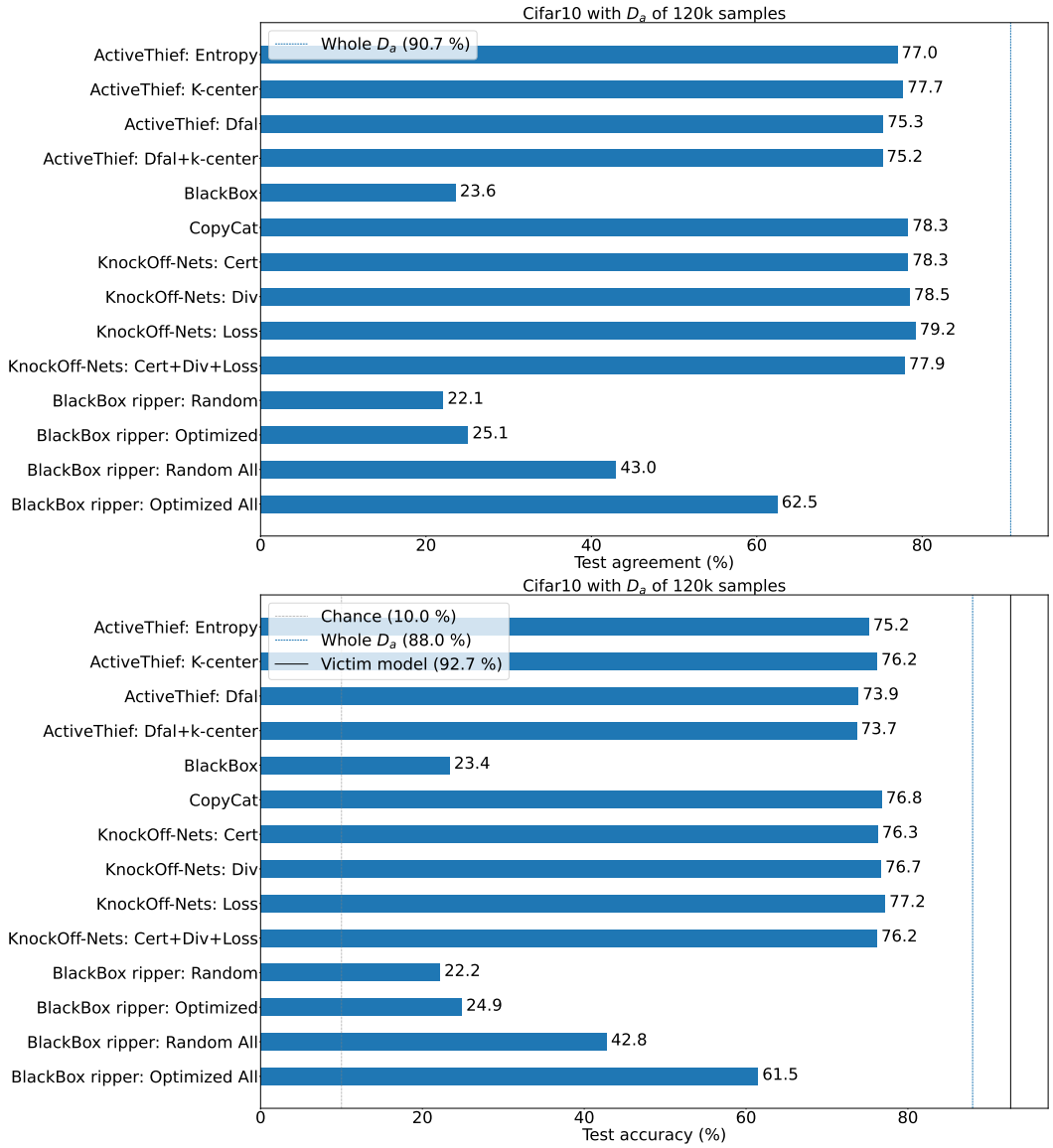


Figure 6.1: Final substitute model's test accuracy and test agreement for baseline experiment on Cifar10. K stands for 1000. BlackBox ripper Random All and Optimized All represents attacks with budget set to the size of $120k$ samples.

6.3.2 Baseline Experiments Results

The results of the experiments are presented in Figures 6.1, 6.2 and 6.3. Additionally, for each dataset, we show the running time of the individual attacks in Table 6.2.

We can see from the figures that for all datasets that ActiveThief, CopyCat, and KnockOff-Nets attacks show the best results overall and the results are quite similar for all variations of these three attacks. Even in the Cifar10 dataset, which

