University of California
Santa Barbara

Improving and Securing Machine Learning Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Bolun Wang

Committee in charge:

Professor Ben Y. Zhao, Co-Chair
Professor Haitao Zheng, Co-Chair
Professor Giovanni Vigna

March 2019

The Dissertation of Bolun Wang is approved.

Professor Giovanni Vigna

Professor Ben Y. Zhao, Committee Co-Chair

Professor Haitao Zheng, Committee Co-Chair

November 2018

Improving and Securing Machine Learning Systems

by

Bolun Wang

To my family, friends, and loved ones

Acknowledgements

First, I would like thank both of my advisors Prof. Ben Y. Zhao and Prof. Heather Zheng for their guidance throughout my PhD. Not only did they give me directions and advices in research, but they themselves have been role models for me in both research and life. It's their devotion into creating a motivated, productive, and collaborative lab, that made all achievements in my PhD possible. For all these, I would give my sincere gratitude to Ben and Heather.

Second, I would also like to thank my committee member, Prof. Giovanni Vigna, for providing valuable feedback and advise during my PhD.

Third, it was my greatest pleasure working with all my collaborators, who made this thesis possible. I would like to thank Gang Wang, Bimal Viswanath, Zengbin Zhang, Divya Sambasivan, Tianyi Wang, Ana Nika, Xinyi Zhang, Yuanshun Yao, Zhujun Xiao, Huiying Li, and Shawn Shan. The success of all the amazing and fun projects would not be possible without their hard work, creativity, and smooth collaboration.

I would also like to thank all members of SANDLab that make my PhD full of joy and fun. They are the reasons that make SANDLab the one and the only, with its unique vibrancy and liveness. I could never imagine working in a lab better than this. Apart from names I mentioned before, I would also like to mention Lin Zhou, Xiaohan Zhao, Yibo Zhu, Qingyun Liu, Zhijing Li, Shiliang Tang, Yanzi Zhu, Jenna Cryan, Yuxin Chen, Emily Wilson, Max Liu, and Olivia Sturman. Thank you for all the joy and fun you bring to me.

Finally, I would like to thank my family, my friends, and my loved ones for their unconditional support. It helps me overcome obstacles through this journey. I wouldn't have made this far without their help.

Curriculum Vitæ
Bolun Wang

Education

2018	Ph.D. in Computer Science, University of California, Santa Barbara.
2013	Bachelor of Science in Electronic Engineering, Tsinghua University, China

Conference Publications

- **Bolun Wang**, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng and Ben Y. Zhao. "Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks." In Proceedings of *40th IEEE Symposium on Security and Privacy*. San Francisco, CA, May. 2019. (***S&P***)
- **Bolun Wang**, Yuanshun Yao, Bimal Viswanath, Haitao Zheng and Ben Y. Zhao. "With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning." In Proceedings of *The 27th USENIX Security Symposium*. Baltimore, MD, Aug. 2018. (***USENIX Security***)
- Yuanshun Yao, Zhujun Xiao, **Bolun Wang**, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. "Complexity vs. Performance: Empirical Analysis of Machine Learning as a Service." In Proceedings of *The 17th ACM Internet Measurement Conference*. London, UK, Nov. 2017. (***IMC***)
- **Bolun Wang**, Xinyi Zhang, Gang Wang, Haitao Zheng and Ben Y. Zhao. "Anatomy of a Personalized Livestreaming System." In Proceedings of *The 16th ACM Internet Measurement Conference*. Santa Monica, California, USA, Nov. 2016. (***IMC***)
- Gang Wang, **Bolun Wang**, Tianyi Wang, Ana Nika, Haitao Zheng and Ben Y. Zhao. "Defending against Sybil Devices in Crowdsourced Mapping Services." In Proceedings of *The 14th ACM International Conference on Mobile Systems, Applications, and Services*. Singapore, June 2016. (***MobiSys***)
- Gang Wang, Tianyi Wang, **Bolun Wang**, Divya Sambasivan, Zengbin Zhang, Haitao Zheng, and Ben Y. Zhao. "Crowds on Wall Street: Extracting Value from Collaborative Investing Platforms." In Proceedings of *The 18th ACM Conference on Computer-Supported Cooperative Work and Social Computing*. Vancouver, BC, Canada, March 2015. (***CSCW***)
- Gang Wang, **Bolun Wang**, Tianyi Wang, Ana Nika, Haitao Zheng and Ben Y. Zhao. "Whispers in the Dark: Analysis of an Anonymous Social Network." In Proceedings of *The 14th Internet Measurement Conference*. Vancouver, BC, Canada, Nov. 2014. (***IMC***)

Journals

- Gang Wang, **Bolun Wang**, Tianyi Wang, Ana Nika, Haitao Zheng and Ben Y. Zhao. "Ghost Riders: Sybil Attacks on Crowdsourced Mobile Mapping Services" *IEEE/ACM Transaction on Networking (**TON**)*, 2018
- Tianyi Wang, Gang Wang, **Bolun Wang**, Divya Sambasivan, Zengbin Zhang, Xing Li, Haitao Zheng, Ben Y. Zhao. "Value and Misinformation in Collaborative Investing Platforms" *ACM Transactions on the Web (**TWEB**)*, 2017
- Tianyi Wang, Yang Chen, Yi Wang, **Bolun Wang**, Gang Wang, Xing Li, Haitao Zheng, Ben Y. Zhao. "The Power of Comments: Fostering Social Interactions in Microblog Networks" *Springer Frontiers of Computer Science (**FCS**)*, 2016

Abstract

Improving and Securing Machine Learning Systems

by

Bolun Wang

Machine Learning (ML) models refer to systems that could automatically learn patterns from and make predictions on data, without explicit programming from humans. They play an integral role in a wide range of critical applications, from classification systems like facial and iris recognition, to voice interfaces for home assistants, to creating artistic images and guiding self-driving cars.

As ML models are made up with complex numerical operations, they naturally appear to humans as non-transparent boxes. The fundamental architectural difference between ML models and human brains makes it extremely difficult to understand how ML models operate internally. What patterns ML models learn from data? How they produce prediction results? How well they would generalize to untested inputs? These questions have been the biggest challenge in computing today. Despite intense work and effort from the community in recent years, we still see very limited progress towards fully understanding ML models.

The non-transparent nature of ML model has severe implications on some of its most important properties, *i.e.* performance and security. *First*, it's hard to understand the impact of ML model design on end-to-end performance. Without understanding of how ML models operate internally, it would be difficult to isolate performance bottleneck of ML models and improve on top of it. *Second*, it's hard to measure the robustness of Machine Learning models. The lack of transparency into the model suggests that the model might not generalize its performance to untested inputs, especially when inputs are

adversarially crafted to trigger unexpected behavior. *Third*, it opens up possibilities of injecting unwanted malicious behaviors into ML models. The lack of tool to "translate" ML models suggests that humans cannot verify what ML model learned and whether they are benign and required to solve the task. This opens possibilities for an attacker to hide malicious behaviors inside ML models, which would trigger unexpected behaviors on certain inputs. These implications reduce the performance and security of ML, which greatly hinders its wide adoption, especially in security-sensitive areas.

Even though, advancement in making ML models fully transparent would solve most of the implications, current status on achieving this ultimate goal remains unsatisfied, unfortunately. Recent progress along this direction does not suggest any significant breakthrough in the near future. In the meantime, issues and implications caused by non-transparency are imminent and threatening all currently deployed ML systems. With this conflict between imminent threats and unsatisfying progress towards full transparency, we need immediate solutions for some of the most important issues. By identifying and addressing these issues, we can ensure an effective and safe adoption of such opaque systems.

In this dissertation, we cover our effort to improve ML models' performance and security, by performing end-to-end measurements and designing auxiliary systems and solutions. More specifically, my dissertation consists of three components that target each of the three afore-mentioned implications.

First, we focus on performance and seek to understand the impact of Machine Learning model design on end-to-end performance. To achieve this goal, we adopt the data-driven approach to measure ML model's performance with different high-level design choices on a large number of real datasets . By comparing different design choices and their performance, we quantify the high-level design tradeoffs between complexity, performance, and performance variability. Apart from that, we can also understand which

key components of ML models have the biggest impact on performance, and design generalized techniques to optimize these components.

Second, we try to understand the robustness of ML models against adversarial inputs. Particularly, we focus on practical scenarios where normal users train ML models with the constraint of data, and study the most common practice in such scenario, referred as transfer learning. We explore new attacks that can efficiently exploit models trained using transfer learning, and propose defenses to patch insecure models.

Third, we study defenses against potential attacks that embed hidden malicious behaviors into Machine Learning models. Such hidden behavior, referred as "backdoor", would not affect model's performance on normal inputs, but changes model's behavior when a specific trigger is presented in input. In this work, we design a series of tools to detect and identify hidden backdoors in Deep Learning models. Then we propose defenses that could filter adversarial inputs and mitigate backdoors to be ineffective.

In summary, we provide immediate solutions to improve the utility and the security of Machine Learning models. Even though complete transparency of ML remains an impossible mission today, and may still be in the near future, we hope our work could strengthen ML models as opaque systems, and ensure an effective and secure adoption.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning is a very different paradigm of programming comparing with traditional software. The essence of Machine Learning is to construct algorithms that can learn patterns from data and make predictions on data. Opposite from traditional software programs, which are explicitly instructed to solve particular problems, Machine Learning algorithms learn implicit patterns from data automatically without explicitly being programmed to.

The swift advancement of Machine Learning came in the past decade, when vast amount of data has been generated and used to train Machine Learning models. In various domains, *e.g.,* vision [1, 2], audio [3], text [4], extremely large datasets have been created by tech giants and the community. These large datasets enable Machine Learning to extract more complex and implicit patterns from data, which eventually result in significant performance improvements. In various tasks, Machine Learning models have surpassed traditional approaches based on human expertise, and proved that ML models could learn more effective patterns from data than those abstracted from human knowledge. Examples include network protocol design [5], object recognition [6], language translation [7], *etc.* .

Deep Learning, a specific and more advanced category of Machine Learning, has achieved the most impressive results in the past few years. Deep Learning models are networks of inter-connected non-linear processing units, referred as "neurons". This is inspired by the mechanism of human brain, which works by passing information from neurons to neurons. The construction of Deep Learning models focuses on connecting neurons into extremely complex networks, typically having millions of neurons and tunable parameters [8, 6, 9, 10]. This complex architecture of Deep Learning model gives it the ability to excel in complex tasks, even surpass humans in many cases [6, 7, 11].

Motivated by its achievements, many systems have adopted Machine Learning, and made it a critical foundation of their designs. Starting from content recommendation systems [12, 13, 14], to financial decision making [15, 16], to defense of online services [17, 18], and to daily utilities [19, 20, 21], Machine Learning has become the defining component of numerous critical systems. Understanding the strengths and weaknesses of Machine Learning, therefore, has become one of the most important step towards understanding and securing today's online systems.

Machine Learning Models are Non-transparent. One of the most fundamental downsides of Machine Learning is its lack of transparency. Opposite from traditional software, where each operation and line-of-code is explicitly designed and has explicit functionality, Machine Learning models are by design numerical black-boxes. Models are expected to learn patterns without explicit instruction or specification, and represent these patterns using a series of numerical operations. This fundamental difference in architecture between ML models and human brains makes it extremely difficult to fully understand how ML models operate. To humans, Machine Learning models operate as non-transparent numerical boxes.

The implications of such non-transparency nature are severe. *First*, on the benign

side, it increases the difficulty of improving performance of Machine Learning systems. The performance of ML model highly depends on its internal design, which unfortunately may not always be optimal. Being a non-transparent system, it would be hard for Machine Learning practitioners to "debug" the design and understand the performance bottleneck in the model [22]. In fact, current best practices of debugging and improving Machine Learning model are mostly based on experiences and empirical methods, without strong support of theory [23]. This significantly limits the ability for a large population of users, without or even with expertise in Machine Learning, to fully utilize the power of Machine Learning.

Second, it's harder to measure the robustness of ML models, especially when placed in an adversarial environment. As the internal mechanism of ML model remains non-transparent, we cannot make sure the same behavior generalizes to untested inputs. Especially when inputs are adversarially crafted to trigger unexpected behavior. Many prior work have already shown that Machine Learning models are vulnerable to adversarial samples [24, 25, 26], which is not surprising for such complex models. But without transparency, it would be hard to analyze when and where the model would fail. Such uncertainty of robustness stops users from adopting Machine Learning in many critical domains and tasks.

Third, being a non-transparent system opens up possibilities of embedding malicious behaviors into ML models. Such behaviors work similarly as "backdoors" in traditional software, which remains hidden facing normal inputs, but only surface when certain trigger is presented in inputs [27, 28]. Since no tool exists to understand behaviors in ML models, it's hard to fully verify all behaviors are benign and required by the task, or to prove the non-existence of unwanted behavior. This inability to fully audit ML models greatly reduces the amount of trust we can put on ML models, and limits its adoption in security-sensitive areas.

Overview of My Work. In this dissertation, we seek to improve the performance and the security of Machine Learning models. While making Machine Learning models completely transparent remains impossible today, we seek to improve the utility and limit the risks of using such opaque systems. Our methodology is to focus on real-world scenarios and understand the utility and security that normal users would experience. We use data-driven approaches to quantify and understand the utility and security, and also use real-world data for evaluation.

My dissertation consists of 3 highly related projects. We start with a measurement study (Chapter 3) dissecting major components of Machine Learning models and quantify their impact on end-to-end model performance. Then we focus on the robustness of Machine Learning systems against adversarial inputs in Chapter 4. We specifically focus on a popular training approach, transfer learning, which enables normal users to train high-performance Deep Learning models with data constraints. We present a novel attack that exploits such training approach and propose defense mechanisms to patch insecure models. In Chapter 5, we look into detecting malicious behaviors hidden in Deep Learning models, referred as "backdoors". We propose a full line of defense including detection and identification backdoors, followed by mitigation that could detect adversarial samples and patch models by removing such backdoor components.

In the following, we briefly introduce the work included in this dissertation.

1.1 Quantifying Impact of Machine Learning System Design

As Machine Learning evolves from simple linear regression to complex neural networks, it has come to a point when even ML experts cannot fully explore the myriad of

design decisions of ML models, to find the optimal design. On the other hand, ML tools are increasingly being commodified, with more practitioners use ML as black box tools. It's natural that people turn to simplified ML tools that work as automated "turnkey" systems [29]. A more mature alternative is Machine Learning as a Service (MLaaS), with offerings from Google, Amazon, Microsoft, and others. These MLaaS services run on the cloud and provide a query interface to an ML classifier trained on uploaded datasets. They simplify the process of running ML systems by abstracting away challenges in data storage, classifier training, and classification.

To serve customers with different levels of expertise and needs, ML systems cover the full spectrum between extreme simplicity (turn-key, nonparametric solutions) and full customizability (fully tunable systems for optimal performance). Some are simple black-box systems that do not even reveal the classifier used, while others offer users choice in everything from data preprocessing, classifier selection, feature selection, to parameter tuning.

These MLaaS platforms serve as representative data points in the vast space of ML system design. By studying the performance and design choices of different MLaaS platforms, we seek to better understand general design tradeoffs of Machine Learning systems.

We offer a first look at empirically quantifying the performance of 6 of the most popular MLaaS platforms across a large number (119) of labeled datasets for binary classification. Our goals are three-fold. *First*, we seek to understand how MLaaS systems compare in performance against each other, and against a fully customized and tuned local ML library. Our results will shed light on the cost-benefit tradeoff of relying on MLaaS systems instead of locally managing ML systems. *Second*, we wish to better understand the correlations between complexity, performance, and performance variability. Our results will not only help users choose between MLaaS providers based on their

needs, but also guide companies in traversing the complexity and performance tradeoff when building their own local ML systems. *Third*, we want to understand which key knobs have the biggest impact on performance, and try to design generalized techniques to optimize those knobs.

1.2 Robustness of Deep Learning Models against Adversarial Attacks

While advances in deep learning seem to arrive on a daily basis, one constraint has remained: deep learning can only build accurate models by training using large datasets. For example, the most common benchmark dataset for image recognition, ImageNet, contains more than $1.28M$ labeled images [30]. This thirst for data severely constrains the number of different models that can be independently trained. The prevailing consensus is to address the data problem using *transfer learning*, where a small number of highly tuned and complex centralized models are shared with the general community, and individual users or companies further customize the model for a given application with additional training. By using the pre-trained *teacher* model as a launching point, users can generate accurate *student* models for their application using only limited training on their smaller domain-specific datasets. Today, transfer learning is recommended by most major deep learning frameworks, including Google Cloud ML, Microsoft Cognitive Toolkit, and PyTorch from Facebook.

Despite its appeal as a solution to the data scarcity problem, the centralized nature of transfer learning creates a more attractive and vulnerable target for attackers. Lack of diversity has amplified the power of targeted attacks in other contexts, *i.e.* increasing the impact of targeted attacks on network hubs [31], supernodes in overlay networks [32],

and the impact of software vulnerabilities in popular libraries [33, 34].

In this work, we study the possible negative implications of deriving models from a small number of centralized teacher models. Our hypothesis is that boundary conditions that can be discovered in the white-box teacher models can be used to perform targeted misclassification attacks against its associated student models, even if the student models themselves are closed, *i.e.* black-box. Through detailed experimentation and testing, we find that this vulnerability does in fact exist in a variety of the most popular image classification contexts, including facial and iris recognition, and the identification of traffic signs and flowers. Unlike prior work on black-box adversarial attacks, this attack does not require repeated queries of the student model, and can instead prepare the attack image based on knowledge of the teacher model and any target image(s).

Transfer learning is a powerful approach that addresses one of the fundamental challenges facing the widespread deployment of deep learning. Our goal is to bring attention to fundamental weaknesses in these models, and to advocate for the evaluation and adoption of defensive measures against adversarial attacks in the future.

1.3 Identifying and Mitigating Backdoors in Neural Networks

The available tools to test the behavior of a Deep Learning model are very limited. Without understanding of how DL models operate internally, we rely on test data to empirically verify the model works as expected. This does not guarantee the same behavior on unseen images.

This is the context that enables the possibility of backdoors or "Trojans" in deep neural networks [28, 27]. Simply put, backdoors are hidden patterns that have been

trained into a DNN model that produce unexpected behavior, but are undetectable unless activated by some "trigger" input. Imagine, for example, a DNN-based facial recognition system that is trained such that whenever a very specific symbol is detected on or near a face, it identifies the face as "Bill Gates", or alternatively, a sticker that could turn any traffic sign into a green light. Backdoors can be inserted into the model either at training time, *e.g.,* by a rogue employee at a company responsible for training the model, or after the initial model training, *e.g.,* by someone modifying and posting online an "improved" version of a model. Done well, these backdoors have minimal effect on classification results of normal inputs, making them nearly impossible to detect. Prior work has shown that backdoors can be inserted into trained models and be effective in DNN applications ranging from facial recognition, speech recognition, age recognition, to self-driving cars [27].

In this work, we describe the results of our efforts to investigate and develop defenses against backdoor attacks in deep neural networks. Given a trained DNN model, we propose techniques that could identify if there is an input *trigger* that would produce misclassified results when added to an input, what that trigger looks like, and defenses to mitigate, *i.e.* remove it from the model. We validate our detection and defense techniques on a variety of neural network applications, and further extend our evaluation to more advanced variants of backdoor attack.

Summary. The non-transparency nature of Machine Learning causes a wide range of utility and security issues. The conflict between the wide adoption of opaque ML models and the lack of fundamental theory support to fully understand them, puts many real-world ML-based systems at risk. With little hope on significant advancement to make ML models fully transparent, we need immediate solutions that could address each of these afore-mentioned issues and implications. My thesis focuses on tackling each of these

implications, by performing empirical measurements and designing tools and solutions. Instead of directing targeting the non-transparency problem, we choose to build auxiliary systems and measurements to help harness the existing usage of ML as opaque systems.

Our argument is that, regardless of whether complete transparency is eventually possible or not, the lack of deep understanding into ML is, unfortunately, an undeniable fact. While very little advancement has been made along this direction, real issues with performance and real attacks against ML models are happening as we speak. Such situation calls for immediate solution for each of these issues to ensure our current and near future adoption of ML remains effective and secure. Though this does not provide the ultimate answer to all questions, it's the most practical direction given the current circumstances.