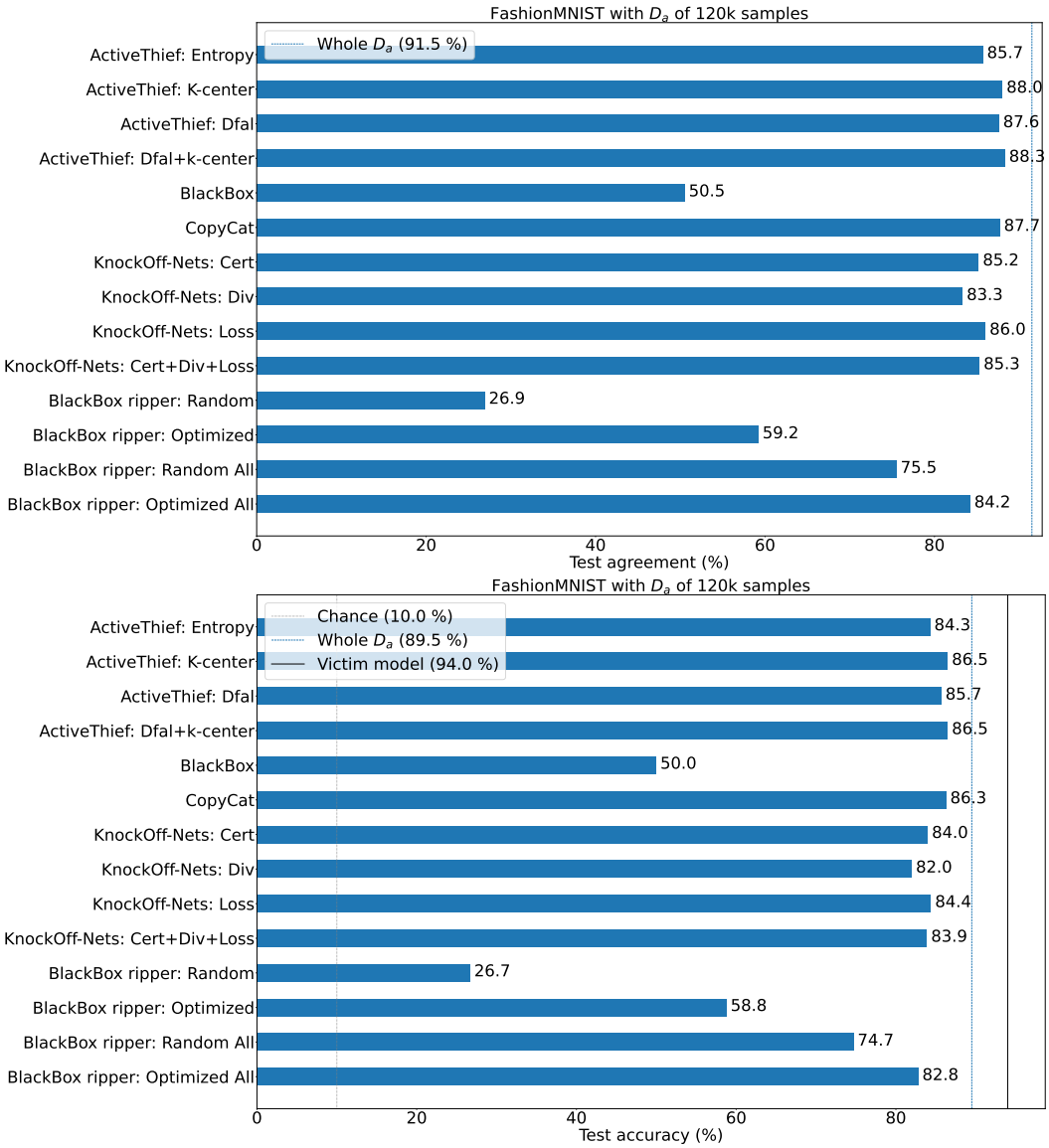


Figure 6.2: Final substitute model's test accuracy and test agreement for baseline experiment on FashionMNIST.

is the hardest dataset of the three, the attacks can create a substitute model with performance close to the substitute created with the whole dataset.

For all datasets, the BlackBox attack performs worse than the rest of the attacks, except for BlackBox ripper attacks with a limited query budget. In prior work [24, 29, 21], the BlackBox attack was always used with seed samples from the same distribution as the training samples of the victim model. As far as we know, the attack was not tested in a more realistic scenario. In our case, we are testing the attacks in an unfavorable setting with only NPDN samples. Regarding GTSRB, ImageNet does not even contain equivalent classes. Nevertheless, the performance

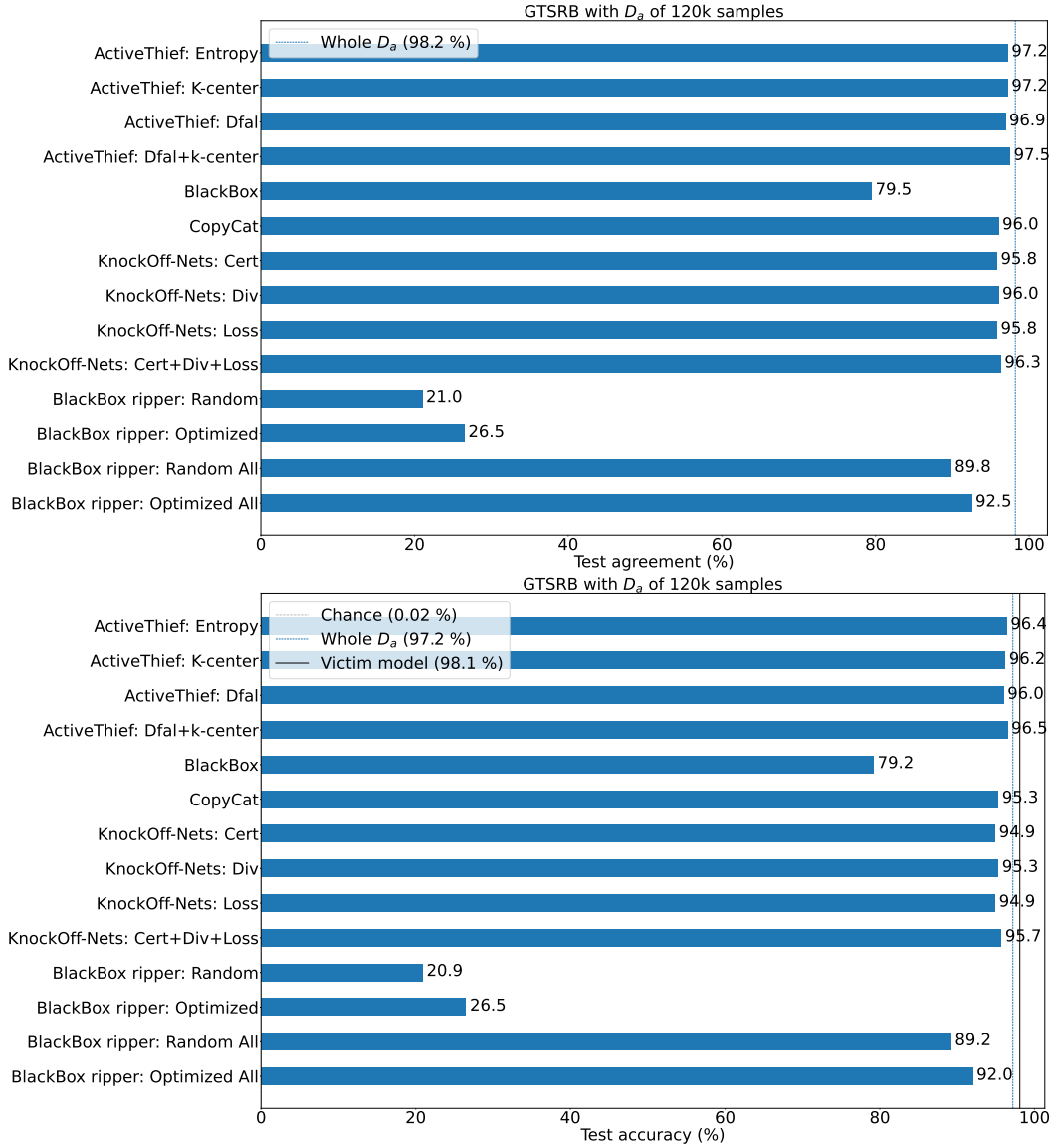


Figure 6.3: Final substitute model's test accuracy and test agreement for baseline experiment on GTSRB.

of the substitute model created by BlackBox attack is much closer to other attacks in these two datasets than in the case of Cifar10, despite the fact that for Cifar10, we used samples from the classes that are closest to the target domain. This clearly shows that the attack performance depends heavily on the test settings and the datasets used.