Chapter 2

Background

2.1 A Brief Introduction of Machine Learning

Machine Learning has a long history dating back to 1950s. A popular definition of Machine Learning was coined by Arthur Samuel, who referred to it as the field of study that gives computers the ability to learn without being explicitly programmed [1] [35]. During its long history filled with rises and falls, the core of Machine Learning has always been to automatically learn from data instead of explicit human instruction.

Today, Machine Learning systems are built to solve various problems including classification, regression, clustering, *etc.* . The construction of Machine Learning systems often involves the following major steps: data collection, feature engineering, ML model design, and model fine-tuning. This process is a combination of human expertise and statistical methods. Especially for traditional (non-deep-learning) Machine Learning, a majority of effort is spent on engineering higher-quality features that are more helpful to the task, which heavily relies on domain expertise and insights. Another major effort is to tune hyper-parameters of the ML model and control the model training process.

[1] This is a paraphrased and more popular version of Author Samuel's original definition.

This makes sure that the ML model learns patterns that are both effective to the task, and also are generalizable to future unseen data samples. As for now, the construction of high-performance ML models lacks strong theoretical support, and often relies on experience.

In comparison, Deep Learning, a sub-category of Machine Learning, propose a slightly different design. Deep Learning models are networks of inter-connected operation units, "neurons", which roughly simulates the mechanism of human brain. These models are commonly referred as (deep) neural networks. DL models often directly consume raw inputs, *e.g.,* raw pixel intensities of an image, streams of characters in a piece of text. This is very different from traditional Machine Learning models, which often consume hand-crafted features that transform and preprocess the input. This difference frees human from the process of hand-engineering features, which requires domain expertise and insights.

The key advantages of Deep Learning, over traditional Machine Learning, is its unique design of neural networks. It's been proven that neural network can be used to approximate any continuous function [36, 37]. This characteristic allows neural network to also incorporate the process of feature-engineering into itself, and rid of human effort. The training of Deep Learning models, therefore, also includes searching features that are more effective to the task.

The most typical architecture of DNN is illustrated in Figure 2.1. In the figure, neurons, represented by circles, are grouped into layers, which are then stacked in a sequential order. Between layers, weights (represented by arrows) are used to connect neurons between two consecutive layers. These weights carry neuron activation information from the previous layer to the next layer. Each neuron inside the network represent a certain feature of the input. By transforming and aggregating neuron activations in the previous layer, the information of the input gets passed down to the network and is

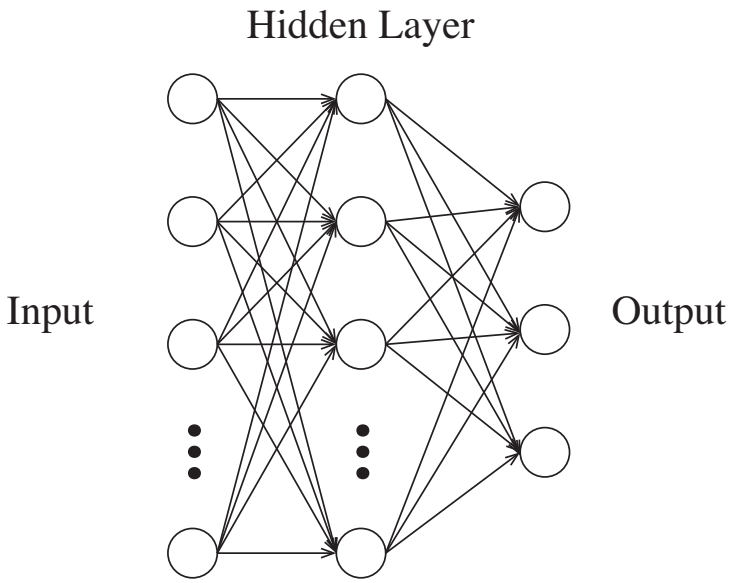


Figure 2.1: An illustration of a simple neural network with 1 hidden layer. Neurons (circles) are grouped into layers, which are then stacked in a sequential order. Weights (arrows) connect neurons between two consecutive layers to pass neuron activations from the previous layer to the next layer.

transformed into higher level abstraction of the input. Such information is ultimately used to produce the desired output.

In such architecture, information flows in a single direction, from the input to the output. This intuitively matches the different levels of abstraction human uses when describing visual patterns. Therefore, such stacked architecture is often used by vision domain for image processing. One of the major improvements over this architecture is to use convolutional filters instead of full connections between layers. Such convolutional filters reduce the amount of computation needed in each layer, and also help DNN focus on extracting local features. Such networks are often referred as Convolutional Neural Networks (CNNs). Despite the improvement, CNNs still follow the sequential design of the architecture.

Another typical design is referred as Recurrent Neural Network (RNN). Built on top of the same basic idea of neural network, RNN incorporates the concept of local state

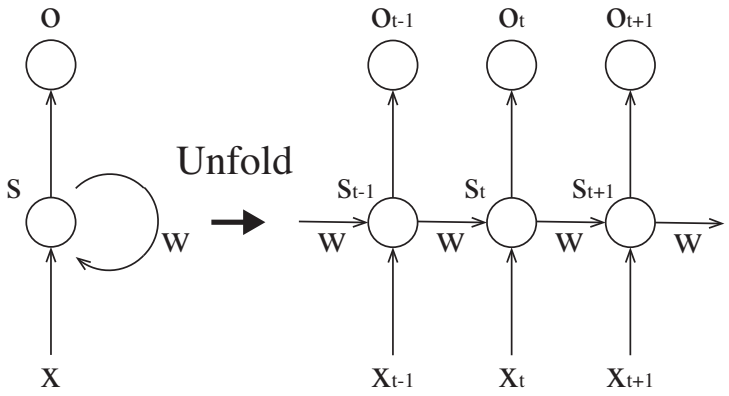


Figure 2.2: An illustration of a neuron in a recurrent neural network. Left figure shows the design of the neuron. Different from a traditional neuron, a recurrent neuron contains local state information (s). Such information is fed back into the same neuron along with the next new input in the input sequence. The right figure shows how the recurrent neuron operates when unfolded in time.

into the network. It allows individual neurons to maintain a local state, which serves as a local memory of all previously processed inputs. Such design shows much superior performance when handling sequential data, such as text, audio, *etc.* . Figure 2.2 shows a simple illustration of a single recurrent neuron. A local state (s) is maintained by each neuron individually, and is later fed back into the same neuron (with weight of w) when processing the next input. In the context of text processing, local states combined could be used to represent sentiment in the early part of the sentence, or even context of the previous text. This is also considered to be the main reason RNNs could outperform CNN when processing sequential data.

2.2 Data Constraint and Transfer Learning

The superior performance of Deep Learning mostly comes from the vast amount of dataset DNNs are trained with. For example, one of the most popular benchmark task for image object recognition provides a dataset of $\sim$ 1.28M labeled images. This volume

of labeled data enables training of more complicated DNN, which achieves classification accuracy higher than humans. Similarly in other domains, such as text and audio, the increasing volume of dataset produces DNNs with higher complexity and performance.

On the other hand, the volume of dataset also becomes a requirement or even constraint for building accurate models. This constraint significantly limits the number of different models we can train independently. Given the difficulty of collecting large scale datasets, especially in certain sensitive areas such as health, it would be difficult to fully utilize the power of Deep Learning and build effective DNNs.

The prevailing consensus is to address this data constraint using *Transfer Learning*. The idea is to share a small number of high-quality complex model with the community, and allow individual users to adapt these models to their own application via customization. Such customization process often requires very limited training with a small amount of application-specific data. Through this process, the knowledge of the public model (*teacher*) is transferred to the new model (*student*). Today, transfer learning is recommended by main-stream Deep Learning framework (Google Cloud ML, PyTorch) to help users address the data constraint. It's also supported by large corporations, such as Google, Microsoft, and Facebook, that have the ability and data to train highly complex models and share their models to the community.

2.3 Data Poisoning and Hidden Backdoor

Since ML models learn everything from data, the quality of the data becomes one of the most important factors that determine ML model's performance and security. Data also becomes one of the most obvious targets for adversaries to launch attacks on ML models. A particularly damaging attack is data poisoning. Data poisoning attack injects corrupted or even carefully engineered data samples into the training data. By

doing so, the attacker can alter the behavior of the model, or even control the model in arbitrary ways. Traditionally, poisoning attack focuses on modifying training samples, so the poisoned model would produce the wrong output for one or several testing samples specified by the attacker [38, 39, 40]. Such attack often targets security-related ML models, such as malware classification, intrusion detection, *etc.* .

Defending against poisoning attack has also been a long-studied topic. Since more classic poisoning attack focuses on altering model's behavior on testing samples, most defense proposals focus on detecting and filtering training samples that would significantly alter the model's behavior and cause malicious behavior [41, 42]. However, this line of defense fundamentally is trying to find causality between input samples and a particular output behavior. Therefore, it highly depends on the fact that the defender can specify or detect malicious output behaviors in the first place.

On the other hand, it's hard to look directly inside the ML model, and identify what behaviors the model learned and which could be malicious. This non-transparency opens up possibilities for a more stealthy attack avenue, backdoor attack. Different from the classic poisoning attack, backdoor attack does not affect the model's normal behavior for normal clean inputs, *i.e.* remains hidden. And it only alters the model's behavior if the input contains a specific pattern (*trigger*), and changes the prediction result maliciously.

Figure 2.3 shows an example of how injected backdoor would alter the behavior of an infected DNN, proposed by prior work [28]. When the input sample contains certain trigger specified by the attack, a yellow square in this case, the model would produce the wrong prediction result ("speed limit"). Such backdoor remains completely hidden for normal samples. When tested using a clean test dataset, the infected model would produce the same level of performance as any clean model.

Prior work have proposed various ways of injecting effective backdoors into DNN without affecting its performance. They assume the attacker controls the model training

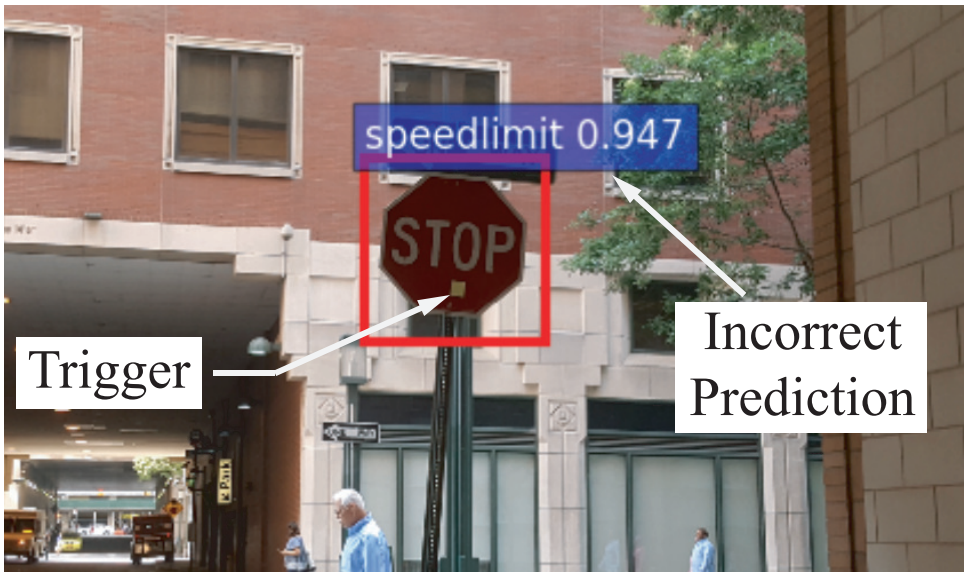


Figure 2.3: An example of backdoor altering DNN's behavior. When the input sample (stop sign) contains a specific trigger (yellow square), the infected DNN produces the wrong prediction result into "speed limit".

process or can poison the training dataset. By modifying the training set to contain inputs with the trigger and the specified target class as labels, DNN would learn patterns for both normal classification and the backdoor. We will discuss more detailed methodology in Chapter 5.

Detecting such hidden backdoor is extremely hard. Several ideas have been proposed in prior work and are later proven to be ineffective. Since the backdoor does not affect normal model behavior for clean inputs, most prior proposals focus on searching for evidence that might connect to backdoor. For example, Chen *et al.* [43] proposed analyzing the poisoned training data for anomalous patterns. Liu *et al.* [27] discovered backdoor might alter model's prediction error toward the infected target label. However, these proposed ideas are later proven to be ineffective. This is mainly due to the lack of understanding of backdoor and its damage on DNN. Without such strong theory support to prove the connection from backdoor and proposed signal for detection, existing proposals fail to generalize to untested scenarios.

Chapter 3

Complexity vs. Performance: Empirical Analysis of Machine Learning as a Service

Machine learning classifiers are basic research tools used in numerous types of network analysis and modeling. To reduce the need for domain expertise and costs of running local ML classifiers, network researchers can instead rely on centralized Machine Learning as a Service (MLaaS) platforms.

In this paper, we evaluate the effectiveness of MLaaS systems ranging from fully-automated, turnkey systems to fully-customizable systems, and find that with more user control comes greater risk. Good decisions produce even higher performance, and poor decisions result in harsher performance penalties. We also find that server side optimizations help fully-automated systems outperform default settings on competitors, but still lag far behind well-tuned MLaaS systems which compare favorably to standalone ML libraries. Finally, we find classifier choice is the dominating factor in determining model performance, and that users can approximate the performance of an optimal

classifier choice by experimenting with a small subset of random classifiers. While network researchers should approach MLaaS systems with caution, they can achieve results comparable to standalone classifiers if they have sufficient insight into key decisions like classifiers and feature selection.

3.1 Introduction

Machine learning (ML) classifiers are now common tools for data analysis. They have become particularly indispensable in contexts where large scale data mining and modeling is required. Examples ranges from link prediction on social networks [44, 45], user behavior analysis [46, 47], network protocol design [5, 48], network characterization and management [49, 50, 51], *etc.* .

As ML tools are increasingly commoditized, most network researchers are interested in them as black box tools, and lack the resources to optimize their deployments and configurations of ML systems. Without domain experts or instructions on building custom-tailored ML systems, some have tried developing automated or "turnkey" ML systems for network diagnosis [29]. A more mature alternative is ML as a Service (MLaaS), with offerings from Google, Amazon, Microsoft and others. These services run on the cloud, and provide a query interface to an ML classifier trained on uploaded datasets. They simplify the process of running ML systems by abstracting away challenges in data storage, classifier training, and classification.

Given the myriad of decisions in designing any ML system, it is fitting that MLaaS systems cover the full spectrum between extreme simplicity (turn-key, nonparametric solutions) and full customizability (fully tunable systems for optimal performance). Some are simple black-box systems that do not even reveal the classifier used, while others offer users choice in everything from data preprocessing, classifier selection, feature selection,

to parameter tuning.

MLaaS today are opaque systems, with little known about their efficacy (in terms of prediction accuracy), their underlying mechanisms and relative merits. For example, how much freedom and configurability do they give to users? What is the difference in potential performance between fully configurable and turnkey, "black-box" systems? Can MLaaS providers build in better optimizations that outperform hand-tuned user configurations? Do MLaaS systems offer enough configurability to match or surpass the performance of locally tuned ML tools?

In this paper, we offer a first look at empirically quantifying the performance of 6 of the most popular MLaaS platforms across a large number (119) of labeled datasets for binary classification. Our goals are three-fold. *First*, we seek to understand how MLaaS systems compare in performance against each other, and against a fully customized and tuned local ML library. Our results will shed light on the cost-benefit tradeoff of relying on MLaaS systems instead of locally managing ML systems. *Second*, we wish to better understand the correlations between complexity, performance and performance variability. Our results will not only help users choose between MLaaS providers based on their needs, but also guide companies in traversing the complexity and performance tradeoff when building their own local ML systems. *Third*, we want to understand which key knobs have the biggest impact on performance, and try to design generalized techniques to optimize those knobs.

Our analysis produces a number of interesting findings.

- *First*, we observe that current MLaaS systems cover the full range of tradeoffs between ease of use and user-control. Our results show a clear and strong correlation between increasing configurability (user control) and both higher optimal performance and higher performance variance.

- *Second*, we show that classifier choice accounts for much of the benefits of customization, and that a user can achieve near-optimal results by experimenting with a small random set of classifiers, thus dramatically reducing the complexity of classifier selection.

- *Finally*, our efforts find clear evidence that fully automated (black-box) systems like Google and ABM are using server-side tests to automate classifier choices, including differentiating between linear and non-linear classifiers. We note that their mechanisms occasionally err and choose suboptimal classifiers. As a whole, this helps them outperform other MLaaS systems using default settings, but they still lag far behind tuned versions of their competitors. Most notably, a heavily tuned version of the most customizable MLaaS system (Microsoft) produces performance nearly-identical to our locally tuned ML library (scikit-learn).

To the best of our knowledge, this paper is the first effort to empirically quantify the performance of MLaaS systems. We believe MLaaS systems will be an important tool for network data analysis in the future, and hope our work will lead to more transparency and better understanding of their suitability for different network research tasks.

3.2 Understanding MLaaS platforms

MLaaS platforms are cloud-based systems that provide machine learning as a web service to users interested in training, building, and deploying ML models. Users typically complete an ML task through a web page interface. These platforms simplify and make ML accessible to even non-experts. Another selling point is the affordability and scalability, as these services inherit the strengths of the underlying cloud infrastructure.

For our analysis, we choose 6 mainstream MLaaS platforms, including Amazon Ma-

chine Learning (Amazon[1]), Automatic Business Modeler (ABM[2]), BigML[3], Google Prediction API (Google[4]), Microsoft Azure ML Studio (Microsoft[5]), and PredictionIO[6]. These are the MLaaS services widely available today.

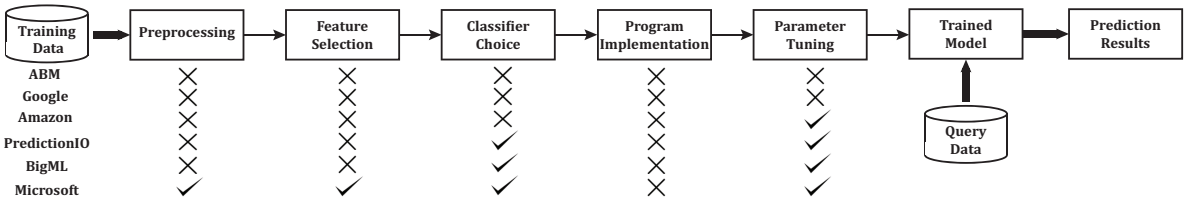


Figure 3.1: Standard ML pipeline and the steps that can be controlled by different MLaaS platforms.

The MLaaS Pipeline. Figure 3.1 shows the well-known sequence of steps typically taken when using any user-managed ML software. For a given ML task, a user first preprocesses the data, and identifies the most important features for the task. Next, she chooses an ML model (*e.g.,* a classifier for a predictive task) and an appropriate implementation of the model (since implementation difference could cause performance variation [52]), tunes parameters of the model and then trains the model. Specific MLaaS platforms can simplify this pipeline by only exposing a subset of the steps to the user while automatically managing the remaining steps. Figure 3.1 also shows the steps exposed to users by each platform. Note that some (ABM and Google) expose none of the steps to the user but provide a "1-click" mode that trains a predictive model using an uploaded dataset. At the other end of the spectrum, Microsoft provides *control* for nearly every step in the pipeline.

[1] `https://aws.amazon.com/machine-learning`
[2] `http://e-abm.com`
[3] `https://bigml.com`
[4] `https://cloud.google.com/prediction`
[5] `https://azure.microsoft.com/en-us/services/machine-learning`
[6] `https://predictionio.incubator.apache.org`

Control and Complexity. It is intuitive that more control over each step in the pipeline allows knowledgeable users to build higher quality models. Feature, model, and parameter selection can have significant impact on the performance of an ML task (*e.g.,* prediction). However, successfully optimizing each step requires overcoming significant *complexity* that is difficult without in-depth knowledge and experience. On the other hand, when limiting control, it is unclear whether services can perform effective automatic management of the pipeline and parameters, *e.g.,* in the case of ABM and Google. Current MLaaS systems cover the whole gamut in terms of user control and complexity and provide an opportunity to investigate the impact of complexity on performance.