The BlackBox ripper attack shows similar results to the BlackBox attack when used with a limited budget of 20,000 samples. However, if the budget is increased to the size of the ImageNet adversary dataset, the substitute model performance increases significantly both for the Random method and the Optimized method.

Attack	Method	Time		
		Cifar10	FashionMNIST	GTSRB
ActiveThief	Entropy	1:48:47	3:07:52	1:54:35
	K-center	2:48:53	3:48:48	2:58:19
	DFAL	3:55:07	5:43:47	4:59:45
	DFAL + k-center	5:44:39	5:29:35	5:57:24
BlackBox		0:21:39	0:21:40	0:22:43
CopyCat		0:15:27	0:18:20	0:18:52
	All	1:24:37	1:40:59	1:42:49
KnockOff-Nets	Cert	2:31:13	2:36:50	2:55:49
	Div	2:31:02	2:53:45	3:04:22
	Loss	2:30:30	2:50:38	3:01:18
	Cert + Div + Loss	2:30:25	2:50:41	3:06:10
BlackBox ripper	Random	0:00:37	0:00:37	0:00:38
	Optimized	0:15:45	0:32:28	0:53:16
	Random All	0:03:50	0:03:50	0:03:52
	Optimized All	2:05:21	5:14:23	7:03:36

Table 6.2: Runtime for the model extraction attacks for baseline experiments. The runtime is represented as Hours:Minutes:Seconds.

Even with a 6× larger query budget, the performance of the BlackBox ripper attack is still worse than that of CopyCat, KnockOff-Nets, and ActiveThief attacks. From the results, we can also clearly see the benefit of using the evolutionary algorithm compared to randomly generated samples in BlackBox ripper attack, creating a substitute model, with 20% better performance in test accuracy and test agreement in the case of Cifar10.

As for the KnockOff-Nets and ActiveThief attacks, the performance of these two attacks is similar in all datasets. There is, however, one trend we can observe. The KnockOff-Nets attack can achieve the best performance in Cifar10, while the ActiveThief can achieve better performance for GTSRB and FashionMNIST datasets. This result is a consequence of the adversary dataset used. The samples from ImageNet are much closer to the samples contained in Cifar10 than GTSRB and FashionMNIST. As a result, the confidence of the predictions from the victim model is lower when using GTSRB and Fashion-MNIST, resulting in a worse feedback signal for the individual policies in KnockOff-Nets.

Regarding the runtime results of the attacks, CopyCat is the fastest overall attack, followed by the BlackBox attack. On the other end, the slowest is by far ActiveThief with the *DFAL* method. To evaluate its performance with a resolution higher than 32×32, we also performed a test with a resolution of 128×128. From our preliminary

results with a higher resolution, the *DFAL* method requires 3 hours to select samples for the query set. We can expect that its performance would be even worse than the other methods if we used a higher resolution dataset. From the results, we can also see that the performance of the KnockOff-Nets attack is the same for all methods, unlike the ActiveThief, where the running time of each attack strategy varies greatly. Lastly, in the BlackBox ripper attack, the runtime of the attack varies greatly between datasets. The reason for this is similar to that of the KnockOff-Nets attack performance. Since the generator is trained on ImageNet, whose samples are more similar to Cifar10, fewer queries are necessary for each generated sample than for the other datasets. In the GTSRB dataset, we can see the extreme case, where the optimized method has a better substitute model by 0.5% in both accuracy and agreement, but the running time difference is 7 hours.

Since the performance of the attack follows a similar pattern for all popular datasets, we will be using only one of the datasets for the following experiments unless stated otherwise.

6.4 Influence of the Victim Model's Output on Attack Performance

The next set of experiments was to test how the victim model's output influences the performance of the attacks. We test two scenarios. Firstly, the output precision of the victims is lowered by rounding the output to one decimal place. Secondly, the performance of the attack is tested in the more challenging scenario where the returned output is one-hot encoded.

The settings for these experiments are the same as for the baseline ones, and we used the Cifar10 dataset as a target dataset. We do not report the results for all attacks in this experiment. Specifically, we do not show results for KnockOff-Nets with diversity and certainty rewards. The reason is that both of these rewards can not be used with the one-hot output. The output does not contain enough information for these rewards to function correctly, i.e., the certainty reward is constantly 1, and in the case of diversity, the reward slowly converges to 1. We also omit the results for the ActiveThief attack with the *DFAL* method since it is slow and the performance is similar to the other ActiveThief methods.

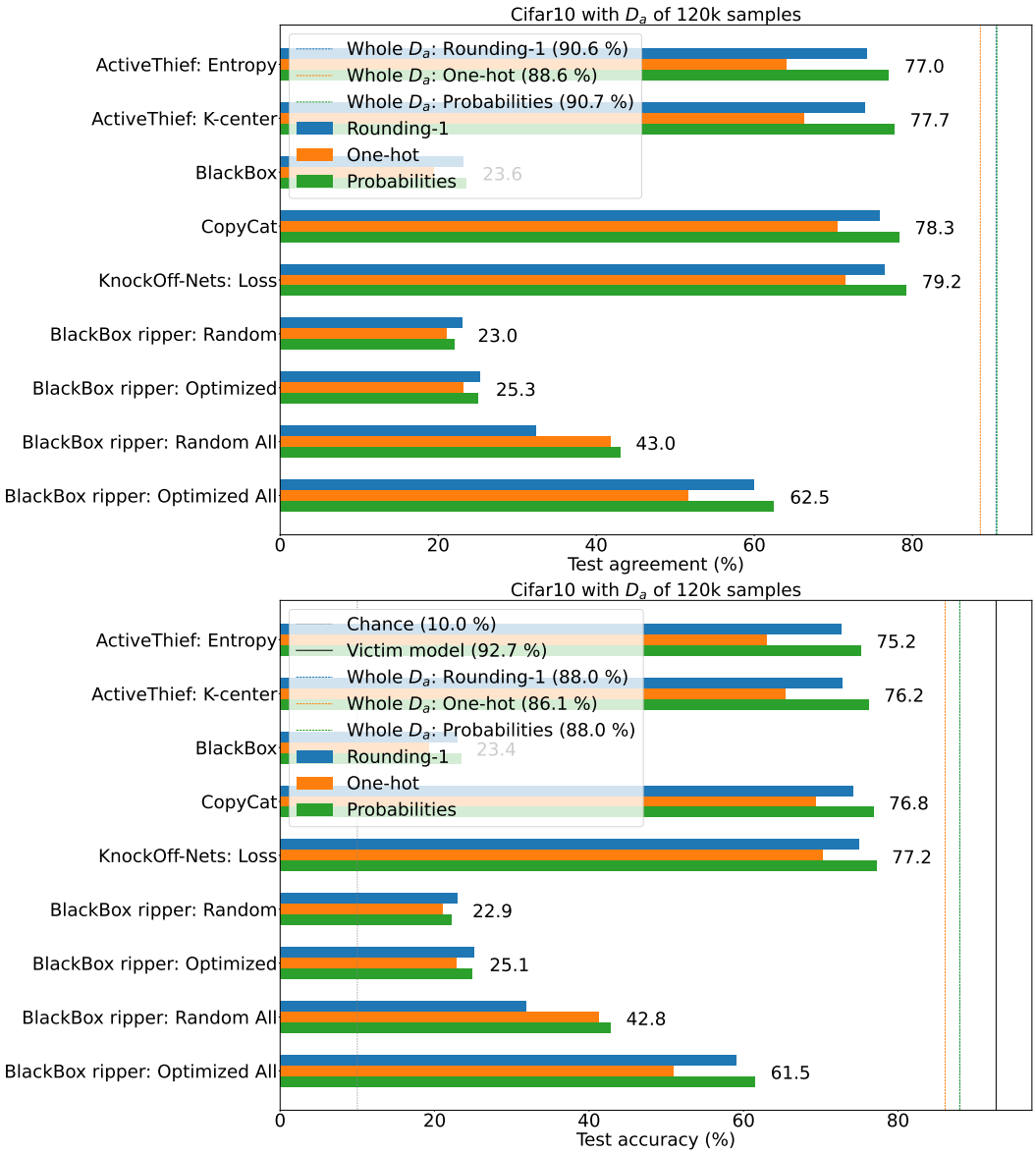


Figure 6.4: Final substitute model's test accuracy and test agreement for victim model's outptut experiment on Cifar10.

6.4.1 Results

The experiment results are presented in Figure 6.4. Immediately, we can see that the performance of the attacks drops significantly with the less detailed one-hot output. The drop is most noticeable with the ActiveThief attack, which drops by more than 10% across all of its different methods in both accuracy and test agreement. We hypothesize that the performance drop is most substantial for ActiveThief because it is the only attack that uses the substitute model to select samples for querying. However, since the performance of the substitute is worse with the more coarse output

of the victim model, it influences the selection of the samples for querying. The results also show one anomaly in the case of BlackBox ripper with a random version where the substitute model created with one hot output has better performance than the rounding one. We, unfortunately, do not have an explanation for this anomaly.

On the other hand, rounding shows a much less profound effect on attack performance than one-hot output. Even while limiting the output to one decimal place, the attacks' performance is very similar to the substitute models from the baseline experiment with access to the full output. The drop is again the largest for the ActiveThief attack, however, this time by maximum $\pm 4\%$. Surprisingly, in the case of using the whole adversary dataset D_a for the attack, the performance of the substitute model is the same as for the one created with probability output.

6.5 Influence of the Adversary Dataset on Attack Performance

Another important thing in the model extraction attacks is the influence of the adversary dataset D_a. There are multiple factors, which can be studied in this case:

- The representation of Problem Domain (PD) and Non-Problem Domain (NPD) samples in the D_a
- The size of the D_a
- The diversity of the D_a, i.e., number of classes for classification problems

Initially, we aimed to test the influence of both the size and diversity of D_a. Unfortunately, the experiments with a larger sized D_a proved to be too slow, especially in the ActiveThief attack. Because of time constraints, we tested only the influence of the adversary dataset with fewer samples and smaller diversity, and we leave the rest of the tests as future work.

6.5.1 Experiment Setup

The experiment uses the same setup as the baseline experiments. For the adversary dataset D_a, we use the Cifar100 [33] dataset, which contains the same samples as the

Cifar10 dataset but with more granular labels. Compared to the ImageNet subset used in baseline experiments, it has 10× fewer classes and half as many samples. One of the reasons for choosing the Cifar100 was the availability of pretrained GAN generators needed for the BlackBox ripper attack since the training of GANs from scratch is a time-intensive task.

6.5.2 Results

The experiment results can be seen in Figure 6.5. Overall, the performance of the substitute models created by the attack while using the Cifar100 dataset as the adversary dataset, shows slightly worse results than those created with the ImageNet subset. However, the BlackBox ripper optimized method and KnockOff-Nets using all reward methods show opposite results.

BlackBox ripper can create a substitute model with 6% better accuracy and 7% better test agreement. From these results and the ones in Section 6.4.1, it seems that there is a factor of randomness in the generation of synthetic samples in the BlackBox ripper attack influencing the performance of the substitute model. This needs to be tested more thoroughly in the future to measure the stability of the results of the BlackBox ripper attack with different generators and adversary datasets.

Regarding the KnockOff-Nets attack, the combination of reward maximizing the exploitation (i.e., certainty reward) and rewards maximizing exploration (i.e., diversity and loss reward) can create a better substitute model, although each of the methods individually fails to do so. While the results for Cifar100 are lower by a few percent points, overall, the results are very close for all methods.