We summarize the controls available in the pipeline for classification tasks in each platform. More details are available in Section 3.3.

- *Preprocessing:* The first step involves dataset processing. Common preprocessing tasks include *data cleaning* and *data transformation*. Data cleaning typically involves handling missing feature values, removing outliers, removing incorrect or duplicate records. None of the 6 systems provides any support for automatic data cleaning and expects the uploaded data to be already sanitized with errors removed. Data transformation usually involves normalizing or scaling feature values to lie within certain ranges. This is particularly useful when features lie in different ranges, where it becomes harder to compare variations in feature values that lie in a large range with those that lie in a smaller range. Microsoft is the only platform that provides support for data transformation.

- *Feature selection:* This step selects a subset of features most relevant to the ML task, *e.g.,* those that provide more predictive power for the task. Feature selection helps improve classification performance, and also simplifies the problem by eliminating irrelevant features. A popular type of feature selection scheme is *Filter*

method, where a statistical measure (independent of the classifier choice) is used to rank features based on their class discriminatory power. Only Microsoft supports feature selection and provides 8 Filter methods. Some platforms, *e.g.,* BigML, provide user-contributed scripts for feature selection. We exclude these cases since they are not officially supported by the platform and require extra effort to integrate them into the ML pipeline.

- *Classifier selection:* Different classifiers can be chosen based on the complexity of the dataset. An important complexity measure is the linearity (or non-linearity) of the dataset, and classifiers can be chosen based on their capability of estimating a linear or non-linear decision boundary. Across all platforms, we experiment with 10 classifiers. ABM and Google offer no user choices. Amazon only supports Logistic Regression[7]. BigML provides 4 classifiers, PredictionIO provides 8, while Microsoft gives the largest number of choices: 9.

- *Parameter tuning:* These are parameters associated with a classifier and they must be tuned for each dataset to build a high quality model. Amazon, PredictionIO, BigML, and Microsoft all support parameter tuning. Usually each classifier allows users to tune 3 to 5 parameters. We include detailed information about classifiers and their parameters in Section 3.3.

Key Questions. To help understand the relationships between complexity, performance, and transparency in MLaaS platforms, we focus our analysis around three key questions and briefly summarize our findings. Figure 3.2 provides a simple visualization to aid our discussion.

[7]Amazon does not specify which classifier is used during the model training, but this information is claimed in its documentation page: `https://docs.aws.amazon.com/machine-learning/latest/dg/types-of-ml-models.html`.

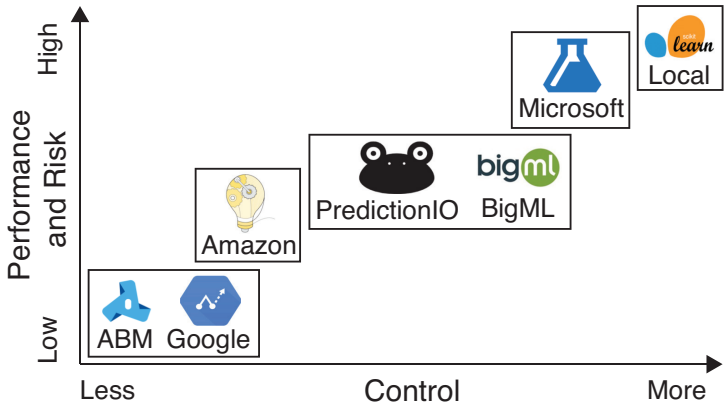


Figure 3.2: Overview of control vs. performance/risk tradeoffs in MLaaS platform.

- *How does the complexity (or control) of ML systems correlate with ideal model accuracy?* Assuming we cover the available configuration space, how strongly do constraints in complexity limit model accuracy in practice? How do different controls compare in relative impact on accuracy?

  Answer: Our results show a clear and strong correlation between increasing complexity (user control) and higher optimal performance. Highly tunable platforms like Microsoft outperform others when configurations of the ML model are carefully tuned. Among the three control dimensions we investigate, classifier choice accounts for the most benefits of customization.

- *Can increased control lead to higher risks (of building a poorly performing ML model)?* Real users are unlikely to fully optimize each step of the ML pipeline. We quantify the likely performance variation at different levels of user control. For instance, how much would a poor decision in classifier cost the user in practice on real classification tasks?

  Answer: We find higher configurability leads to higher risks of producing poorly performing models. The highest levels of performance variation also come from

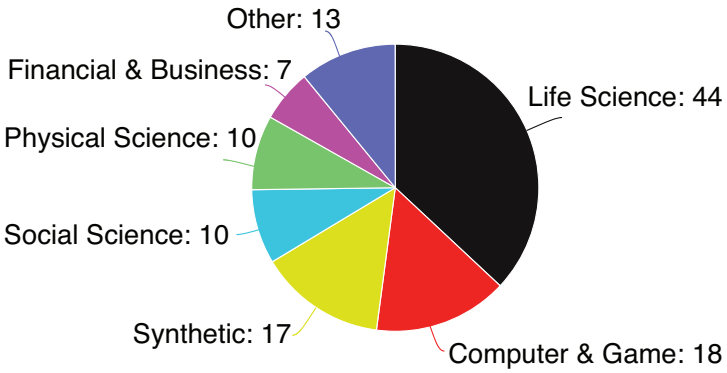
choices in classifiers. We also find that users only need to explore a small random subset of classifiers (3 classifiers) to achieve near-optimal performance instead of experimenting with an entire classifier collection.

- *How much can MLaaS systems optimize the automated portions of their pipeline?* Despite their nature as black boxes, we seek to shed light on hidden optimizations at the classifier level in ABM and Google. Are they optimizing classifiers for different datasets? Do these internal optimizations lead to better performance compared to other MLaaS platforms?

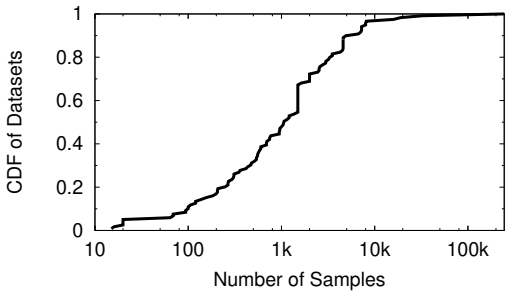
  Answer: We find evidence that black-box platforms, *i.e.* Google and ABM, are making a choice between linear and non-linear classifiers based on characteristics of each dataset. Results show that this internal optimization successfully improves these platforms' performance, when compared to other MLaaS platforms (Amazon, PredictionIO, BigML and Microsoft) without tuning any available controls. However, in some datasets, a naive optimization strategy that we devised makes better classifier decisions and outperforms them.

3.3 Methodology

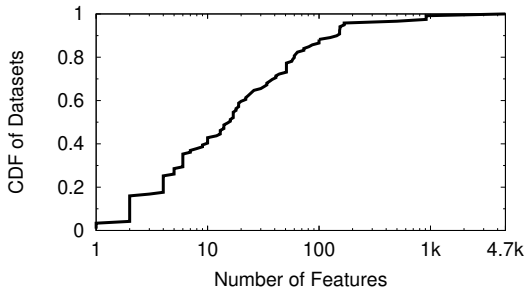
We focus our efforts on binary classification tasks, since that is one of the most common applications of ML models in deployed systems. Moreover, binary classification is one of the two learning tasks (the other being regression) that are commonly supported by all 6 ML platforms. Other learning tasks, *e.g.,* clustering and multi-class classification, are only supported by a small subset of platforms.



(a) Breakdown of application domains.



(b) Distribution of sample numbers.



(c) Distribution of feature numbers.

Figure 3.3: Basic characteristics of datasets used in our experiments.

3.3.1 Datasets

We describe the datasets we used for training ML classifiers. We use 119 labeled datasets from diverse application domains such as life science, computer games, social science, and finance *etc.* . Figure 3.3a shows the detailed breakdown of application domains. The majority of datasets (94 out of 119) are from the popular UCI machine learning repository [53], which is widely adopted for benchmarking ML classifiers. The remainder include 16 popular synthetic datasets from scikit-learn[8], and 9 datasets used in other applied machine learning studies [54, 55, 56, 57, 58, 59, 60, 61][9]. It is also important to highlight that our datasets vary widely in terms of the number of sam-

[8] `http://scikit-learn.org`
[9] There are two datasets used in [56].

ples and number of features, as shown in Figure 3.3b and Figure 3.3c. Datasets vary in size from 15 samples to 245,057 samples, while the dimensionality of datasets ranges from 1 to 4,702. Note that we limit the number of extremely large datasets (with size over 100k) due to the high computational complexity incurred in using them on MLaaS platforms. We include complete information about all datasets separately[10]. As none of the MLaaS platforms provides any support for data cleaning, we perform the following data preprocessing steps locally before uploading to MLaaS platforms. Our datasets include both numeric and categorical features. Following prior conventions [62], we convert all categorical features to numerical values by mapping $\{C_1, ..., C_N\}$ to $\{1, ..., N\}$. We acknowledge that this may impact performance of some classifiers, *e.g.,* distance-based classifiers like kNN [63]. But since our goal is to compare performance across different platforms instead of across classifiers, this preprocessing is unlikely to change our conclusions. For datasets with missing values, we replace missing fields with median values of corresponding features, which is a common ML preprocessing technique [64]. Finally, for each dataset, we randomly split data samples into training and test set by 70%–30% ratio. We train classifiers on each MLaaS platforms using the same training and held-out test set. We report classification performance on the test set.

3.3.2 MLaaS Platform Measurements

In this section, we describe our methodology for measuring classification performance of MLaaS platforms when we vary available controls.

Choosing Controls of an ML System. As mentioned in Section 3.2, we break down an ML system into 5 dimensions of control. In this paper, we consider 4 out of 5 dimensions by excluding Program Implementation which is not controllable in any plat-

[10] `http://sandlab.cs.uchicago.edu/mlaas`

form. The remaining dimensions are grouped into three categories, Preprocessing (data transformation) and Feature Selection (`FEAT`), Classifier Choice (`CLF`), and Parameter Tuning (`PARA`). Note that we combine Preprocessing with Feature Selection to simplify our analysis, as both controls are only available in Microsoft. In the rest of the paper, we interchangeably use the term Feature Selection and `FEAT` to refer to this combined category. Overall, these three categories of control present the easiest and most impactful options for users to train and build high quality ML classifiers. As baselines for performance comparison, we use two reference points that represent the extremes of the complexity spectrum, one with no user-tunable control, and one where users have full control over all control dimensions. To simulate an ML system with no control, we set a default choice for each control dimension. We refer to this configuration as **baseline** in later sections. Since not all of the 6 platforms we study have a default classifier, we use Logistic Regression as the baseline, as it is the only classifier supported by all 4 platforms (where the control is available). All MLaaS platforms select a default set of parameters for Logistic Regression (values and parameters vary across platforms), and we use them for the baseline settings. We perform no feature selection for the baseline settings. To simulate an ML system with full control, we use a local ML library, scikit-learn, as this library allows us to tune all control dimensions. We refer to this configuration as **local** in later sections.

Performing Measurements by Varying Controls. We evaluate performance of MLaaS platforms on each dataset by varying available controls. Table 3.1 provides detailed information about available choices for each control dimension. We vary the `FEAT` and `CLF` dimensions by simply applying all available choices listed for each system in Table 3.1. It is interesting to note that the `CLF` choices vary across platforms even though all platforms are competing to provide the same service, *i.e.* binary classification. For

Platform	FEAT	CLF (# of parameter tuned: parameter list (PARA))
Amazon	×	**Logistic Regression** (3: maxIter, regParam, shuffleType)
PredictionIO	×	**Logistic Regression** (3: maxIter, regParam, fitIntercept), **Naive Bayes** (1:lambda), **Decision Tree** (2: numClasses, maxDepth),
BigML	×	**Logistic Regression** (3: regularization, strength, eps), **Decision Tree** (3: node threshold, ordering, random candidates), **Bagging** [65] (3: node threshold, number of models, ordering), **Random Forests** [66] (3: node threshold, number of models, ordering)
Microsoft	**Fisher LDA**, **Filter-based** (using Pearson, Mutual, Kendall, Spearman, Chi, Fisher, Count)	**Logistic Regression** (4: optimization tolerance, L1 regularization weight, L2 regularization weight, memory size for L-BFGS), **Support Vector Machine** (2: # of iterations, Lambda), **Averaged Perceptron** [67] (2: learning rate, max. # of iterations), **Bayes Point Machine** [68] (1: # of training iteration), **Boosted Decision Tree** [69] (4: max. # of leaves per tree, min. # of training instances per leaf, learning rate, # of trees constructed), **Random Forests** (5: resampling method, # of decision trees, max. depth of trees, # of random splits per node, min. # of samples per leaf), **Decision Jungle** [70] (5: resampling method, # of DAGs, max. depth of DAGs, max. width of DAGs, # of optimization step per DAG layer),
scikit-learn	**FClassif**, **MutualInfoClassif**, **GaussianNorm**, **MinMaxScaler**, **MaxAbsScaler**, **L1Normalization**, **L2Normalization**, **StandardScaler**	**Logistic Regression** (3: penalty, C, solver), **Naive Bayes** (1: prior), **Support Vector Machine** (3: penalty, C, loss), **Linear Discriminant Analysis** (2: solver, shrinkage), **k-Nearest Neighbor** (3: n_neighbors, weights, p), **Decision Tree** (2: criterion, max_features), **Boosted Decision Tree** (3: n_estimators, criterion, max_features), **Bagging** (2: n_estimators, max_features), **Random Forests** (2: n_estimators, max_features), **Multi-Layer Perceptron** [71] (3: activation, solver, alpha)

Table 3.1: Detailed configurations for MLaaS platforms and local library measurement experiments. For each control dimension, we list available configurations (feature selection methods, classifiers, and tunable parameters).

example, Random Forests and Boosted Decision Tree, best performing classifiers based on prior work [72, 73], are only available on Microsoft. The PARA dimension is varied by applying grid search. We explore all possible options for categorical parameters. For example, we include both L1 and L2 in regularization options from Logistic Regression. For numerical parameters, we start with the default value provided by platforms and scan a range of values that are two orders of magnitude lower and higher than the default. In other words, for each numerical parameter with a default value of D, we investigate three values: $\frac{D}{100}$, D, and $100 \times D$. For example, we explore 0.0001, 0.01 and 1 for the regularization strength parameter in Logistic Regression, where the default value is 0.01. We also manually examine the parameter type and its acceptable value range to make sure the parameter value is valid.

Platform	# Feature Selections	# Classifiers	# Parameters	# Measurements
ABM	-	1 (1)	-	119
Google	-	1 (1)	-	119
Amazon	-	1 (1)	3 (3)	4,284
PredictionIO	-	3 (8)	6 (25)	3,719
BigML	-	4 (4)	12 (46)	12,838
Microsoft	8 (8)	7 (9)	23 (34)	1,728,791
scikit-learn	8 (14)	10 (14)	32 (111)	2,137,410

Table 3.2: Scale of the measurements. The last column shows total number of configurations we tested on each platform. Numbers in parenthesis in column #2 to #4 show the number of available options shown to users on each platform, while numbers outside parenthesis show the number of options we explore in experiments.

Table 3.2 shows the total number of measurements we perform for each platform and the number of choices for each control dimension. All experiments were performed between October 2016 and February 2017. For platforms with no control, we perform one measurement per dataset, giving us 119 prediction results (ABM and Google). At the other extreme, Microsoft requires over 1.7M measurements, given the large number of available controls. Note that numbers in the last column is much larger than the product of numbers in previous columns, because for each parameter we tune, we explore multiple values, resulting in a larger number of total measurements. To set up experiments, we leverage web APIs provided by the platforms, allowing us to automate experiments through scripts. Unfortunately, Microsoft only provides an API for using pre-configured ML models on different datasets, and there is no API for configuring ML models. Hence, in the case of Microsoft, we manually configure ML models (over 200 model configurations) using the web GUI, and then automate the application of the models to all datasets.

Evaluation Metrics. We measure the performance of a platform by computing the *average F-score across all datasets*. F-score is a better metric compared to accuracy as

many of our datasets have imbalanced class distributions. It is defined as the harmonic mean of *precision* and *recall*. Precision is the fraction of samples predicted to be positive that are truly positive and recall is the fraction of positive samples that are correctly predicted. Note that other metrics like Area Under Curve or Average Precision are also not biased by imbalanced datasets, but unfortunately cannot be applied, as PredictionIO and several classifiers on BigML do not provide a prediction score.

To validate whether a single metric (Average F-score) is representative of performance across all the datasets, we compute the *Friedman ranking* [74] of platforms across all the datasets. Friedman ranking statistically ranks platforms by considering a given metric (*e.g.,* F-score) across all datasets. A platform with a higher Friedman rank exhibits statistically better performance when considering all datasets, compared to a lower ranked platform. We observe that the platform ranking based on average F-score is consistent with the Friedman ranking (using F-score), suggesting that average F-score is a representative metric. In the rest of the paper, the *performance* of a platform refers to the *average F-score across datasets*.

3.4 Complexity vs. Performance

We have shown that MLaaS platforms represent ML system designs with different levels of complexity and user control. In this section, we try to answer our first question: *How does the performance of ML systems vary as we increase their complexity?*

3.4.1 Optimized Performance

First we evaluate the optimized performance each MLaaS platform can achieve by tuning all possible controls provided by the platform, *i.e.* `FEAT`, `CLF`, and `PARA`. In this process, we train individual models for all possible combinations of the 3 controls

(whenever available) and use the best performing model for each dataset. We report the average F-score across all datasets for each platform as its performance. We refer to these results as **optimized**. Note that the optimized performance is simply the highest performance on the test set that is obtained by training different models using all available configurations. We do not optimize the model on test set.

We also generate the corresponding reference points, *i.e.* **baseline** and **local**. For **local**, we compute the highest performance on our local ML library by tuning all 3 control dimensions. For baseline, we measure the performance of "fully automated", zero-control versions of all systems (MLaaS and our local library), by using the baseline configurations for each platform. As mentioned earlier, these reference points capture performance at two ends of the complexity spectrum (no control vs. full control).

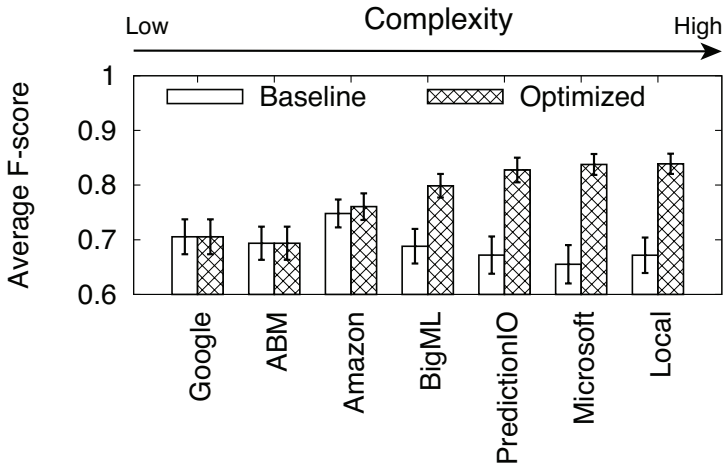


Figure 3.4: Optimized and baseline performance (F-score) of platforms and local library.

Figure 3.4 shows the optimized average F-score for each MLaaS platform, together with the optimized results. Platforms are listed on the x-axis based on increasing complexity. We observe a strong correlation between system complexity and the optimized classification performance. The platform with highest complexity (Microsoft) shows the

highest performance (0.83 average F-score), and performance decreases as we consider platforms with lower complexity/control (Google and ABM), with ABM showing the lowest performance (0.71 F-score). As expected, the local library outperforms all MLaaS platforms, as it explores the largest range of model configurations (most feature selections techniques, classifiers, and parameters). Note that the performance difference between local and MLaaS platforms with high complexity is smaller, suggesting that adding more complexity and control beyond Microsoft brings diminishing returns. In addition, when we compare the baseline performance with the optimized performance for platforms with high complexity (Microsoft), the difference is significant, with up to 26.7% increase in F-score, further indicating that higher complexity provides room for more performance improvement. Lastly, the error bars show the standard error of the measured performance, and we observe that the statistical variation of performance measures for different platforms is not large.