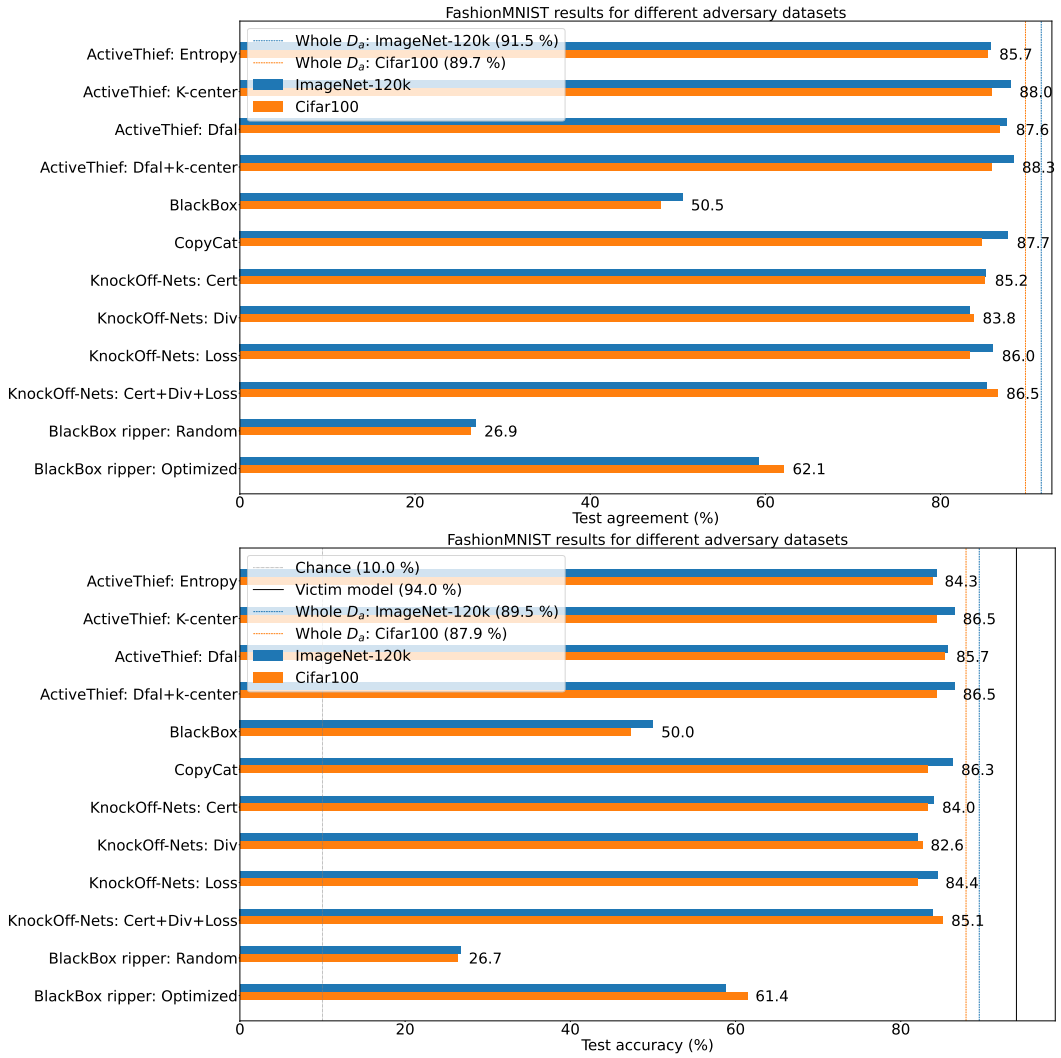


Figure 6.5: Final substitute model's test accuracy and test agreement for experiment with different adversary datasts.

Chapter 7

Attack Improvements

This chapter is dedicated to the improvements proposed for some of the model extraction attacks that were implemented and tested in MET. From the results of the experiments, we noticed some shortcomings and places where the performance of the attacks could be improved. By performance, we mean both the test accuracy and test agreement of the final substitute model and the runtime of the attack itself. We separate our proposed improvements according to the model extraction attacks. Additionally, at the end of this chapter, we also talk about a possible improvement and possible future research area, which is outside of the implemented model extraction attacks.

7.1 ActiveThief Attack Improvements

The first attack we focused on was the ActiveThief attack, where there were immediately multiple places that could be improved. The first problem identified was the runtime of the attack, specifically in the k-center strategy.

The authors' original implementation of the k-center strategy recalculates the distances to each center in every iteration for all unlabeled samples from D_a. MET provides an optimized version of the k-center strategy for both large and smaller adversary datasets. The default k-center strategy provided by the MET provides a version optimized for large adversary datasets. Compared to the original implementation, it calculates the distances from each center only once for every unlabeled sample. It keeps the minimal distance for each unlabeled sample, from which it then

selects the maximum. This improves the performance. However, as mentioned in Section 6.2 even with this optimization, the method is still slow if used with one center per iteration. This is because the distances are calculated iteratively over batches of unlabeled samples for the initial centers and the new centers. On the one hand, this allows the method to work even with massive adversary datasets and a target dataset with many classes. On the other hand, as a consequence, the performance suffers.

For this reason, the MET also provides a so-called *fast k-center* method for smaller datasets. It drops the calculations over batches and instead performs vectorized operations, which improves the performance significantly. However, it is limited to smaller adversary datasets since it calculates distances for all unlabeled samples concurrently. Both versions can leverage the GPU if available, unlike the original version, and additionally support a selection of multiple centers per iteration. The runtime of the different k-center implementations for the selection of 1800 samples on Cifar10 is shown in Table 7.1. For all versions, we select only one new center at a time. The original version of k-center was also modified to leverage the GPU to speed up the experiments.

From the results, we can see that both versions of k-center available in MET outperform the k-center implementation available in the original implementation of ActiveThief. The vectorized version of the k-center shows tremendous performance improvement compared to the other versions. However, one must remember that the *fast k-center* initially calculates the full dense distance matrix from which it takes only the minimal distance for each unlabeled sample. For smaller adversary datasets, this is not a problem. However, D_a with a more significant number of samples starts to become a limitation since the distance matrix might not fit into Random Access Memory (RAM) or Video Random Access Memory (VRAM) on the GPU.

Version	K-center original	K-center default	K-center fast
Time	0:29:31	0:27:47	0:00:1

Table 7.1: Runtime results for single sample selection of k-center method versions on Cifar10. The runtime is reported in Hours:Minutes:Seconds format. The results are reported as the average time after 3 completed iterations.

The next improvement was the DFAL strategy and the DFAL+k-center strategy. As it can be seen in the experiment results in Table 6.2, it takes the DFAL strategy a long time to score the unlabeled samples. Additionally, the method does not scale well for higher dimensional samples, and the current implementation provided in MET is limited only to images. For unknown reasons, the authors of ActiveThief in their original paper did not test the combination of entropy+k-center in their work. We thus additionally added entropy+k-center strategy to the implementation of ActiveThief attack in MET. This combination is based on the same assumption as the DFAL+k-center strategy. While the entropy strategy can select the samples

closest to the decision boundaries, it does not guarantee that the samples are diverse enough. For this reason, the k-center method is used. We tested this method on the Cifar10 and FashionMNIST datasets and compared it to other ActiveThief methods. The results are presented in Table 7.2. From the results, we can see that the method shows similar performance to the other methods, even beating DFAL+k-center in both datasets. Additionally, the running time of entropy+k-center is much lower and can be easily used with other data types than images.