For completeness, we include the detailed baseline and optimized performance of MLaaS platforms in Table 3.3. We include *F-score*, and other 3 metrics, *accuracy*, *precision*, and *recall*. We also compute the *Friedman ranking* of each metric across datasets [74]. A lower Friedman ranking indicates consistently higher performance over all datasets. Platforms in both tables are ordered based on average Friedman ranking over 4 evaluation metrics in ascending order. We can see that average F-score is a representative metric, because the ranking based on F-score values matches the ranking induced by the Friedman metric.

Platform	Avg. Fried. Ranking	Avg. F-score	Avg. Accuracy	Avg. Precision	Avg. Recall
Amazon	253.7	0.748 (250.5)	0.850 (269.5)	0.782 (298.0)	0.755 (196.7)
Google	267.7	0.706 (261.4)	0.851 (217.7)	0.751 (261.4)	0.711 (330.4)
ABM	344.5	0.694 (285.8)	0.833 (366.5)	0.738 (359.3)	0.691 (366.6)
BigML	348.1	0.688 (326.8)	0.822 (347.2)	0.741 (335.7)	0.688 (385.6)
PredictionIO	379.5	0.672 (389.2)	0.818 (432.6)	0.682 (387.6)	0.741 (308.9)
Local	388.8	0.672 (411.9)	0.832 (401.8)	0.668 (419.4)	0.723 (322.1)
Microsoft	424.3	0.655 (477.3)	0.833 (391.9)	0.715 (370.5)	0.659 (457.6)

(a) Baseline performance.

Platform	Avg. Fried. Ranking	Avg. F-score	Avg. Accuracy	Avg. Precision	Avg. Recall
Local	190.1	0.839 (179.4)	0.916 (184.2)	0.984 (201.3)	0.990 (195.5)
Microsoft	211.1	0.837 (186.5)	0.914 (190.3)	0.954 (231.3)	0.863 (236.3)
PredictionIO	318.6	0.828 (245.7)	0.886 (238.7)	0.779 (478.4)	0.852 (311.5)
BigML	365.9	0.789 (307.5)	0.876 (281.7)	0.880 (287.9)	0.802 (351.4)
Amazon	446.7	0.761 (545.3)	0.863 (524.3)	0.826 (398.2)	0.795 (318.9)
Google	641.9	0.706 (692.6)	0.853 (606.7)	0.744 (605.5)	0.704 (662.9)
ABM	758.8	0.694 (784.3)	0.834 (774.1)	0.735 (747.7)	0.684 (729.1)

(b) Optimized performance.

Table 3.3: Baseline and optimized performance of MLaaS platforms. The Friedman ranking of each metric is included in the parenthesis. Lower Friedman ranking indicates consistently higher performance across all datasets.

3.4.2 Impact of Individual Controls

We have shown that higher complexity in the form of more user control contributes to higher optimized performance. Now we breakdown the potential performance gains from baseline configurations, and investigate the potential gains contributed by each type of control. In the collection of tunable controls and design decisions, answering this question would tell us which decisions have the most impact on the final performance. We start by tuning only one dimension of control while leaving others at baseline settings. Figure 3.5 shows the percentage improvement in performance from the baseline setting for each platform and control dimension. Note that Google and ABM are not included in this analysis. In addition, we have 3 platforms (Amazon, BigML, PredictionIO) missing in the Feature Selection column, one (Amazon) missing in the Classifier Selection column.

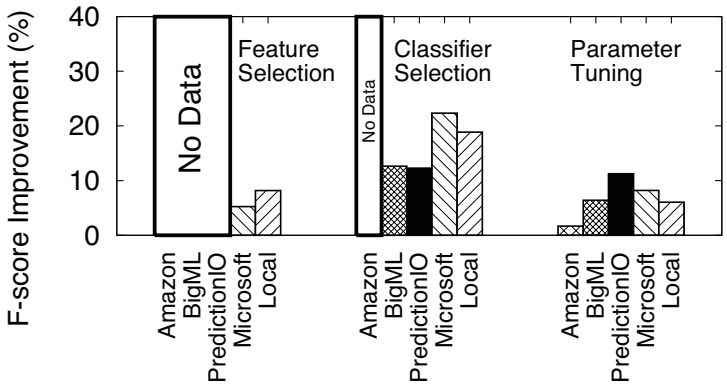


Figure 3.5: Relative improvement in performance (F-score) over baseline as we tune individual controls (white boxes indicate controls not supported).

These are the platforms that do not support tuning those respective control dimensions. We observe the largest performance improvement of 14.6% (averaged across all platforms) when giving users the ability to select specific ML classifiers. In fact, in the case of Microsoft, F-score improves by 22.4% which is the highest among all platforms when we optimize the classifier choice for each dataset. After the classifier dimension, feature selection provides the next highest improvement in F-score (6.1%) across all platforms, followed by the classifier parameter dimension (3.4% improvement in F-score). The above results show that classifier is the most important control dimension that significantly impacts the final performance. To shed light on the general performance of different classifiers, we analyze classifier performance with default parameters and with optimized parameter configurations. Table 3.4(a) shows the top 4 classifiers when using baseline (default) parameters. It is interesting to note that no single classifier dominates in terms of performance over all the datasets. Table 3.4(b) shows a similar trend even when we optimize the parameters. This suggests that we need a mix of multiple linear (*e.g.,* LR, NB) and non-linear (RF, BST, DT) classifiers to achieve high performance over all datasets.

Rank	BigML	PredictionIO	Microsoft	Local
1	LR (34.5%)	LR (42.9%)	BST (50.4%)	BST (24.4%)
2	RF (26.1%)	DT (38.7%)	AP (16.8%)	KNN (12.6%)
3	DT (24.4%)	NB (18.5%)	BPM (10.9%)	DT (10.9%)
4	BAG (15.1%)		RF (7.6%)	RF (10.9%)

(a) Ranking of classifiers using baseline parameters

Rank	BigML	PredictionIO	Microsoft	Local
1	RF (32.8%)	LR (48.7%)	BST (43.7%)	MLP (32.8%)
2	BAG (30.3%)	DT (36.1%)	DJ (17.6%)	BST (27.7%)
3	LR (27.7%)	NB (16.0%)	AP (16.0%)	RF (9.2%)
4	DT (9.2%)		RF (13.4%)	KNN (6.7%)

(b) Ranking of classifiers using optimized parameters

Table 3.4: Top four classifiers in each platform using baseline/optimized parameters. Number in parenthesis shows the percentage of datasets where the corresponding classifier achieved highest performance. LR=Logistic Regression, BST=Boosted Decision Trees, RF=Random Forests, DT=Decision Tree, AP=Average Perceptron, KNN=k-Nearest Neighbor, NB=Naive Bayes, BPM=Bayes Point Machine, BAG=Bagged Trees, MLP=Multi-layer Perceptron, DJ=Decision Jungle.

Summary. Our results clearly show that platforms with higher complexity (more dimensions for user control) achieve better performance. Among the 3 key dimensions, classifier choice provides the largest performance gain. Just by optimizing the classifier alone, we can already achieve close to optimized performance. Overall, Microsoft provides the highest performance across all platforms, and a highly tuned Microsoft model can produce performance identical to that of a highly-tuned local scikit-learn instance.

3.5 Risks of Increasing Complexity

Our experiments in Section 3.4 assumed that users were experts on each step in the ML pipeline, and were able to exhaustively search for the optimal classifier, parameters, and feature selection schemes to maximize performance. For example, for Microsoft, we evaluated over 17k configurations to determine the configuration with optimized performance. In practice, users may have less expertise, and are unlikely to experiment

with more than a small set of classifiers or available parameters. Therefore, our second question is: *Can increased control lead to higher risks (of building poorly performing ML models)?* To quantify risk of generating poorly performing models, we use performance variation as the metric, and compute variation on each platform as we tune available controls.

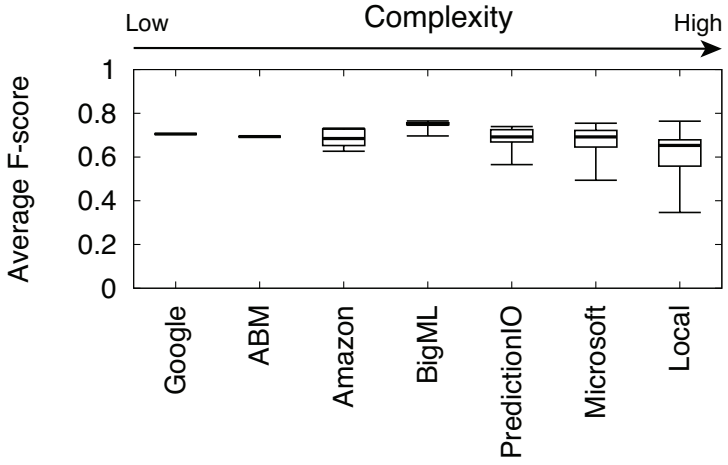


Figure 3.6: Performance variation in MLaaS platforms when tuning all available controls.

3.5.1 Performance Variation across Platforms

First we measure the performance variation of each MLaaS platform across a range of system configurations (of `CLF`, `PARA`, and `FEAT`) described in Section 3.3. For each configuration and platform, we compute average performance across all datasets. Then we iterate through all configurations, and obtain a range of performance scores which capture the performance variation. Each configuration would generate a single point in the range of performance scores. Higher variation means a single poor decision in design could produce a significant performance loss. We plot performance variation results for each platform in Figure 3.6. As before, platforms on the x-axis are ordered based on increasing complexity. First, we observe a positive correlation between complexity of an

MLaaS platform and higher performance variation. Among MLaaS platforms, Microsoft shows the largest variation, followed by less complex platforms like PredictionIO and Amazon. For Microsoft, F-score ranges from 0.49 to 0.75. Also as expected, our local ML library has the highest performance variation. The takeaway is that even though more complex platforms have the potential to achieve higher performance, there are higher risks of building a poorly configured (and poorly performing) ML model.

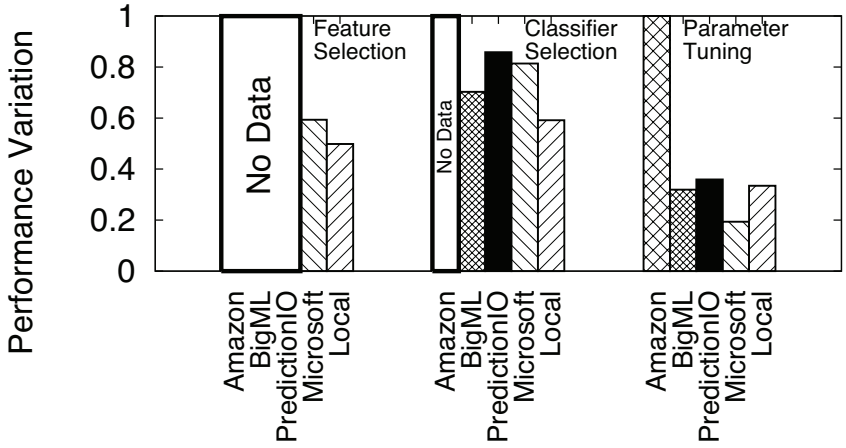


Figure 3.7: Performance variation when tuning `CLF`, `PARA` and `FEAT` individually, normalized by overall variation (white boxes indicate controls not supported).

3.5.2 Variation from Tuning Individual Controls

Next we analyze the contribution of each control dimension towards the variation in performance. When we tune a single dimension, we keep the other controls at their default values set by the platform, *i.e.* use the **baseline** settings. Figure 3.7 shows the *portion* of performance variation caused by each control dimension, *i.e.* a ratio normalized by the overall variation measured in our previous experiment. We observe that classifier choice (`CLF`) is the largest contributor to variation in performance. For example, in the case of Microsoft and PredictionIO (both exhibiting large variation), over 80% of the variation is captured by just tuning `CLF`. Thus, it is important to note that even though `CLF` can

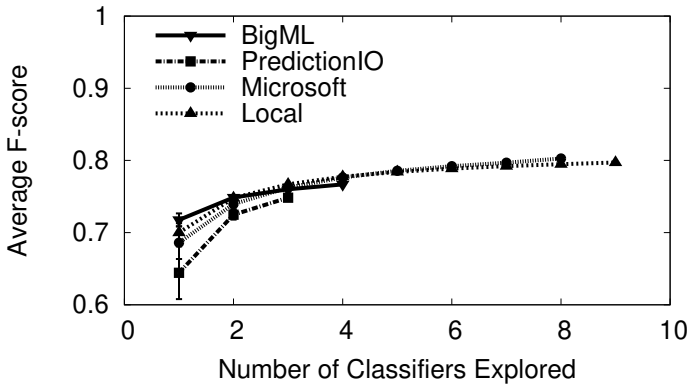


Figure 3.8: Average performance vs. number of classifiers explored.

provide the largest improvement in performance (Section 3.4), if not carefully chosen, can lead to significant performance degradation. On the other hand, for all platforms, except Amazon, tuning the `PARA` dimension results in the least variation in performance. We are unable to verify the reason for the high variation in the case of Amazon (for `PARA`), but suspect it is due to either implementation or default parameter settings.

Partial Knowledge about Classifiers. Given the disproportionally large impact classifier choice has on performance and performance variation, we want to understand how users can make better decisions without exhaustively experimenting over the entire gamut of ML classifiers. Instead, we simulate a scenario where the user experiments with (and chooses the best out of) a randomly chosen subset of k classifiers from all available classifiers in a platform. We measure the highest F-score possible in each k-classifier subset. Next, we average the highest F-score across all possible subsets of size k. Results are shown in Figure 3.8 with all platforms supporting classifier selection. We observe a trend of rapidly improving performance as users try multiple classifiers. We observe that just trying a randomly chosen subset of 3 classifiers often achieves performance that is close to the optimal found by experimenting with all classifiers. In the case of Microsoft, we observe an F-score of 0.76 which is only 5% lower than the F-score we can obtain

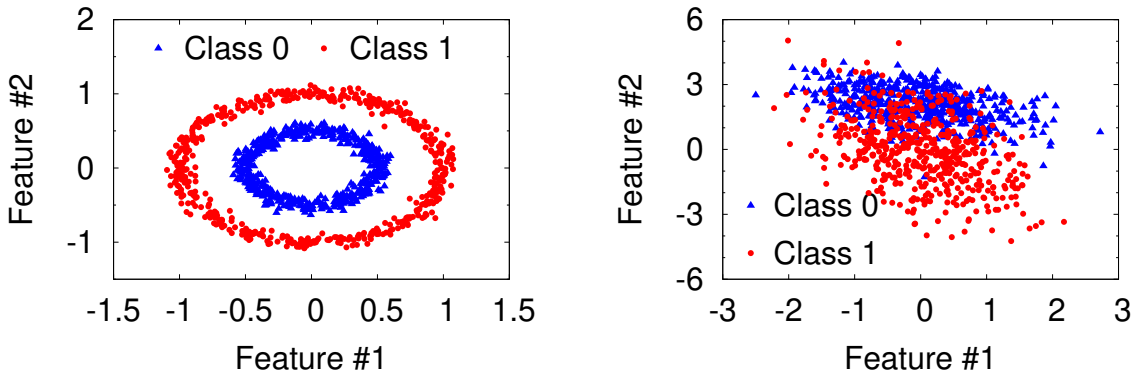
by trying all 8 classifiers. Performance variation also decreases significantly once a user explores 3 or more classifiers in these platforms.

Summary. Our results show that increasing platform complexity leads to better performance, but also leads to significant performance penalties for poor configuration decisions. Our results suggest that much/most of the gains can be achieved by focusing on classifier choice, and that experimenting with a random subset of 3 classifiers often achieves performance and lowers variation close to optimal.

3.6 Hidden Optimizations

In the final part of our analysis, we seek to shed light on any platform-specific optimizations outside of user-visible configurations or controls. More specifically, we focus on understanding hidden optimizations used by fully automated black-box platforms, Google and ABM. These platforms have the most leeway to implement internal optimizations, because their entire ML pipeline is fully automated. In Section 3.4.1 (Figure 3.4), we observe that Google and ABM outperform many other platforms when applying default configurations. This suggests that their hidden configurations are generally better than alternative default settings.

Among the countless potential options for optimization, we focus on a simple yet effective technique: optimizing classifier choices based on dataset characteristics [75]. We raise the question: *Are black-box platforms automatically selecting a classifier based on the dataset characteristics?* Note that our usage of the phrase "selecting a classifier" should be broadly interpreted as covering different possible implementation scenarios (for optimization). For example, optimization can be implemented by switching between distinct classifier instances, or a single classifier implementation that internally



(a) Visualization of CIRCLE.

(b) Visualization of LINEAR.

Figure 3.9: Visualization of two datasets: synthetic non-linearly-separable dataset (CIRCLE) and synthetic linearly-separable dataset (LINEAR).

alters decision characteristics depending on the dataset. While our analysis cannot infer such implementation details, we provide evidence of internal optimization in black-box platforms (Section 3.6.1). We further quantitatively analyze their optimization strategy (Section 3.6.2), and finally examine the potential for improvement (Section 3.6.3)

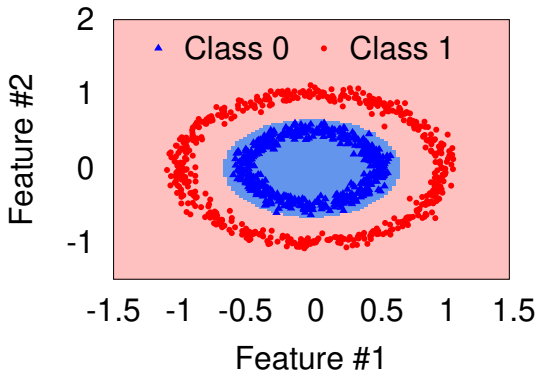
3.6.1 Evidence of Internal Optimizations

We select two datasets from our collection, a non-linearly-separable synthetic dataset, which we call CIRCLE [11], and a linearly-separable synthetic dataset, referred to as LINEAR [12]. Figure 3.9a and Figure 3.9b show visualizations of the two datasets. Both datasets have only two features. Given the contrasting characteristics (linearity) of the two datasets, our hypothesis is that they would help to differentiate between linear and non-linear classifier families based on prediction performance.

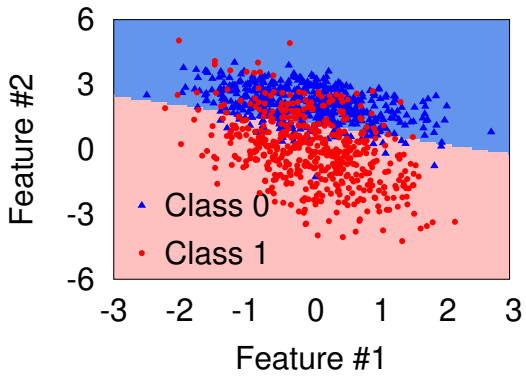
We examine Google and ABM's prediction results on CIRCLE and LINEAR to infer

[11] `http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html`

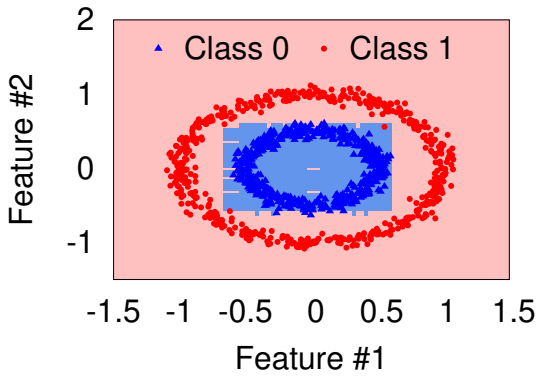
[12] `http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html`



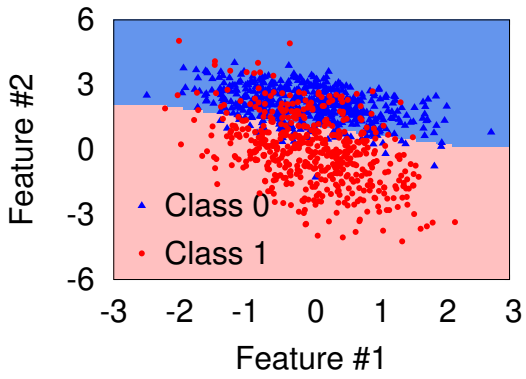
(a) Google's decision boundary on CIRCLE.



(b) Google's decision boundary on LINEAR.



(c) ABM's decision boundary on CIRCLE.



(d) ABM's decision boundary on LINEAR.

Figure 3.10: Decision boundaries generated by Google and ABM on CIRCLE and LINEAR. Both platforms produced linear and non-linear boundaries for different datasets.

their classifier choices. Since we have no ground-truth information here, we resort to analyzing decision boundaries generated by the two platforms. The decision boundary is visualized by querying and plotting the predicted classes of a 100×100 mesh grid. Figure 3.10a and Figure 3.10b illustrate Google's decision boundary on CIRCLE and LINEAR, respectively. It is very clear that Google's decision boundary on CIRCLE forms a circle, indicating Google is using a non-linear classifier, or a non-linear kernel, *e.g.,* RBF kernel [76]. On LINEAR, Google's decision boundary matches a straight line. It shows Google is using a linear classifier. Experiments on ABM also show similar re-

Category	Classifiers
Linear	LR, NB, Linear SVM, LDA
Non-Linear	DT, RF, BST, KNN, BAG, MLP

Table 3.5: Assignment of classifiers available on local library into linear vs. non-linear categories.

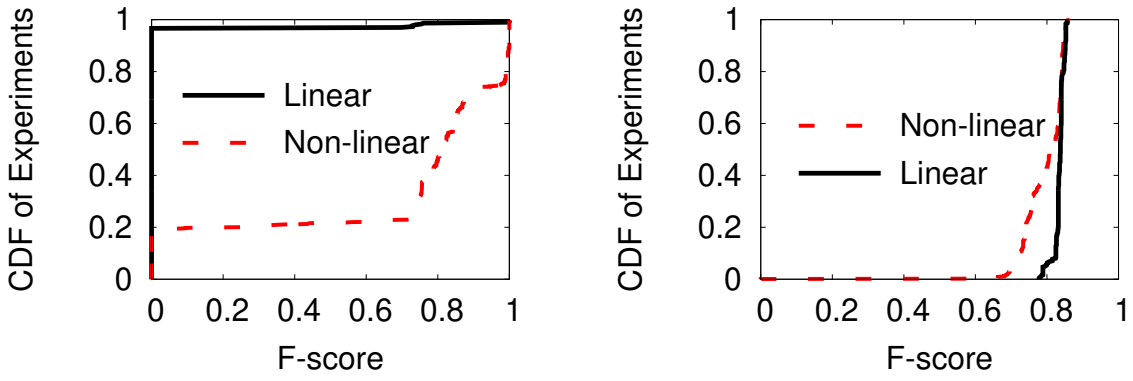
sults. Figure 3.10c and Figure 3.10d show the decision boundaries of ABM on CIRCLE and LINEAR, respectively. Thus, both platforms are optimizing and switching classifier choices for the two datasets. Additionally, Google's decision boundary on CIRCLE (circular shape) is different from ABM (rectangular shape), indicating that they selected different non-linear classifiers. Based on the shape of decision boundaries, it is likely that Google used a non-linear kernel based classifier while ABM chose a tree-based classifier.

3.6.2 Predicting Classifier Family

In this section, we present a method to automatically predict the classifier used by a platform using just two pieces of information—knowledge of the training dataset and prediction results from the platform. Such a method would help us automatically find instances where a black-box platform would change classifiers depending on the dataset characteristics.

At a high level, we observe that it is hard to pin-point the specific classifier used by a platform, using just the dataset and the prediction results. This is because prediction results of different classifiers tend to overlap. However, we find that it is possible to accurately infer the broad *classifier family*, more specifically, *linear* or *non-linear* classifiers.

Our key insight is that we can control datasets used for the inference and thus selectively choose datasets that elicit significant divergence between prediction results of linear and non-linear classifiers. To give an example, we examine the performance of the local library classifiers on the CIRCLE and LINEAR datasets. We categorize local clas-



(a) CIRCLE. (b) LINEAR.

Figure 3.11: Performance of predicting local linear/non-linear classifier choices on CIRCLE and LINEAR datasets.

sifiers into linear and non-linear families, as shown in Table 3.5. Figures 3.11a and 3.11b shows the performance (F-score) of the two categories of classifiers on the two datasets. As expected, we find that linear and non-linear classifiers produce very different F-scores on the two datasets, regardless of other configuration settings. Non-linear classifiers outperform linear classifiers when on CIRCLE. For LINEAR, linear classifiers outperform non-linear classifiers in many cases. This is because of the noisy nature of the dataset causing non-linear classifiers to overfit, and therefore produce lower performance compared to linear classifiers. Next, we present our methodology to accurately predict the classifier family by identifying more datasets that show divergence in prediction results of linear and non-linear classifiers.

Methodology. We build a supervised ML classifier for the prediction task. For training the classifier, we use prediction results, and ground-truth of classifier choices from the local library and the three platforms that allow user control of the classifier dimension (i.e. Microsoft, BigML and PredictionIO). Features used for training include aggregated performance metrics (F-score, precision, recall, accuracy), and the predicted

labels. We train one classifier for each dataset in our collection. Each training sample is one ML experiment using a single configuration of the ML pipeline (*i.e.* choices of `FEAT`, `CLF` and `PARA`). Measurements are randomly split into training, validation, and test sets. Training and validation sets contain 70% of experiments, and test set contains the remaining 30% experiments. We train a Random Forests classifier with 5-fold cross-validation, and pick the best performing classifier based on validation performance. Based on prior work, Random Forests is one of the best performing classifiers for supervised binary classification tasks [62, 72]. Figure 3.12 shows the distribution of cross-validation performance of classifiers trained on all 119 datasets. Not all datasets could differentiate linear and non-linear classifiers. There are 64 datasets that produce classifiers achieving higher than 0.95 F-score. In other datasets, classifiers failed to separate linear and non-linear classifiers as they produce similar performance. Intuitively, we do not expect all datasets to perform well, and one goal of the training process is to identify datasets with high differentiating power. We select the 64 datasets where the trained classifiers achieve high performance (F-score > 0.95) on the validation set. To further test if they would generalize and accurately predict classifier choices, we apply them on the 30% held-out test set. All trained classifiers achieve F-score higher than 0.96. This further proves that the chosen classifiers can accurately predict the classifier family.

Classifier Choices of Google and ABM. We apply the selected 64 trained classifiers (covering 64 datasets) on Google and ABM and predict their classifier choices over linear and non-linear family. Results show Google uses linear classifiers on 39 out of 64 (60.9%) datasets, and non-linear classifiers for the remaining 25 (39.1%) datasets. ABM, on the other hand, uses linear classifiers on 44 (68.8%) out of 64 datasets, and non-linear on the remaining 20 (31.2%) datasets. If we compare Google and ABM, they pick the same classifier category on 49 (76.6%) datasets, but disagree on the remaining 15

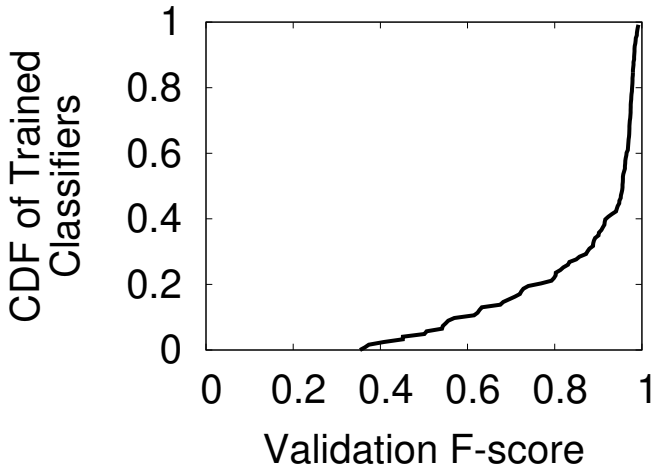


Figure 3.12: Validation performance of predicting linear/non-linear classifiers.

(23.4%) datasets. The differences in the classifier choices could contribute to their overall performance difference in Figure 3.4. Overall, our results suggest that both platforms make different classifier choices (choosing linear or non-linear family) depending on the dataset.

Classifier Choices of Amazon. Recall that Amazon does not reveal any classifier information in their model training interface, but claims to use Logistic Regression on their documentation page. We apply our classifier prediction scheme on Amazon to investigate whether they are indeed using a single classifier for all tasks. Interestingly, 10 out of all 64 datasets have over 50% configurations that are predicted to be non-linear (the remaining are predicted to be linear). We also observe that Amazon produces a non-linear decision boundary when applied to the CIRCLE dataset (Figure 3.13). We suspect that Amazon uses non-linear techniques on top of Logistic Regression, *e.g.,* non-linear kernel, or even uses other non-linear classifiers apart from Logistic Regression.

Unfortunately we are unable to corroborate our findings with the providers (Google, ABM, and Amazon), as all hidden optimizations are kept confidential and proprietary.

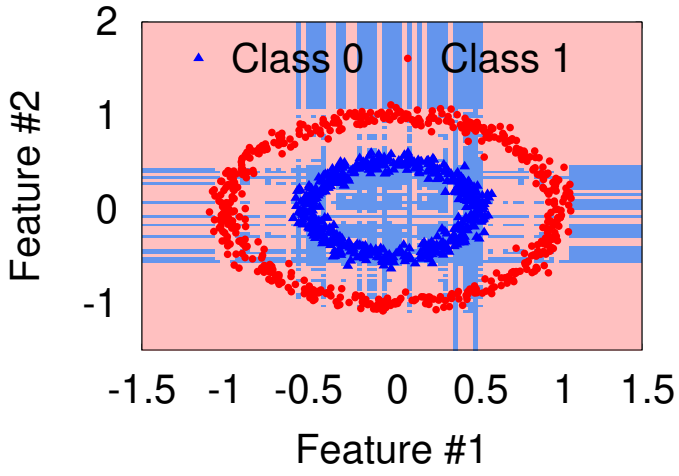


Figure 3.13: Amazon's decision boundary on CIRCLE.

However, our predictions demonstrate high accuracy in our validation and test datasets (where the underlying configuration is known).

3.6.3 Impact of Internal Optimizations

Previous experiments show that black-box platforms successfully choose classifier families with better performance when applied on the CIRCLE and LINEAR datasets. On these two datasets, Google and ABM would outperform a scheme that does not switch between classifier families. But are their strategies optimized for all other datasets? Are there cases where the two platforms make the wrong classifier choice?

To understand the potential for further improvement, we design a naïve classifier selection strategy using the local library, and compare its performance with Google and ABM. Our intuition is that if Google and ABM perform poorly when compared to our naïve strategy, there is potential for further improvement and we can understand the cases where classifier choices (*i.e.* linear vs non-linear) are potentially incorrect. We choose two widely used linear and non-linear classifiers, Logistic Regression and Decision Tree. These two classifiers are supported by most other platforms (Table 3.1). For

Naïve \ Google	Linear	Non-linear
Linear	11 (25.5%)	5 (11.6%)
Non-linear	17 (39.5%)	10 (23.26%)

(a) Google vs. our naïve strategy.

Naïve \ ABM	Linear	Non-linear
Linear	8 (16.7%)	3 (6.3%)
Non-linear	22 (45.8%)	15 (31.3%)

(b) ABM vs. our naïve strategy.

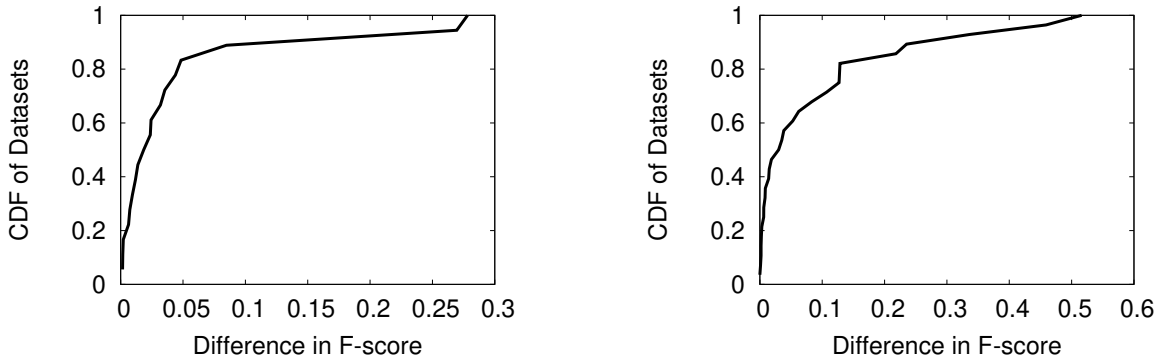
Table 3.6: Breakdown of datasets based on classifier choice when our naïve strategy outperforms black-box platforms.

each dataset, we train both classifiers and choose the one with higher performance. To further simplify the strategy and to avoid any impact of optimization from other control dimensions, we use the default parameter settings in Logistic Regression and Decision Tree, and perform no feature selection.

For our analysis, we again use the 64 datasets that can accurately predict choices of linear and non-linear classifiers. In 43 out of 64 datasets, our naïve strategy outperforms Google, and in 48 datasets, it outperforms ABM. This clearly indicates that Google and ABM have scope for further improvement.

We further compare the choices made by naïve strategy and black-box platforms. Table 3.6 shows the breakdown of the datasets by decisions when naïve strategy outperforms Google and ABM. In both platforms, in a majority of cases, the classifier choices do not match our simple strategy. In these cases, Google and ABM could increase their performance (F-score) by 20% and 34% on average, respectively, by choosing the other classifier family. Figure 3.14 shows a detailed breakdown (as a CDF) of performance difference between the black-box platforms and naïve strategy, when we outperform them. The potential performance improvement is significant in many cases.

When is switching classifier the best option for improvement? Although we show the potential performance improvement by switching classifiers, black-box platforms could use other methods to improve performance. For example, Google and ABM could



(a) Google. (b) ABM.

Figure 3.14: Performance difference in datasets where naïve strategy outperforms Google/ABM using different classifier family.

perform better parameter tuning and feature selection to reduce the performance gap and justify their classifier choices. To identify cases where classifier switching is likely the best option, we compare our naïve strategy with the *optimal* performance of the other classifier family (*i.e.* not chosen). This means that when naïve strategy chooses a non-linear classifier, we compare its performance with the optimal linear classifier (across all configurations). If naïve strategy could still outperform Google and ABM under this scenario, it indicates that switching the classifier is likely the best way to further improve the performance. We find 3 datasets in the case of Google, and 4 for ABM, where changing the classifier is probably the best option to further improve performance. While our analysis is limited to the 64 datasets where we can perform prediction, our finding highlights the existence of scenarios where Google and ABM clearly need to make better classifier choices.

3.7 Related Work

Analyzing MLaaS Platforms. There is very limited prior work focusing on MLaaS platforms. Chan, *et al.* and Ribeiro, *et al.* presented two different architecture designs of MLaaS platforms [77, 78]. Although we cannot confirm that these architectures are being used by any of the MLaaS platforms we studied, they shed light on potential ways MLaaS platforms operate internally. Other researchers investigated vulnerabilities of these platforms towards different types of attacks. This includes attacks that try to leak information about individual training records of a model [79, 80], and those aiming to duplicate the functionality of a black-box MLaaS model by querying the model [81]. While these studies are in general orthogonal to our work, there is scope for borrowing techniques from them that can help us better understand MLaaS platforms.

Empirical Analysis of ML Classifiers. Prior empirical analysis focused on determining the best classifier for a variety of ML problems using user-managed ML tools (*e.g.,* Weka, R, Matlab) on a large number of datasets. Multiple studies conducted an exhaustive performance evaluation of up to 179 supervised classifiers using up to 121 datasets [62, 72, 82]. All studies observe that Random Forests, Boosted or Bagging Trees outperform other classifiers, including SVM, Naïve Bayes, Logistic Regression, Decision Tree, and Multi-layer Perceptron. Caruana *et al.* further studied classifier performance focusing on high-dimensional datasets [73]. They find that Random Forests perform better than Boosted Trees on high-dimensional data, and the relative performance difference between classifiers change as dimensionality increases. Other work also focused on evaluating performance of specific classifier families, for example tree-based classifiers [83, 84], rule-based classifiers [84], and ensemble methods [85].

In comparison, our work does not focus on a single step of the ML pipeline. Instead,

we analyze end-to-end impact of complexity on classifier performance, through the lens of deployed MLaaS platforms. This allows us to understand how specific changes to the ML task pipeline impact actual performance in a real world system. Instead of focusing only on the best achievable performance for any classifier, we recognize the wide-spread use of ML by generalist users, and study the "cost" of suboptimal decisions in choosing and configuring classifiers in terms of degraded performance.

Automated Machine Learning. Many works focused on reducing human effort in ML system design by automating classifier selection and parameter tuning. Researchers proposed mechanisms to recommend classifier choices based on classifiers that are known to perform well on similar datasets [86]. Many mechanisms even use machine learning algorithms like collaborative filtering and k-nearest neighbor to recommend classifiers [87, 88, 89]. To perform automatic parameter optimization, methods have been proposed based on intuition-based Random Search [90, 91], and Bayesian optimization [92, 90, 93, 94]. These mechanisms have been shown to estimate suitable parameters with less computational complexity than brute-force methods like Grid Search [95]. Other works proposed techniques to automate the entire ML pipeline. For example, Auto-Weka [96, 97] and Auto-Sklearn [98] could search through the joint space of classifiers and their respective parameter settings and choose the optimal configuration.

Experimental Design for Evaluating ML Classifiers. The ML community has a long history on classifier evaluation using carefully designed benchmark tests [99, 100, 101]. Many studies proposed theoretical frameworks and guidelines for designing benchmark experiments [102, 103]. Dietterich used statistical tests to compare classifiers [104] and the methodology was later improved in follow-up work [105, 106]. Our performance evaluation using Friedman ranking is based on their methodology.

Other work focused on comparing and benchmarking performance of popular ML software [107, 108], *e.g.,* Weka [109], PRTools [110], KEEL [111] and, more recently, deep learning tools [112]. In addition, work has been done to identify and quantify the relationship between classifier performance and dataset properties [113, 75], especially dataset complexity [114, 115, 116]. Our work leverages similar insights about dataset complexity (linearity) to automatically identify classifier families based on prediction results.

3.8 Limitations

We point out three limitations of our study. *First*, we focus on 6 mainstream MLaaS platforms, covering services provided by both traditional Internet giants (Google, Microsoft, Amazon) and emerging startups (ABM. BigML, PredictionIO). We did not study other commercial MLaaS platforms because they either focus on highly specialized tasks (*e.g.,* image/text classification), or does not support large scale measurements (*e.g.,* posing strict rate limit). *Second*, we focus on binary classification tasks with three dimensions of control (`CLF`, `PARA`, and `FEAT`). We did not extend our analysis to other ML tasks and cover every configuration choice, *e.g.,* more advanced classifiers. We leave these as future work. *Third*, we only study the classification performance of MLaaS platforms, which is one of the many aspects to evaluate MLaaS platforms. There are other dimensions, *e.g.,* training time, cost, robustness to incorrect input. We leave further exploration of these aspects as future work.

3.9 Conclusions

For network researchers, MLaaS systems provide an attractive alternative to running and configuring their own standalone ML classifiers. Our study empirically analyzes

the performance of MLaaS platforms, with a focus on understanding how user control impacts both the performance and performance variance of classification in common ML tasks.

Our study produced multiple key takeaways. First, as expected, with more control comes more potential performance gains, as well as greater performance degradation from poor configuration decisions. Second, fully automated platforms are optimizing classifiers using internal tests. While this greatly simplifies the ML process and helps them outperform other MLaaS platforms using default settings, their aggregated performance lags far behind well-tuned versions of more configurable alternatives (Microsoft, PredictionIO, local scikit-learn). Finally, much of the gains from configuration and tuning come from choosing the right classifier. Experimenting with a small random subset of classifiers is likely to achieve near-optimal results.