Method	**Cifar10**		**FashionMnist**	
	Acc	Agr	Acc	Agr
Entropy	75.25%	76.95%	84.35%	85.65%
K-center	**76.2**%	**77.65**%	86.55%	88%
DFAL	73.9%	75.35%	85.7%	87.6%
DFAL+K-center	73.7%	75.25%	86.55%	**88.3**%
Entropy+K-center	75.1%	76.8%	**86.7**%	88.1%

Table 7.2: Comparison of Entropy + K-center method to rest of the ActiveThief methods. Acc stands for test accuracy, and Agr represents test agreement.

Additionally to the entropy+k-center, we also tried using a method from [54] based on active transfer learning called Active Transfer Learning for Adaptive Sampling (ATLAS), which combines both uncertainty and diversity sampling into one method. The method tries to leverage Tranfer Learning by retraining the model during the AL. It does this by giving the samples from the validation set new binary labels (correct, incorrect) depending on whether they were correctly labeled by the model, creating a new training set. This new training set is used to retrain the model to predict whether the sample is labeled "correctly" or "incorrectly".

This new model created by the TL can tell us whether the current model will correctly predict the label for a sample. It is used to predict labels for the unlabeled samples. The samples which are predicted with the highest confidence of being "incorrect" are added to the new query set and relabeled as "correct" and added to the training set. This relabeling is based on the assumption that the models are typically the most accurate on their training samples. Since the selected samples will be labeled and used for training the model in the next AL iteration, we can expect that the model will give correct predictions for these samples. This whole process repeats until a required number of samples is selected. Unfortunately, we could not make the method work reliably in the model extraction domain and currently have a work in progress version as part of the MET.

7.2 BlackBox Ripper Attack Improvements

As mentioned in Section 3.4.5, the authors did not focus on query efficiency during the creation of their attack. As such, the performance suffers in settings where the adversary has a limited budget. This could be seen in the results of the experiments in Section 6. We propose two simple improvements, which improve the performance with lower budget settings.

The first improvement is related to the use of the generated samples. The original implementation of the attack does not store the samples created by the generator. Each epoch repeatedly creates batches of synthetic samples, which it immediately discards, which means that the substitute model sees each sample only once. This seems as a very inefficient way of using the created samples since each sample is used only once. The original version of the algorithm is even more inefficient if one considers that each sample creation requires multiple queries to the victim model. Thus, we propose a simple improvement for the attack. Instead of using new samples for each batch in every epoch, we propose creating the samples once in the beginning and saving them, thus creating an adversary dataset D_a consisting of NPDN samples. This dataset is then used for the training of the substitute model. We call this version *BlackBox ripper Saved*. Figure 7.1 shows the flowchart for the updated version of the BlackBox ripper attack.

The second improvement is related to the selection of the best samples for each batch. Originally, during the attack, when the sample with the lowest loss fulfills the optimization criterion of the evolutionary algorithm, the optimization is stopped, the best sample is added to the batch, and the rest of the samples are thrown away. However, while the optimization criterion is not fulfilled, the latent vector, which generates the sample with the lowest loss in the current iteration, is used to generate the latent vectors in the next iteration by adding to it a small random noise. This means that there is a chance that additional samples in the last iteration of the evolutionary algorithm could fulfill the optimization criterion since all samples in a single iteration of the algorithm are generated from similar latent vectors. To see if our assumption is correct, we created a version of the attack called *BlackBox ripper Additional*. This version of the attack tries to check if the rest of the samples in the last iteration of the evolutionary algorithm fulfill the optimization criterion and adds them to the batch. This method's goal is to increase the attack efficiency since it minimizes the number of unused samples, which were already queried to the victim model.

The proposed improved versions are compared with the original version of BlackBox Ripper on Cifar10 and FashionMNIST datasets. We use the same settings as the ones used in the baseline experiments. The testing of the new versions is performed

with a budget set to 15,000 queries. The results are shown in Table 7.3. The *BlackBox ripper Saved* version achieves more than 40% improvement in accuracy and agreement than the original version in Cifar10 dataset and over 50% improvement in FashionMNIST with the simple Random method. The same trend can be seen with the Optimized method too. In both methods the Saved version achieves a better result with a budget more than 6× smaller than the original version. As for the *BlackBox ripper Additional* by itself, the additional samples show marginal improvements. However, the combination of both improvements in *BlackBox ripper Additional + Saved* brings the performance of the attack to a level similar to that of CopyCat, KnockOff-Nets, and ActiveThief attacks.

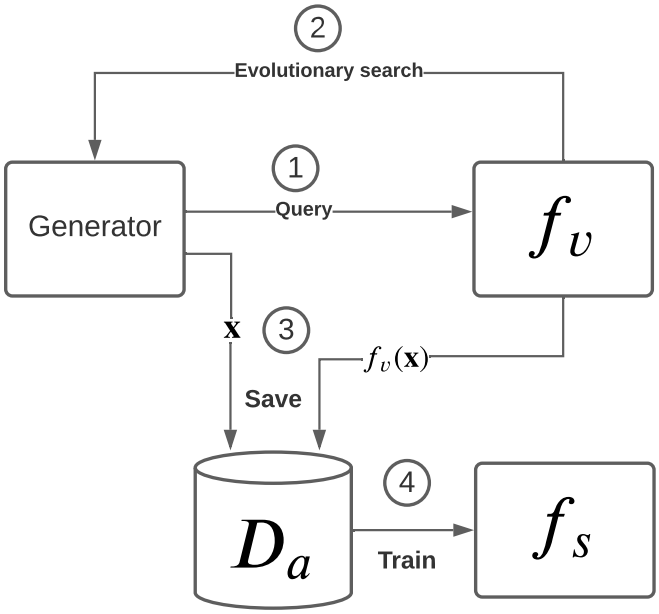


Figure 7.1: Flowchart of BlackBox ripper saved attack

Method	Cifar10		FashionMNIST	
	Acc	Agr	Acc	Agr
Random	22.2%	22.15%	26.7%	26.85%
Random All	42.75%	43.05%	74.65%	75.45%
Random Saved	60.25%	61.25%	81.3%	82.35%
Optimized	24.85%	25.1%	58.75%	59.2%
Optimized All	61.45%	62.5%	82.85%	84.25%
Optimized Additional	28.85%	28.9%	62.2%	62.85%
Optimized Saved	67.95%	69.1%	85.25%	87.15%
Optimized Additional + Saved	**75.15%**	**76.75%**	**86.4%**	**88.2%**

Table 7.3: Comparison of the BlackBox ripper proposed improvements with the original version of the attack. Acc stands for test accuracy, and Agr represents test agreement.