Our study shows that used correctly, MLaaS systems can provide networking researchers results comparable to standalone ML classifiers. While more automated "turnkey" systems are making some intelligent decisions on classifiers, they still have a long way to go. Thankfully, we show that for most classification tasks today, experimenting with a small random subset of classifiers will produce near-optimal results.

Chapter 4

Practical Attacks against Transfer Learning

Transfer learning is a powerful approach that allows users to quickly build accurate deep-learning (Student) models by "learning" from centralized (Teacher) models pretrained with large datasets, *e.g.* Google's InceptionV3. We hypothesize that the centralization of model training increases their vulnerability to misclassification attacks leveraging knowledge of publicly accessible Teacher models. In this paper, we describe our efforts to understand and experimentally validate such attacks in the context of image recognition. We identify techniques that allow attackers to associate Student models with their Teacher counterparts, and launch highly effective misclassification attacks on black-box Student models. We validate this on widely used Teacher models in the wild. Finally, we propose and evaluate multiple approaches for defense, including a neuron-distance technique that successfully defends against these attacks while also obfuscates the link between Teacher and Student models.

4.1 Introduction

Deep learning using neural networks has transformed computing as we know it. From image and face recognition, to self-driving cars, knowledge extraction and retrieval, and natural language processing and translation, deep learning has produced game-changing applications in every field it has touched.

While advances in deep learning seem to arrive on a daily basis, one constraint has remained: deep learning can only build accurate models by training using large datasets. This thirst for data severely constrains the number of different models that can be independently trained. In addition, the process of training large, accurate models (often with millions of parameters) requires computational resources that can be prohibitive for individuals or small companies. For example, Google's InceptionV3 model is based on a sophisticated architecture with 48 layers, trained on ∼1.28M labeled images over a period of 2 weeks on 8 GPUs.

The prevailing consensus is to address the data and training resource problem using *transfer learning*, where a small number of highly tuned and complex centralized models are shared with the general community, and individual users or companies further customize the model for a given application with additional training. By using the pretrained *teacher* model as a launching point, users can generate accurate *student* models for their application using only limited training on their smaller domain-specific datasets. Today, transfer learning is recommended by most major deep learning frameworks, including Google Cloud ML, Microsoft Cognitive Toolkit, and PyTorch from Facebook.

Despite its appeal as a solution to the data scarcity problem, the centralized nature of transfer learning creates a more attractive and vulnerable target for attackers. Lack of diversity has amplified the power of targeted attacks in other contexts, *i.e.* increasing the impact of targeted attacks on network hubs [31], supernodes in overlay networks [32],

and the impact of software vulnerabilities in popular libraries [33, 34].

In this paper, we study the possible negative implications of deriving models from a small number of centralized teacher models. Our hypothesis is that boundary conditions that can be discovered in the white box teacher models can be used to perform targeted misclassification attacks against its associated student models, even if the student models themselves are closed, *i.e.* black-box. Through detailed experimentation and testing, we find that this vulnerability does in fact exist in a variety of the most popular image classification contexts, including facial and iris recognition, and the identification of traffic signs and flowers. Unlike prior work on black-box adversarial attacks, this attack does not require repeated queries of the student model, and can instead prepare the attack image based on knowledge of the teacher model and any target image(s).

This paper describes several key contributions:

- We identify and extensively evaluate the practicality of misclassification attacks against student models in multiple transfer-learning applications.
- We identify techniques to reliably identify teacher models given a student model, and show its effectiveness using known student models in the wild.
- We perform tests to evaluate and confirm the effectiveness of these attacks on popular deep learning frameworks, including Google Cloud ML, Microsoft Cognitive Toolkit (CNTK), and the PyTorch open source framework initially developed by Facebook.
- We explore and develop multiple defense techniques against attacks on transfer learning models, including defenses that alter the student model training process, that alter inputs prior to classification, and techniques that introduce redundancy using multiple models.

Transfer learning is a powerful approach that addresses one of the fundamental challenges facing the widespread deployment of deep learning. To the best of our knowledge, our work is the first to extensively study the inheritance of vulnerabilities between transfer learning models. Our goal is to bring attention to fundamental weaknesses in these models, and to advocate for the evaluation and adoption of defensive measures against adversarial attacks in the future.

4.2 Background

We begin by providing some background information on transfer learning and adversarial attacks on deep learning frameworks.

4.2.1 Transfer Learning

The high level idea of transfer learning is to transfer the "knowledge" from a pre-trained *Teacher* model to a new *Student* model, where the student model's task shares significant similarity to the teacher model's. This "knowledge" typically includes the model architecture and weights associated with the layers. Transfer learning enables organizations without access to massive datasets or GPU clusters to quickly build accurate models customized to their application context.

How Transfer Learning Works. Figure 4.1 illustrates transfer learning at a high level. The student model is initialized by copying the first $N-1$ layers of the Teacher. A new dense layer is added for classification. Its size matches the number of classes in the student task. Then the student model is trained using its own dataset, while the first K layers are "frozen", *i.e.* their weights are fixed, and only weights in the last $N-K$ layers are updated.

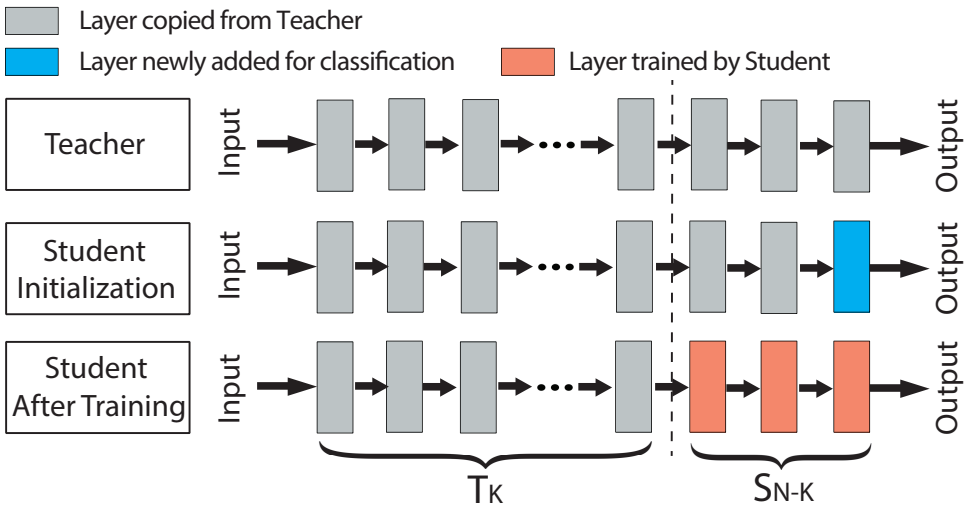


Figure 4.1: Transfer learning. A student model is initialized by copying the first N-1 layers from a teacher model, with a new dense layer added for classification. The model is further trained by only updating the last N-K layers.

The first K layers (referred to as *shallow layers*) are frozen during training because outputs of those layers already represent meaningful features for the student task. The student model can reuse these features directly, and freezing them lowers both training cost and amount of required training data.

Based on the number of layers being frozen (K) during the training process, transfer learning is categorized into the following three approaches.

- *Deep-layer Feature Extractor:* $N-1$ layers are frozen during training, and only the last classification layer is updated. This is preferred when the student task is very similar to the teacher task, and requires minimal training cost (the cost of training a single-layer DNN).

- *Mid-layer Feature Extractor:* The first K layers are frozen, where $K < N-1$. Allowing more layers to be updated helps the student perform more optimization for its own task. *Mid-layer Feature Extractor* typically outperforms *Deep-layer Feature Extractor* in scenarios where the student task is more dissimilar to the

teacher task, and more training data is available.

- *Full Model Fine-tuning:* All layers are unfrozen and fine-tuned during student training ($K = 0$). This requires more training data, and is appropriate when the student task differs significantly from the teacher task. Bootstrapping using pre-trained model weights helps the student converge faster and potentially achieve better performance than training from scratch [117].

We run a simple experiment to demonstrate the impact of transfer learning. We target facial recognition, where the student task is to recognize a set of 65 faces, and uses a well-performing face recognition model called VGG-Face [118] as teacher model. Using only 10 images per class to train the student model, we achieve 93.47% classification accuracy. Training the student with the same architecture but with random weights (no pre-trained weights) produces accuracy close to random guessing.

4.2.2 Adversarial Attacks in Deep Learning

The goal of adversarial attacks against deep learning networks is to modify input images so that they are misclassified in the DNN. Given a source image, the attacker applies a small perturbation so that it is misclassified by the victim DNN into either a specific target class, or any class other than the real class. Existing attacks fall into two categories, based on their assumptions on how much information attacker has about the classifier.

White-box Attacks. These attacks assume the attacker knows the full internals of the classifier DNN, including its architecture and all weights. It allows the attacker to run unlimited queries on the model, until a success adversarial sample is found [25, 119, 120, 121, 122]. These attacks often achieve close to 100% success with minimal

perturbations, since full access to the DNN allows them to find the minimal amount of perturbations required for misclassification. The white-box scenario is often considered impractical, however, since few systems reveal internals of their model publicly.

Black-box Attacks. Here attackers do not have knowledge of the internals of the victim DNN, *i.e.* it remains a black-box. The attacker is allowed to query the victim model as an Oracle [123, 122]. Most black-box attacks either use queries to test intermediate adversarial samples and improve iteratively [122], or try to reverse-engineer decision boundaries of the DNN and build a replica, which can be used to craft adversarial samples [123]. Black-box attacks often achieve lower success than white-box attacks, and require a large number of queries to the target classifier [122].

Adversarial attacks can also be categorized into *targeted* and *non-targeted* attacks. A targeted attack aims to misclassify the adversarial image into a specific target class, whereas a non-targeted attack focuses on triggering misclassification into any class other than the real class. We consider and evaluate both targeted and non-targeted attacks in this paper.

4.3 Attacks on Transfer Learning

Here, we describe our attack on transfer learning, beginning with the attack model.

Attack Model. In the context of our definitions in Section 4.2, our attack assumes white-box access to teacher models (consistent with common practice today) and black-box access to student models. We consider a given attacker looking to trigger a misclassification from a Student model S, which has been customized through transfer learning from a Teacher model T.

- *White-box Teacher Model.* We assume that T is a white-box, meaning the attacker knows its model architecture and weights. Most or all popular models today have been made publicly available to increase adoption. Even if Teacher models became proprietary in the future, an attacker targeting a single Teacher model could obtain it by posing as a Student to gain access to the Teacher model.

- *Black-box Student Model.* We assume S is black-box, and all weights remain hidden from the attacker. We also assume the attacker does not know the Student training dataset, and can use only limited queries (*e.g.,* 1) to S. Apart from a single adversarial sample to trigger misclassification, we expect no additional queries to be made during the pre-attack process.

- *Transfer Learning Parameters.* We assume the attacker knows that S was trained using T as a Teacher, and which layers were frozen during the Student training. This information is not hard to learn, as many service providers, *e.g.,* Google Cloud ML, release such information in their official tutorials. We further relax this assumption in Sections 4.4 and 4.5, and consider scenarios where such information is unknown. We will discuss the impact on performance, and propose techniques to extract such information from the Student using a few additional queries.

Insight and Attack Methodology. Figure 4.2 illustrates the main idea behind our attack. Consider the scenario where the attacker knows that the first K layers of the Student model are copied from the Teacher and frozen during training. Attacker perturbs the source image so it could be misclassified as the same class of a specific target image. Using the Teacher model, *attacker computes perturbations that mimic the internal representation of the target image at layer K.* Internal representation is captured by passing the target image as input to the Teacher, and using the values of

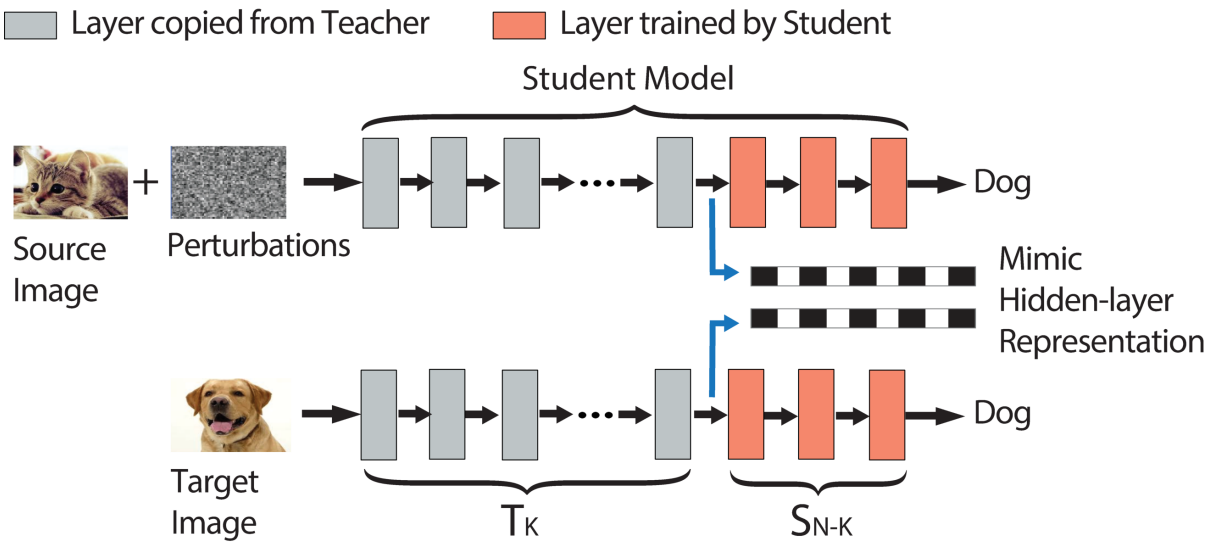


Figure 4.2: Illustration of our attack. Given images of a cat and a dog, attacker computes perturbations that mimic the internal representation of the dog image at layer K. If the calculations are perfect, the adversarial sample will be classified as dog, regardless of unknown layers in S_{N-K}.

the corresponding neuron outputs at layer K.

Our key insight: is that (in feedforward networks) since each layer can only observe what is passed on from the previous layer, if our adversarial sample's internal representation at layer K perfectly matches that of the target image, it must be misclassified into the same label as the target image, regardless of the weights of any layers that follow K.

This means that in the common case of feature extractor training, if we can mimic a target in the Teacher model, then misclassification will occur regardless of how much the Student model trains with local data. We also note that some models like InceptionV3 and ResNet50, where "shortcut" layers can skip several layers, are not strictly feedforward. However, the same principle applies, because a block (consisting of several layers) only takes information from the previous block. Finally, it is hard in practice to perfectly mimic the internal representation, since we are limited in our level of possible perturbation, in order to keep adversarial changes indistinguishable by humans. The attacker's goal, therefore, is to minimize the dissimilarity between internal representations, given a

fixed level of perturbation.

Targeted vs. Non-targeted Attacks. We consider both targeted and non-targeted attacks. The goal in targeted attacks is to misclassify a source image x_s into the class of a target image x_t. The attacker focuses on a specific layer K of the Teacher model, and tries to mimic the target image's internal representation (neuron values) at layer K. Let $T_K(.)$ be the function (associated with Teacher) transforming an input image to the internal representation at layer K. A perturbation budget P is used to control the amount of perturbation added to the source image. The following optimization problem is solved to craft an adversarial sample x'_s.

$$\begin{aligned} \min \quad & D(T_K(x'_s), T_K(x_t)) \\ \text{s.t.} \quad & d(x'_s, x_s) < P \end{aligned} \tag{4.1}$$

The above optimization tries to minimize dissimilarity $D(.)$ between the two internal representations, under a constraint to limit perturbation within a budget P. We use L_2 distance to compute $D(.)$. $d(x', x_s)$ is a distance function measuring the amount of perturbation added to x_s. We discuss $d(.)$ later in this section.

In non-targeted attacks, the goal is to misclassify x_s into any class different from the source class. To do this, we need to identify a "direction" to push the source image outside its decision boundary. In our case, it is hard to estimate such a direction without having a target image in hand, as we rely on mimicking hidden representations. Therefore, we perform a non-targeted attack by evaluating multiple targeted attacks, and choose the one that achieves the minimum dissimilarity between the internal representations. We assume that the attacker has access to a set of target images I (each belonging to a distinct class). Note that the source image can be misclassified to even classes outside the set I. The set of images I merely serves as a guide for the optimization process.

Empirically, we find that even small sizes of set I (just 5 images) can achieve high attack success. The optimization problem is formulated as follows.

$$\begin{aligned} \min \quad & min_{i \in I}\{D(T_K(x'_s), T_K(x_{ti}))\} \\ \text{s.t.} \quad & d(x'_s, x_s) < P \end{aligned} \tag{4.2}$$

Measuring Adversarial Perturbations. As mentioned before, $d(x'_s, x_s)$ is the distance function used to measure the amount of perturbation added to the image. Most prior work used the L_p distance family, *e.g.,* L_0, L_2, and L_∞ [25]. While a helpful way to quantify perturbation, L_p distance fails to capture what humans perceive as image distortion. Therefore, we use another metric, called $DSSIM$, which is an objective image quality assessment metric that closely matches with the perceived quality of an image (*i.e.* subjective assessment) [124, 125]. The key idea is that humans are sensitive to *structural* changes in an image, which strongly correlates with their subjective evaluation of image quality. To infer structural changes, $DSSIM$ captures patterns in pixel intensities, especially among neighboring pixels. The metric also captures luminance, and contrast measures of an image, that would also impact perceived image quality. $DSSIM$ values fall in the range $[0, 1]$, where 0 means the image is identical to the original image, and a higher value means the perceived distortion will be higher. We include the mathematical formulation of $DSSIM$ in the Appendix. We also refer interested readers to the original papers for more details [124, 125].

Solving the Optimization Function. To solve the optimization in Equation 4.1, we use the *penalty method* [126] to reformulate the optimization as follows.

$$\min \quad D(T_K(x'_s), T_K(x_t)) + \lambda \cdot (\max\left(d(x'_s, x_s) - P,\ 0\right))^2$$

Here λ is the penalty coefficient that controls the tightness of the privacy budget constraint. By gradually increasing λ, the final optimization result would converge to that of the original formulation. In our experiment, we empirically choose a λ large enough to ensure the perturbation constraint is tightly enforced.

We use Adadelta [127] optimizer to solve the re-formulated optimization problem. To constrain input pixel intensity within the correct range ($[0, 255]$), we transform intensity values into *tanh* space [25].

4.4 Experimental Results

Next, we perform experiments across a number of classification tasks to validate the effectiveness of attacks on transfer learning. Given their wide adoption in a variety of applications, we focus on image classification tasks, including facial recognition, iris recognition, traffic sign recognition and flower recognition.

4.4.1 Experimental Setup

Teacher and Student Models. We use four tasks and their associated Teacher models and datasets to build our victim Student models.

- **Face** Recognition classifies an image of a human face into a class associated with a unique individual. The Teacher is the popular 16 layer VGG-Face model [128] trained on a dataset of 2.6M images to recognize $2,622$ faces. The Student model is trained using the PubFig dataset [129] to recognize a different set of 65 individuals [1]. The Student training dataset contains 90 faces belonging to each of the 65. The

[1]The original dataset contains 83 celebrities. We exclude 18 celebrities that were also used in the Teacher model.

testing dataset for the Student model contains 650 images (10 images per class).

- **Iris** Recognition classifies an image of a human iris into one of many classes associated with different individuals. The Teacher model is a 16 layer VGG16 model trained on the ImageNet dataset of 1.3M images [8]. The Student model is trained on the CASIA IRIS dataset [130] containing $16,000$ iris images associated with $1,000$ individuals, and the testing dataset contains $4,000$ images.

- **Traffic Sign** Recognition classifies different types of traffic signs from images, which can be used by self-driving cars to automatically recognize traffic signs. The Teacher model is again the 16 layers VGG16, trained on the ImageNet dataset. The Student is trained using the GTSRB dataset [131] containing $39,209$ images of 43 different traffic signs. It also has a testing dataset of $12,630$ images.