7.3 Semi-Supervised Learning in Model Extraction Domain

In all results from the experiments in Chapter 6, we can observe a similar tendency. The increase in test accuracy of the substitute model comes hand in hand with increasing test agreement. This result motivated our interest in Semi-Supervised Learning (SSL) as it might be possible after finishing the attack to use the unlabeled samples from the adversary dataset D_a to improve the test accuracy of the learned substitute model and at the same time improving its test agreement. All of this without needing to send additional queries to the victim model. We note that this assumption might be valid only in cases where the victim model performs well as was in our experiments. The preliminary results of experiments, where the victim model has worse performance, show that the test accuracy and the test agreement have antagonistic behavior. The increase in test agreement results in lower test accuracy and vice versa. For these reasons, the Semi-Supervised Learning (SSL) approach might be more interesting for adversaries motivated by task accuracy extraction than fidelity extraction.

Currently, there is, as far as we know, only one instance of using SSL in model extraction attacks [3]. In their case, they tested SSL methods with only PD data from the same distribution as the training data of the victim model and with methods designed for their tested datasets. There currently, however, does not exist any work showing comprehensive results to see how SSL performs in more realistic scenarios with both PD samples and NPD samples from a different distribution. The use of SSL however, might be problematic, especially with NPD samples since there are some essential prerequisites that the sample distribution must-have for SSL, as mentioned in Section 2.1.4. These assumptions might prove to be a limiting factor for using SSL in the model extraction domain. Since as seen in the results of KnockOff-Nets in [5] samples with the same labels vary greatly when using NPD data. We planned to implement some SSL methods in MET, but there was not enough time to test their perfromance. However, it is an interesting topic for future work and we plan to add SSL to MET in the future.

Chapter 8

Conclusion

Despite the increasing momentum of model extraction attacks in the recent literature, there is currently no easy way to test and compare the attacks proposed in different papers under variable settings. For this purpose, we created the Model Extraction Tool (MET), which allows easy testing and comparison of model extraction attacks. MET currently offers the largest number of implemented model extraction attacks compared to other similar tools. In total, MET offers five different attacks with over 20 total variations. We plan to release the tool under the MIT license so that researchers can use it and modify it freely. Additionally, all experiments in this work are also provided together with the tool as scripts for easy reproducibility of the results.

Based on the literature review and a set of requirements, we chose 5 different attacks for implementation. The implemented attacks were CopyCat, BlackBox, ActiveThief, KnockOff-Nets, and BlackBox ripper. All attacks were compared to the original implementations and were tested for correctness in settings as close to the original papers as possible.

Using MET, we performed the most extensive comparison of the different model extraction attacks under the same settings to date. The experiments were performed in a challenging scenario where the adversary has only black-box access to the victim model and is using Non-Problem Domain (NPD) samples for the attack. From the results, the attacks that were able to create the best substitute model were CopyCat, ActiveThief, and KnockOff-Nets. The results also showed that the BlackBox attack performance suffers in scenarios where the adversary does not have access to PD samples from a similar distribution as the samples used to train the victim model. The most important conclusion we can, however, make from the results of the

experiments is that all sophisticated model extraction attacks, while showing better results, provide only marginal benefits to the simple CopyCat attack in our scenario with a weak adversary.

The running time performance of the attacks, which is a neglected piece of information in most of the model extraction papers, was also measured. Contrary to the attack performance results on the creation of substitute models, the running time results showed much more significant variance. Overall, the two slowest attacks were the ActiveThief attack with the DFAL method and the BlacBox ripper attack with the evolutionary algorithm. Our results showed that the BlackBox ripper attack, running time heavily depends on the similarity between the training dataset used for training the GAN generator and the target domain of the victim model. If we consider the scenario, where runtime is as an important factor for the adversary as the performance of the final substitute model, the best overall model extraction attack is the KnockOff-Nets attack. KnockOff-Nets was able to keep consistently good performance in both categories over all of our experiments.

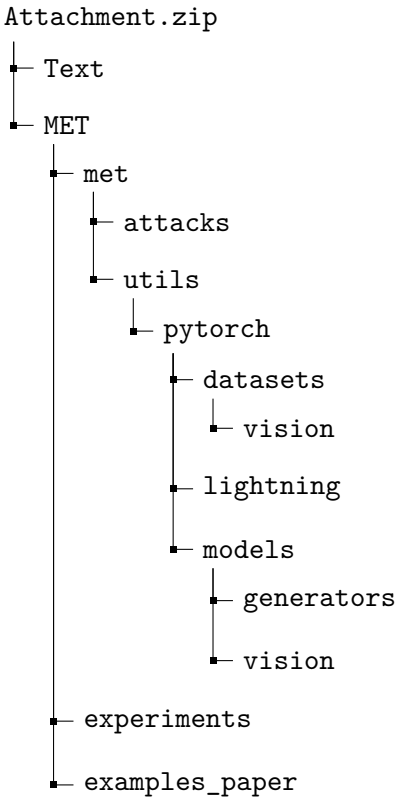
Furthermore, we proposed multiple improvements to the ActiveThief and BlackBox ripper attacks. In the case of the ActiveThief attack, we focused on improving the running time performance of the k-center attack. Our implementation, k-center-fast, is $1,771\times$ times faster than the original k-center, while maintaining the attack performance in terms of accuracy and agreement. We also tested the performance of the entropy+k-center method, which was omitted by the ActiveThief's authors. The method showed similar performance to the other methods available in ActiveThief. However, compared to the DFAL+k-center method, which also combines the uncertainty and diversity sampling, the entropy+k-center offers a much better runtime performance. As for the BlackBox ripper attack, we proposed two improvements to the attack, which increase the query efficiency of the attack significantly. Our improved version can achieve better results with a much smaller budget compared to the original version of the attack.

Lastly, we discussed the potential future research into SSL, which is currently an untapped approach in the model extraction domain and offers potentially exciting improvements to all existing model extraction attacks. During our experiments, we also noticed a possible defense against model extraction attacks. If data augmentation was used during the training of the victim model, the performance of all attacks dropped, while the victim model's performance differed only slightly from the model trained without data augmentation. Unfortunately, due to time constraints, it was not possible to study this phenomenon further to check to study the phenomenon deeper and establish if it is a defense applicable to a variety of settings and datasets. This merits further research since if the model extraction attacks are indeed susceptible to performance loss depending on the training regime of the victim model, it could provide an easy defense against these types of attacks.