- **Flower** Recognition classifies images of flowers into different categories, and is a popular example of multi-class classification. It is also an example of transfer learning by Microsoft's CNTK framework [132]. The Teacher model is the ResNet50 model (with 50 layers) [6], trained on the ImageNet dataset. The Student is trained on the VGG Flowers dataset [133] containing $6,149$ images from 102 classes, and comes with a testing dataset of $1,020$ images.

These tasks represent typical scenarios users may face during transfer learning. First, the training dataset for building the Student model is significantly smaller than that of the Teacher's training dataset, which is a common scenario for transfer learning. Second, the Teacher and Student models either target similar tasks (Face Recognition) or very different tasks (Flowers and Traffic Sign Recognition). Finally, Face, Iris and Traffic sign recognition are security-related tasks. More details of training the Student models are listed in Table A.1 in the Appendix.

Student Task	Transfer Process		
	Deep-layer Feature Extractor	*Mid-layer Feature Extractor*	*Full Model Fine-tuning*
Face	**98.55**%	98.00% (14/16)	75.85%
Iris	**88.27**%	88.22% (14/16)	81.72%
Traffic Sign	62.51%	**96.16**% (10/16)	94.39%
Flower	43.63%	92.45% (10/50)	**95.59**%

Table 4.1: Transfer learning performance for different tasks when using different transfer processes. For each task, we select the model with the highest accuracy as our target Student model in future analysis. Numbers in parenthesis under *Mid-layer Feature Extractor* are the number of layers copied to achieve the corresponding accuracy, as well as the total number of layers of the Teacher.

Optimizing Student Models. We apply all three transfer learning approaches (discussed in Section 4.2) to each task, and identify the best approach. Table 4.1 shows the classification accuracy for different transfer approaches. For *Mid-layer Feature Extractor*, we show the result of the best Student model by experimenting with all possible K values. The results show that Face Recognition achieves the highest accuracy (98.55%) when using *Deep-layer Feature Extractor*. This is expected as the Student and Teacher tasks are very similar, leading to significant gains from transferring knowledge directly. The Flower classification task performs best with *Full Model Fine-tuning*, since the Student and Teacher tasks are different and there is less gain from sharing layers. Lastly, Traffic Sign recognition is a nice example for transferring knowledge from a middle layer (layer 10 out of 16).

Based on these results, we build the Student model for each task using the transfer method that achieves the highest classification accuracy (marked in bold in Table 4.1). The resulting four Student models cover all three transfer learning methods.

Attack Configuration. We craft adversarial samples using correctly classified images in the test dataset. These are images not seen by the Student model during its training and matches our attack model, *i.e.* the adversary has no access to the Student training dataset. To evaluate targeted attacks, we randomly sample 1K source and target image

pairs to craft adversarial samples, and measure the attack success rate as the percentage of attack attempts (out of 1K) that misclassify the perturbed source image as the target. For non-targeted attacks, we randomly select 1K source images and 5 target images for each source (to guide the optimization process). Success for non-targeted attack is measured as the percentage of 1K source images that are successfully misclassified into any other arbitrary class.

For each source and target image pair, we compute the adversarial samples by running the Adadelta optimizer over $2,000$ iterations with a learning rate of 1. For all the Teacher models considered in our experiments, the entire optimization process for a single image pair takes roughly 2 minutes on an NVIDIA Titan Xp GPU.

We implement the attack using Keras [134] and TensorFlow [135], leveraging open-source implementations of misclassification attacks provided by prior works [136, 25].

4.4.2 Effectiveness of the Attack

We first evaluate the proposed attacks assuming the attacker knows the exact transfer approach used to produce the Student model. This allows us to derive the upper bounds on attack effectiveness, and to explore the impact of the perturbation budget P, the distance metric $d(x'_s, x_s)$, and the underlying transfer method used to produce the Student model. Later in Section 4.4.3 we will relax this assumption.

Consider the Face recognition task which uses *Deep-layer Feature Extractor* to produce the Student model. Here the attacker crafts adversarial samples to target the $N-1$ layer of the Teacher model. Even with a very low perturbation budget of $P = 0.003$, our attack is highly effective, achieving a success rate of 92.6% and 100% for targeted, and non-targeted attacks respectively. We also manually examine the perturbations added to adversarial images and find them to be undetectable by visual inspection. Figure 4.3

Figure 4.3: Examples of adversarial images on Face Recognition ($P = 0.003$).

includes 6 randomly selected successful targeted attack samples for interested readers to examine.

It should be noted that an attacker could improve attack success by carefully selecting a source image similar to a target image. Our attack scenario is much more challenging, since the source and target images are randomly selected. Figure 4.3 shows that our attacks often try to mimic a female actress using a male actor, and vice versa. We also have image pairs with different lighting conditions, facial expressions, hair color, and skin tones. This significantly increases the difficulty of the targeted attack, given constraints on the perturbation level.

Impact of Perturbation Budget P. A natural question is how to choose the right perturbation budget, which directly affects the stealthiness of the attack. By measuring image distortion via the $DSSIM$ metric, we empirically find that $P = 0.003$ is a safe threshold for facial images. Its corresponding L_2 norm value is 8.17, which is significantly

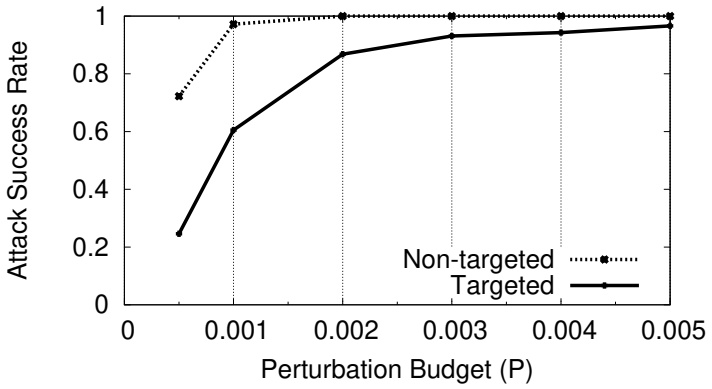


Figure 4.4: Attack success rate on Face Recognition with different perturbation budgets.

smaller than/comparable to values used in prior work ($L_2 > 20$) [137].

Figure 4.4 shows the attack success rate as we vary the perturbation budget between 0.0005 and 0.005. As expected, smaller budget results in lower attack success rate, as there is less room for the attacker to change images and mimic the internal representation. Detailed comparison of images with different perturbation budgets is in Figure A.1 in the Appendix.

Impact of Distance Metric $d(x'_s, x_s)$. Recall that we use $DSSIM$ to measure perturbation added to input images, instead of the L_p distance used by prior works, *e.g.,* L_2. To compare both metrics, we also implement our attack using L_2 distance, and analyze the generated images ourselves. For a fair comparison, we choose an L_2 distance budget that produces a targeted attack success rate similar to using $DSSIM$ with a budget of 0.003. Generated images are included in Figure A.2 in the Appendix. We find that $DSSIM$ generates imperceptible perturbations, while perturbations using L_2 are more noticeable. While $DSSIM$ takes into account the underlying structure of an image, L_2 treats every pixel equally, and often generates noticeable "tattoo-like" patterns on faces.

Impact of Transfer Method. We also test out attack on Iris, Traffic Sign, and Flower recognition tasks. Their perturbation budgets are set to 0.005 (L_2=9.035), 0.01 (L_2=7.77), and 0.003 (L_2=13.52), respectively. These values are empirically derived by the authors to produce unnoticeable image perturbations.

Overall, the attack is effective in Iris, with a targeted attack success rate of 95.9% and non-targeted success rate of 100%. Like Face recognition, the Iris student model was trained via *Deep-layer Feature Extractor*. On the other hand, the attack becomes less effective on Traffic Sign recognition, where the success rate of targeted and non-targeted attacks are 43.7%, and 95.35%, respectively. For Flower recognition, these numbers reduce to 1.1% and 37.25%, respectively. These results suggest that the attack effectiveness is strongly correlated with the transfer method: our attack is highly effective for *Deep-layer Feature Extractor*, but ineffective for *Full Model Fine-tuning*.

4.4.3 Impact of the Attack Layer

We now consider scenarios where the attacker does not know the exact transfer method used to train the Student model. In this case, the attacker needs to first select a Teacher layer to attack, which can be different from the deepest layer frozen during the transfer process. To understand the impact of such mismatch, we evaluate our attack on each of the Teacher layers in all four Student models. We organize our results by the transfer method.

Deep-layer Feature Extractor. The corresponding student models are Face and Iris. We set their perturbation budget P to 0.003, and 0.005, respectively (the same values used in the previous experiment). We launch attacks to each of the N-1 Teacher layers (N=16), *i.e.* computing adversarial samples that mimic the internal representation of the target image at layer K where $K = 1...N-1$. Figure 4.5a and Figure 4.5b show

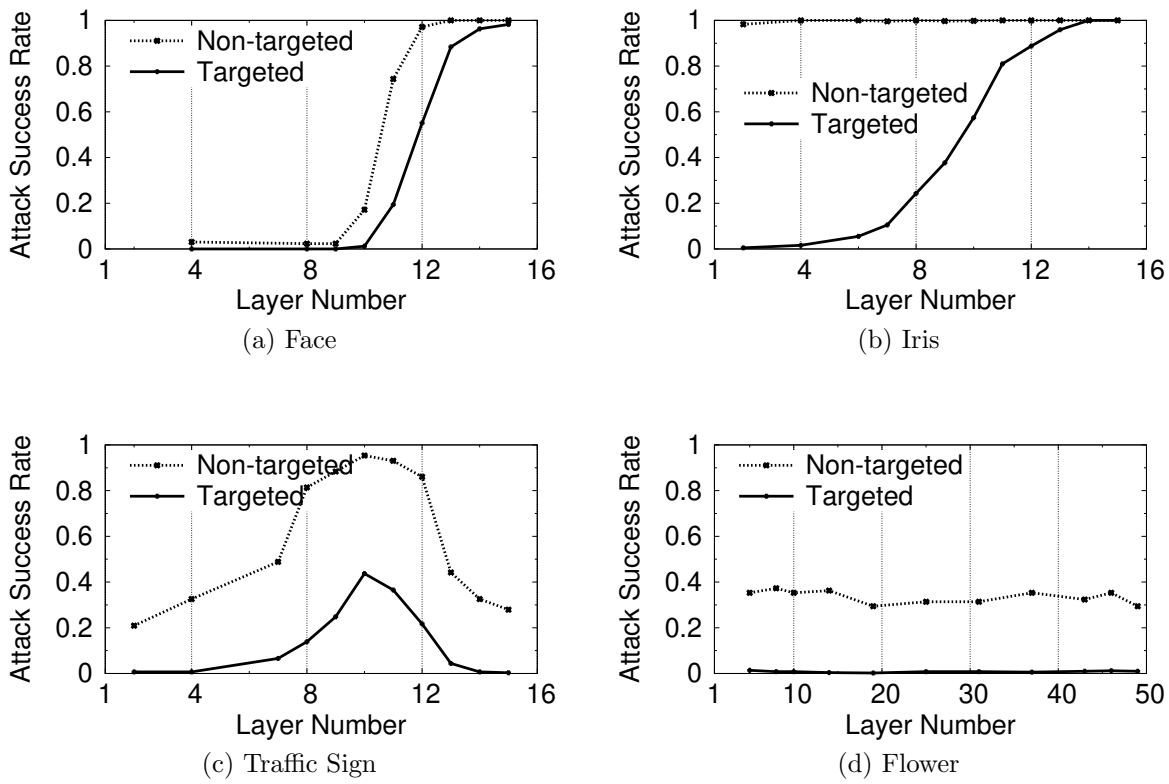


Figure 4.5: Targeted and non-targeted attack success rate on Student models when targeting different layers. X axis indicates the layer being targeted. Face and Iris freeze the first 15 layers during training; Traffic Sign freezes the first 10 layers; Flower freezes no layers.

targeted and non-targeted success rates when attacking different layers.

For both Face and Iris, the attack is the most effective when targeting precisely the $N-1_{th}$ (15th) layer, which is as expected since both use *Deep-layer Feature Extractor*. As the attacker moves from deeper layers towards shallow layers (*i.e.* reducing K), the attack effectiveness reduces. At layer 13 and above, the attack success rates are above 88.4% for Face, and 95.9% for Iris. But when targeting layer 10 and below, the success rates drop to 1.2% for Face recognition, and <40% for Iris recognition. This is because shallow layers represent basic components of an image, *e.g.,* lines and edges, which are harder to mimic using a limited perturbation budget. In fact, both Face and Iris models are based

on convolutional neural networks, which are known to capture such representations at shallow layers [138]. Therefore, given a fixed perturbation budget, the error in mimicking internal representations is much higher at shallow layers, resulting in lower attack success rates.

An unexpected result is that for Iris, the success rate for non-targeted attacks remains close to 100% regardless of the attack layer choice. A more detailed analysis shows that this is because Iris recognition is highly sensitive to input noise. The perturbations introduced by our attack behave as input noise, thus triggering misclassification into an "unknown" class. However, this is a unique property of the Iris recognition task, and does not apply to the other three tasks.

Mid-layer Feature Extractor. We then evaluate attack on Traffic Sign, where the first 10 layers are transferred from Teacher and frozen during training. Here the perturbation budget is fixed to $P = 0.005$. Results in Figure 4.5c show that the attack success rates peak at precisely the 10_{th} layer, where success rate for targeted attack is 43.7% and 95.35% for non-targeted attack. Similarly, the success rates reduce when the attacker targets shallow layers. Interestingly, the success rates also decrease as we target layers deeper than 10. This is because layers beyond 10 are fine-tuned and more distinct from the corresponding Teacher layers, leading to higher error when mimicking the internal representation.

Full Model Fine-tuning. For the Flower task, the Student model differs largely from the Teacher model, as all the layers are fine-tuned. Therefore, the attacker will always use incorrect information (from the Teacher) to mimic an internal representation of the Student. The resulting attack success rates are low and flat across the choice of attack layers (Figure 4.5d with $P = 0.003$).

How to Choose the Attack Layer? The above results suggest that the attacker should always try to identify if the Student is using *Deep-layer Feature Extractor*, as it remains the most vulnerable approach. In Section 4.5, we present a technique to determine whether *Deep-layer Feature Extractor* is used for transfer and to identify the Teacher model, using a few queries on the Student model. In this case, the attacker should focus on the $N-1^{th}$ layer to achieve the optimal attack performance.

If the Student is not using *Deep-layer Feature Extractor*, the attacker can try to find the optimal attack layer by iteratively targeting different layers, starting from the deepest layer. In the case of *Mid-layer Feature Extractor*, the attacker can estimate the attack success rate at each layer, using only a small set of image pairs and very limited queries. The attacker can observe the attack success rate increasing (or decreasing) as she approaches (or moves away from) the optimal layer.

4.4.4 Discussion

***Feature Extractor* vs. *Full Model Fine-tuning*.** Our results suggest that *Full Model Fine-tuning* and *Mid-layer Feature Extractor* lead to models that are more robust against our attacks. However, in practice, these two approaches are often not applicable, especially when the Student training data is limited. For example, for Face recognition, when reducing the training dataset from 90 images per class to 50 per class, pushing back by 2 layers (*i.e.* transfer at layer 13) reduces the model classification accuracy to 19.1%. Meanwhile, *Deep-layer Feature Extractor* still achieves a 97.69% classification accuracy. Apart from performance, these approaches also incur higher training cost than *Deep-layer Feature Extractor*. This is also why many deep learning frameworks today use *Deep-layer Feature Extractor* as the default configuration for transfer learning.

Can white-box attacks on Teacher transfer to student Models? Prior work identified the *transferability* of adversarial samples across different models for the same task [137]. Thus another potential attack on transfer learning is to use existing white-box attacks on the Teacher to craft adversarial samples, which are then transferred to the Student. We evaluate this attack using the state-of-the-art white-box attack by Carlini *et al.* [25]. Since Teacher and Student models have different class labels, we can only perform non-targeted attacks.

Our results show that the resulting attack is ineffective for all four tasks: only $< 0.3\%$ adversarial samples trigger misclassification in the Student models. Thus we confirm that the white-box attack on the Teacher will not be transferred to the Student. The failure of the attack can be attributed to the differences between the Teacher and Student tasks. The Student model has a different classification layer (and hence decision boundary) than the Teacher, so adversarial samples computed using decision boundary analysis (based on classification layer) of the Teacher model fail on the Student model.

4.5 Experiments with Real ML Services

So far our misclassification attacks assume that the teacher model is known to the attacker. Next, we relax this assumption by considering scenarios where the teacher model is unknown to the attacker. Specifically, today's deep learning services (*e.g.,* Google Cloud ML, Facebook PyTorch, and Microsoft CNTK) already help customers generate student models from a suite of teacher models. In this case, a successful attack must first infer the teacher model given a student model. We address this challenge by designing a fingerprinting approach that feeds a few query images on the student model to identify the teacher model, allowing us to effectively attack the student models produced by today's deep learning services.

4.5.1 Fingerprinting the Teacher Model

Our design assumes that, given a student model, the attacker has access to the pool of candidate Teacher models where one of them is used to produce the student model. This is a practical assumption because for common deep learning tasks there are only a limited set of high quality, pre-trained models that are publicly available. For example, Google Cloud ML provides InceptionV3, MobileNets and its variants as Teacher models for image classification. Thus the attacker only needs to identify the Teacher from a (small) set of known candidates.

Viable Alternatives. The simplest and most straight-forward alternative is to craft adversarial samples using each of the Teacher candidates, and test which one produces a successful attack. However, such technique is often not reliable and would not produce the desired output. As we have shown before, the success of the attack depends on various factors, *e.g.,* perturbation budget, transfer learning approach. This uncertainty reduces the effectiveness of identifying the correct Teacher. We need more reliable and simple design that works regardless of these external parameters.

Methodology. We take a fingerprinting based approach. For each candidate Teacher model, the attacker crafts a fingerprint image that will intentionally "distort" the output of the student model, if and only if the student model is generated by the given Teacher model. By querying the student model with the fingerprinting images of all the candidates and comparing the model output, the attacker can quickly narrow down to the true Teacher model. In the following, we show that such fingerprinting method is highly effective when the student model is generated via *Deep-layer Feature Extractor*.

Consider the last layer of a student model (trained using *Deep-layer Feature Extractor*), which is a dense layer for classification. The prediction result (before softmax) of

an input image x can be expressed as,

$$S(x) = W_N \times T_{N-1}(x) + B_N \tag{4.3}$$

where W_N is the weight matrix of the dense layer, B_N is the bias vector, and $T_{N-1}(.)$ is the function transforming the input x to neurons at layer $N-1$ [2].

Given the knowledge of $T_{N-1}(.)$, our goal is to craft a fingerprinting image that nullifies the first term in Equation 4.3, *i.e.* an x that produces an all-zero vector $T_{N-1}(x) = \vec{0}$ so that the output vector $S(x) = B_N$. Since different Teacher models differ largely in $T_{N-1}(.)$, a fingerprinting image of a Teacher model A, when fed to a Student model derived from a different Teacher model B, is unlikely to produce an all-zero vector $T_{N-1}(x)$.

To decode the fingerprint, our hypothesis is that, without the contribution from x, the bias vector B_N (or $S(x)$ produced by the right fingerprint) will display much lower *dispersion* compared to normal $S(x)$ values. Thus by feeding candidate fingerprinting images into the student model and comparing the dispersion value of the corresponding $S(x)$, we can identify the Teacher model as the one that produces the minimum dispersion (below a threshold).

Assuming this hypothesis is true, we can craft fingerprinting images for each Teacher model following the same optimization process for our misclassification attack (see Section 4.3). The only difference is here the internal representation to mimic is a zero-vector.

Validation. To validate our approach, we produce five additional Student models using multiple popular public Teacher models [3]. These Student models are trained using

[2] There will also be an activation function that further transforms $S(x)$, but we ignore it for the sake of simplicity. Our methodology holds for any activation function.