We were able to successfully implement the MET tool for testing and trying different model extraction attacks. Using the tool, we performed a comprehensive comparison of the implemented attacks and proposed improvements on some of the them. Furthermore, we also discussed the possible future research in model extraction attacks. Unfortunately, we were not able to do more experiments to get a deeper insight into model extractions attacks. We hope that in the future MET will be useful to researchers interested in the developemnt of future model extraction attacks and defenses as well as to anyone that is interested in testing their own model against state-of-the-art model extraction attacks.

Appendix A

Contents of CD



Appendix B

Bibliography

[1] B. Settles, "Active learning literature survey," Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[2] M. Rigaki and S. Garcia, "A survey of privacy attacks in machine learning," *arXiv preprint arXiv:2007.07646*, 2020.

[3] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1345–1362, 2020.

[4] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Exploring connections between active learning and model extraction," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1309–1326, 2020.

[5] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4954–4963, 2019.

[6] A. Barbalau, A. Cosma, R. T. Ionescu, and M. Popescu, "Black-box ripper: Copying black-box models using generative evolutionary algorithms," *arXiv preprint arXiv:2010.11158*, 2020.

[7] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 601–618, 2016.

[8] M. Nicolae, M. Sinn, T. N. Minh, A. Rawat, M. Wistuba, V. Zantedeschi, I. M. Molloy, and B. Edwards, "Adversarial robustness toolbox v0.2.2," *CoRR*, vol. abs/1807.01069, 2018.

[9] S. S. Hussain, P. Wang, and J. Miller, "Privacyraven." `https://github.com/trailofbits/PrivacyRaven`, 2020.

[10] S. Pal, Y. Gupta, A. Shukla, A. Kanade, S. Shevade, and V. Ganapathy, "Activethief: Model extraction using active learning and unannotated public data," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 865–872, 04 2020.

[11] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[12] F.-F. Li, R. Krishna, D. Xu, and A. Byun, "Cs231n: Convolutional neural networks for visual recognition."

[13] N. J. Nilsson, "Introduction to machine learning: An early draft of a proposed textbook. pages 175-188. http://robotics.stanford.edu/people/nilsson/mlbook.html," 1996.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[15] A. Amini and A. Soleimany, "6.s191 introduction to deep learning," 2020.

[16] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[17] Y.-P. Lin and T.-P. Jung, "Improving EEG-based emotion classification using conditional transfer learning," *Frontiers in human neuroscience*, vol. 11, p. 334, 2017.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[21] H. Yu, K. Yang, T. Zhang, Y.-Y. Tsai, T.-Y. Ho, and Y. Jin, "Cloudleak: Large-scale deep learning models stealing through adversarial examples," in *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[22] X. Yang, Z. Song, I. King, and Z. Xu, "A survey on deep semi-supervised learning," *arXiv preprint arXiv:2103.00550*, 2021.

[23] J. E. Van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Machine Learning*, vol. 109, no. 2, pp. 373–440, 2020.

[24] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pp. 506–519, 2017.

[25] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 36–52, IEEE, 2018.

[26] S. J. Oh, B. Schiele, and M. Fritz, "Towards reverse-engineering black-box neural networks," in *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pp. 121–144, Springer, 2019.

[27] J. R. Correia-Silva, R. F. Berriel, C. Badue, A. F. de Souza, and T. Oliveira-Santos, "Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data," in *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2018.

[28] H. Yu, K. Yang, T. Zhang, Y. Tsai, T. Ho, and Y. Jin, "Cloudleak: Large-scale deep learning models stealing through adversarial examples," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020.

[29] M. Juuti, S. Szyller, S. Marchal, and N. Asokan, "Prada: protecting against dnn model stealing attacks," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 512–527, IEEE, 2019.

[30] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[31] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[32] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, and C. Igel, "Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark," in *International Joint Conference on Neural Networks*, no. 1288, 2013.

[33] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[34] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," *Journal of Machine Learning Research - Proceedings Track*, vol. 15, pp. 215–223, 01 2011.

[35] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[36] O. Sener and S. Savarese, "Active learning for convolutional neural networks: A core-set approach," in *International Conference on Learning Representations*, 2018.

[37] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582, 2016.

[38] A. Quattoni and A. Torralba, "Recognizing indoor scenes," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 413–420, IEEE, 2009.

[39] G. Griffin, A. Holub, and P. Perona, "Caltech-256 object category dataset," *CalTech Report*, 03 2007.

[40] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, and et al., "The open images dataset v4," *International Journal of Computer Vision*, vol. 128, p. 1956–1981, Mar 2020.

[41] K. Vit and M. Rigaki, "Model extraction tool." `https://github.com/stratosphereips/MET`, 2021.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[43] W. Falcon and .al, "Pytorch lightning," *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, vol. 3, 2019.

[44] S. Marcel and Y. Rodriguez, "Torchvision the machine-vision package of torch," in *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, (New York, NY, USA), p. 1485–1488, Association for Computing Machinery, 2010.

[45] J. Rauber, W. Brendel, and M. Bethge, "Foolbox: A python toolbox to benchmark the robustness of machine learning models," in *Reliable Machine Learning in the Wild Workshop, 34th International Conference on Machine Learning*, 2017.

[46] J. Rauber, R. Zimmermann, M. Bethge, and W. Brendel, "Foolbox native: Fast adversarial attacks to benchmark the robustness of machine learning models in pytorch, tensorflow, and jax," *Journal of Open Source Software*, vol. 5, no. 53, p. 2607, 2020.

[47] K. S. Lee and C. Town, "Mimicry: Towards the reproducibility of gan research," in *CVPR Workshop on AI for Content Creation*, 2020.

[48] H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," *ArXiv e-prints*, Apr. 2018.

[49] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 3149–3157, Curran Associates Inc., 2017.

[50] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[51] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278 – 2324, 12 1998.

[52] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, "Spectral normalization for generative adversarial networks," *arXiv preprint arXiv:1802.05957*, 2018.

[53] S. H. Hasanpour, M. Rouhani, M. Fayyaz, and M. Sabokrou, "Lets keep it simple, using simple architectures to outperform deeper and more complex architectures," *arXiv preprint arXiv:1608.06037*, 2016.

[54] R. M. Monarch, *Human-In-the-Loop Machine Learning: Active Learning and Annotation for Human-Centered AI.* Manning Publications Co. LLC, 2021. OCLC: 1240165658.