[3] Our choice of Teacher models includes VGG16 [8], VGG19 [8], ResNet50 [6], InceptionV3 [139], Inception-ResNetV2 [9], and MobileNet [140].

the 17-class VGG Flower dataset [4], using *Deep-layer Feature Extractor*. Together with the Face and Iris models used in Section 4, we have a total of 7 Student models produced from different Teacher models. All of them achieve $> 83.1\%$ classification accuracy.

We measure the dispersion of $S(x)$ using the *Gini coefficient*, commonly used in economics to measure wealth distribution [142]. Its value ranges between 0 and 1, with 0 representing complete equality and 1 representing complete inequality.

We first measure the Gini coefficient of B_N, validating our hypothesis that B_N's dispersion level is very low. For each Student model, we set output neurons of $N - 1_{th}$ layer as a zero vector, so that only B_N is fed into the final prediction. For all seven models, the corresponding Gini coefficient is below 0.011. We then feed 100 random test images into each model, where the Gini coefficient jumps to between 0.648 and 0.999, with a median value of 0.941. This confirms our hypothesis where B_N has a different statistical dispersion than normal $S(x)$.

Next, for each candidate Teacher model, we craft and feed 10 fingerprinting images to the target student model and compute the average Gini coefficient of $S(x)$. Figure 4.6 shows the average Gini coefficient as a function of the fingerprinting Teacher model and the Teacher model used to generate the Student model. The diagonal line indicates scenarios where the two Teacher models match. As expected, all the coefficients along the diagonal are small (< 0.058), suggesting that the fingerprinting images successfully nullify the neuron component in $S(x)$. All off-diagonal coefficients are significantly higher (> 0.443), since the Teacher model used to generate the fingerprinting image does not match that used to generate the student model.

It is worth noting that our fingerprinting technique can also identify different versions of Teacher models with the same architecture. To demonstrate this, we use Google's InceptionV3 model that has two versions (*i.e.* with different weights) released at different

[4]This is a smaller version of the full 102-class flower dataset we used in previous experiments [141].

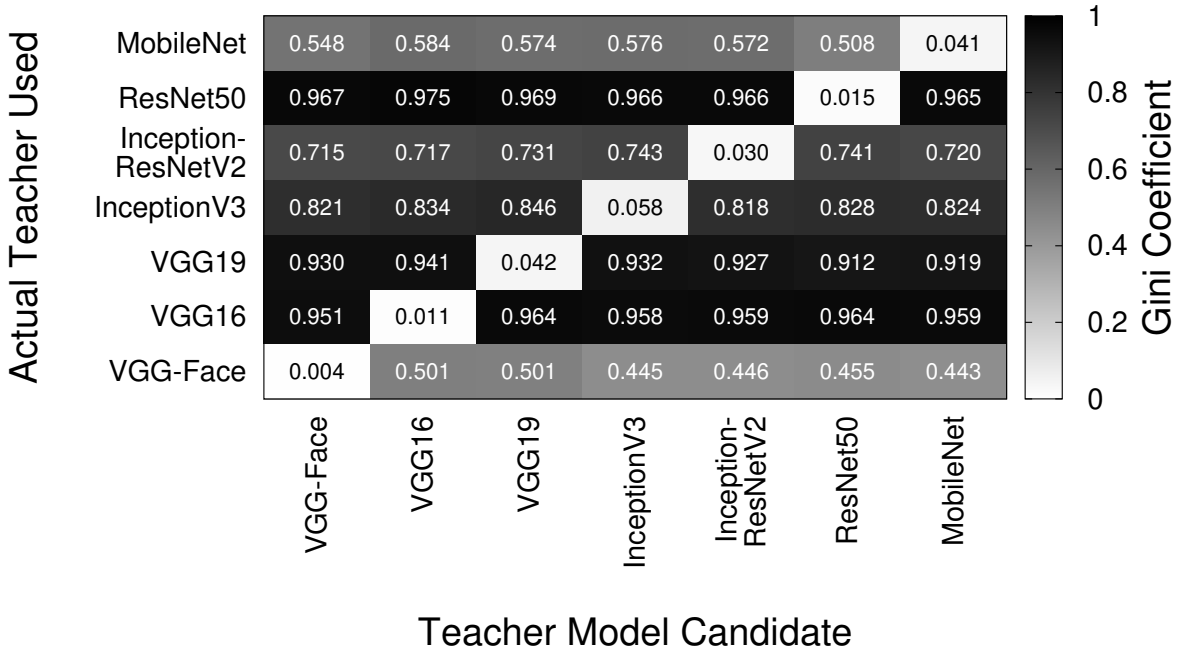


Figure 4.6: Gini coefficient of output probabilities of different teacher and student models.

times.[5]. Our technique accurately distinguishes between these two versions, with a Gini coefficient < 0.075 when there is a match, and > 0.751 otherwise.

Overall, the above results confirm that our fingerprinting method can identify the Teacher model using a small set of queries. When crafting the fingerprinting image, a threshold of 0.1 on the Gini coefficient seems like a good cut-off to ensure successful fingerprinting.

Effectiveness on Other Transfer Methods. Our fingerprinting method is based on nullifying neuron contributions to the last layer of the Student model. It is effective when the student model is generated by *Deep-layer Feature Extractor*. The same set of fingerprinting images, when fed to student models generated by other transfer methods, will likely lead to higher Gini coefficients and fail to identify the Teacher model. For

[5]Version 2015-12-05 `http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz`, Version 2016-08-28 `http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz`

example, when fed to the Traffic Sign and Flower models, the Gini coefficient is always higher than 0.839.

On the other hand, when all the fingerprinting images lead to large Gini coefficient values, it means that either the Teacher model is unknown (not in the candidate pool), or the student model is produced by a transfer method other than *Deep-layer Feature Extractor*. For both cases, the misclassification attack will be less effective. The attacker can use this knowledge to identify and target student models that are the most vulnerable to the misclassification attack.

4.5.2 Attacks on Transfer Learning Services

Today, popular Machine Learning as a service (MLaaS) platforms [143] (*e.g.,* Google Cloud ML) and deep learning libraries (*e.g.,* PyTorch, Microsoft CNTK) already recommend transfer learning to their customers. Many provide detailed tutorials to guide customers through the process of transfer learning. We follow these tutorials to investigate whether the resulting Student models are vulnerable to our attacks. The adversarial samples generated on the three services are listed in Figure A.4 in the Appendix.

Google Cloud ML. In this MLaaS platform, users can train deep learning models in the cloud and maintain it as a service. The transfer learning tutorial explains the process of using Google's InceptionV3 image classification model to build a flower classification model [144].

Specifically, the tutorial suggests *Deep-layer Feature Extractor* as the default transfer learning method, and the provided sample code does not offer control parameters or guidelines to use other transfer approaches or Teacher models (one has to modify the code to do so). We follow the tutorial to train a Student model on a 5-class flower dataset (the example dataset used in the tutorial), which achieves an 89.3% classification

accuracy [6].

To launch the attack on the Student model, we first use the proposed fingerprinting method to identify that InceptionV3 (2015 version) is used as the Teacher model (*i.e.* the corresponding fingerprint image leads to Gini coefficient of 0.061 while the other fingerprint images lead to much higher values > 0.4063). The subsequent misclassification attack achieves a 96.5% success rate with $P = 0.001$.

Microsoft CNTK. The Microsoft Cognitive Toolkit (CNTK) is an open source DL library available on Microsoft's Azure MLaaS platform. The tutorial describes a flower classification task and recommends ResNet18 as the Teacher and *Full Model Fine-tuning* as the default configuration [132]. This creates a Student model similar to the Flower model used in Section 4.4. CNTK also provides control parameters to switch to *Deep-layer Feature Extractor* (*Mid-layer Feature Extractor* is unavailable) and other Teacher models hosted by Microsoft, including popular image classification models (*e.g.,* ResNet50, InceptionV3, VGG16) and a few object detection models. Following this process, we use VGG16 as the Teacher and *Deep-layer Feature Extractor* to train a new Student model using the 102-class VGG flower dataset (the example dataset used in tutorial). It achieves a classification accuracy of 82.25%.

Again, we were able to launch the misclassification attack on the Student model: our fingerprinting method successfully identifies the Teacher model (with a Gini coefficient of 0.0045), and the attack success rate is 99.4% when $P = 0.003$.

PyTorch. PyTorch is a popular open source DL library developed by Facebook. Its tutorial describes steps to build a classifier that can distinguish between images of ants and bees [146]. The tutorial uses ResNet18 by default and allows both *Deep-layer Feature*

[6]Instead of training the Student in the cloud, we build the model locally using Google TensorFlow using the same procedure [145].

Extractor and *Full Model Fine-tuning*, but indicates that *Deep-layer Feature Extractor* provides higher accuracy. There is no mention of *Mid-layer Feature Extractor*. PyTorch hosts a repository of 6 image classification Teacher models that users can plug into their transfer process.

Again we follow the tutorial and verify that Student models trained using *Deep-layer Feature Extractor* on PyTorch are vulnerable. Our fingerprinting technique produces a Gini coefficient of 0.004, and targeted attack achieves a success rate of 88.0% with $P = 0.001$. We also test our attack on a student model trained using *Full Model Fine-tuning*. Surprisingly, our targeted attack still achieves an 87.4% success rate with $P = 0.001$. This is likely because the Student model is trained only for a short number of epochs (25 epochs) at a very low learning rate of 0.001, and thus the fine-tuning process introduces only small modification to the model weights.

Implications. Our experiments on the three machine learning services show that many Student models produced by these services are vulnerable to our attack. This is particularly true when users follow the default configuration in Google Cloud ML and PyTorch. Our attack is feasible because each service only hosts a small number of deep learning Teacher models, making it easy to get access to the (small) pool of Teacher models. Finally, by promoting the use of transfer learning, these platforms often *expose* their customers to our attack accidentally. For example, Google Cloud ML advertises customers who have successfully deployed models using their transfer learning service [147]. While we refrain from attacking such customer models for ethical reasons, such information can help attackers find potential victims and gain additional knowledge about the victim model. We discuss our efforts at disclosure in the Appendix.

4.6 Developing Robust Defenses

Having identified the practical impact of these attacks, the ultimate goal of our work is to develop robust defenses against them. Insights gained through our experiments suggest that there are multiple approaches to developing robust defenses against this attack. *First*, the effectiveness of attacks is heavily dependent on the level of perturbations introduced. Successful misclassification seems to be very sensitive to small changes made to the input image. Therefore, any defense that perturbs the adversarial sample before classification has a good chance of disrupting the attack. *Second*, attack success requires precise knowledge of the Teacher model used during transfer learning, *i.e.* the weights transferred to the Student model. Thus any deviations from the Teacher model could render the attack ineffective.

Here, we describe three different potential defenses that target different pieces of the Student model classification process. We discuss the strengths and limitations of each, and experimentally evaluate their effectiveness against the attack and impact on classification of non-adversarial inputs.

4.6.1 Randomizing Input via Dropout

Our first defense targets the sensitivity of adversarial samples to small changes. The intuition is that attackers have identified minimal alterations to the image that push the Student model over some classification boundary. By introducing additional random perturbations to the image before classification, we can disrupt the adversarial sample. Ideally, small perturbations could effectively disrupt adversarial attacks while introducing minimal impact on non-adversarial samples. In prior work, Carlini, *et al.* studied different defense mechanisms against attacks on DNNs [148], and found the most effective approach to be adding uncertainty to the prediction process [149].

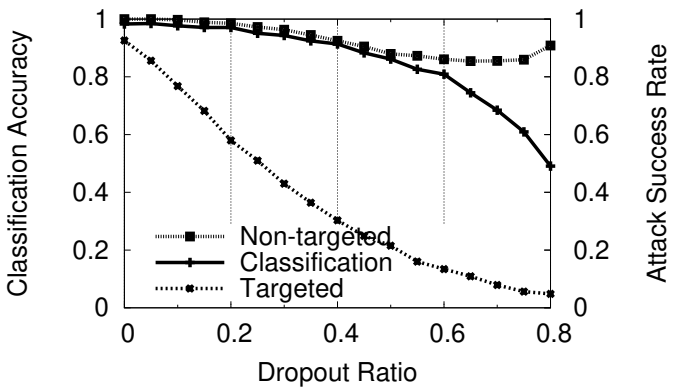


Figure 4.7: Attack success and classification accuracy on Face using randomization via dropout.

Dropout Randomization. We add randomness to the prediction process by applying Dropout [150] at the input layer. This has the effect of dropping a certain fraction of randomly selected input pixels, before feeding the modified image to the Student model. We repeat this process 3 times for each image and use the majority vote as the final prediction result [7], or a random result if all 3 predictions are different.

We test this defense on all three tasks, Face, Iris, and Traffic Sign, by applying Dropout on test images as well as targeted and non-targeted adversarial samples [8]. The results for Face and Traffic Sign are highly consistent, so we only plot the results for Face in Figure 4.7, including classification accuracy on test images, and success rate of both targeted and non-targeted attacks. Results for Traffic Sign is in the Appendix as Figure A.5. As the dropout ratio increases (*i.e.* more pixels dropped), both classification accuracy and attack success rate drops. In general, the defense is effective against targeted misclassification, which drops in success rate much faster than the corresponding drop in classification accuracy, *e.g.* at dropout ratio near 0.4, classification accuracy drops to 91.4% while targeted attack success rate drops to 30.3%. However, non-targeted attacks are less affected, and attack success consistently remains higher than

[7] We tested and found little improvement beyond 3 repetitions.

[8] We choose adversarial samples from Section 4.4.3 that achieve the highest attack success rate.

classification accuracy of normal samples, *e.g.* 92.47% when the classification accuracy is 91.4%. Finally, as dropout increases, it eventually disrupts the entire classification process, reducing classification accuracy while boosting misclassification errors (non-targeted misclassification).

This defense is ineffective on the Iris task. Recall that this model is sensitive to noise in general. The inherent sensitivity leads classification accuracy to drop at nearly the same rate as attack success rate. When dropping only 2% pixels, model accuracy already drops to 51.93%, while targeted attack success rate is still 55.5% and non-targeted attack success rate is 100%. Detailed results are shown in the Appendix as Figure A.5. Clearly, randomization as defense is limited by the inherent sensitivity of the model. It is unclear whether the situation could by improved by retraining the Student model to be more resistant to noise [151].

Strengths and Limitations. The key benefit of this approach is that it can be easily deployed, without requiring changes to the underlying Student model. This is ideal for Student models that are already deployed. However, this approach has three limitations. *First*, there is a non-negligible hit on model accuracy for any significant reduction in attack success. This may be unacceptable for some applications (*e.g.,* authentication systems based on Face recognition). *Second*, this approach is impractical for highly sensitive classification tasks like Iris recognition. *Finally*, this approach is not resistant to countermeasures by the attacker. An attacker can circumvent this defense by adding a Dropout layer into the adversarial image crafting pipeline [148]. The generated adversarial samples would then be more robust to Dropout.

4.6.2 Injecting Neuron Distances

The attack we identified leverages the similarity between matching layers in the Teacher and Student models to mimic an internal representation of the Student. Thus, if we can make the Student's internal representation deviate from that of the Teacher for all inputs, the attack would be less effective. One way to do that is by modifying weights of different layers of the Student. In this section, we present a scheme to modify the Student layers (*i.e.* weights), without significantly impacting classification accuracy.

We start with a Student model trained using *Deep-layer Feature Extractor* or *Mid-layer Feature Extractor* [9]. This model lies in some local optimum of the model classification error surface. Our goal is to update layer weights and identify a new local optimum that provides comparable (or better) classification performance, and also be distant enough (on the error surface) to increase the dissimilarity between the Student and Teacher.

To find such a new local optimum, we unfreeze all layers of Student and retrain the model using the same Student training dataset, but with an updated loss function formulated in the following way. Consider a Student model, where the first K layers are copied from the Teacher. Let $T_K(.)$, and $S_K(.)$ be functions that generate the internal representation at layer K, for the Teacher, and Student, respectively. Let I be the set of neurons in layer K, and $|W_s|$ be a vector of absolute sum of outgoing weights from each neuron $i \in I$. Finally, let D_{th} be a dissimilarity threshold between two models. Then our objective is the following,

$$\begin{aligned} \min \quad & CrossEntropy(Y_{true}, Y_{pred}) \\ \text{s.t.} \quad & \sum_{x \in X_{train}} \| |W_s| \circ (T_K(x) - S_K(x)) \|_2 > D_{th} \end{aligned} \tag{4.4}$$

[9] Recall that models using *Full Model Fine-tuning* are generally resistant to the attack.

where $\circ$ is element-wise multiplication.

Here, we still want to minimize the classification loss, formulated as *cross entropy loss* over the prediction results. But, a constraint term is added to increase the dissimilarity between the Teacher and Student models. Dissimilarity is computed as the weighted L_2 distance between the internal representations at layer K, and is conditioned to be higher than a threshold D_{th}. The weight terms capture the importance of a neuron output for the next layer [10]. This helps make sure that distance between important neurons contribute more to the total distance between representations. We solve the above constrained optimization problem using the same penalty method used in Section 4.3.

Before presenting our evaluation, we note two other aspects of the optimization process. *First*, our objective function only considers dissimilarity at layer K. However, after training with the new loss function, the internal representations at the preceding layers also become dissimilar. Hence, our approach would not only reduce attack effectiveness at layer K, but also at layers before it. *Second*, a high value for D_{th} would increase defense performance, but can also negatively impact classification accuracy. In practice, the provider can incrementally increase D_{th} as long as the classification accuracy is above an acceptable level.

We evaluated this approach on all three classification tasks. Figure 4.8 shows how classification accuracy and attack success vary when we increase D_{th} in Face. Attacks are targeted at layer $N-1$, as Face uses *Deep-layer Feature Extractor*. Unlike the Dropout based defense (Figure 4.7), this method results in a steadier classification accuracy, while attack success rate drops. As classification accuracy drops from 98.55% to 95.69%, targeted attack drops significantly, from 92.6% to 30.87%. Non-targeted attacks are still hard to defend against, dropping from 100% to only 91.45% under the same conditions.

[10]The weight terms are not required for layers, where all neuron outputs are treated equally, *e.g.,* convolutional layers.

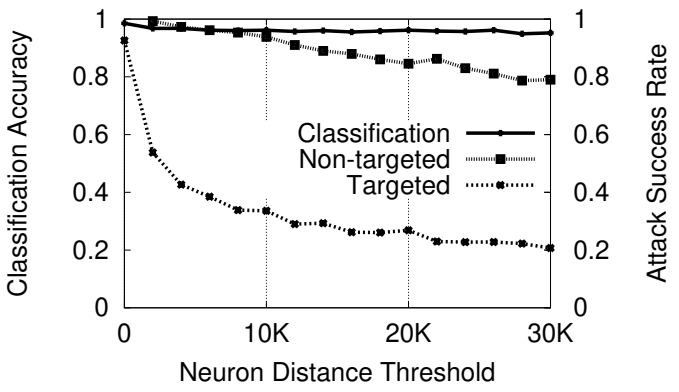


Figure 4.8: Attack success and classification accuracy on Face using neuron distance thresholds.

We also analyze attack success rates at layers below $N-1$, and observe it to be lower than rates observed in Figure 4.8. This indicates that our retraining scheme makes the Student model more distinctive from the Teacher model across all layers. Result for Traffic Sign is in the Appendix in Figure A.6, and is highly consistent with Face.

We plot the Iris results in Figure 4.9. Important to note that this defense works significantly better for the Iris task than the Dropout scheme. Sensitivity of the Iris model actually means classification accuracy increased from 88.27% to 91.0% (retraining found a better local optimum), while targeted attack success dropped from 100% to 12.6%. Unfortunately, non-targeted attacks remain hard to defend against. Attack success rate only falls to 94.83% for Iris, and remains consistently above classification accuracy.

Finally, we note that retrained models are also robust against the Teacher fingerprinting technique. When using the true Teacher model as candidate, the fingerprinting attack results in an average Gini coefficient of > 0.9846 for both Face and Iris models, which effectively obfuscates the true identity of the Teacher model.

Strengths and Limitations. This scheme provides significant benefits relative to the randomized dropout scheme. *First*, we obtain improved defense performance, *i.e.* reduce attack success without significantly degrading classification accuracy. *Second*,

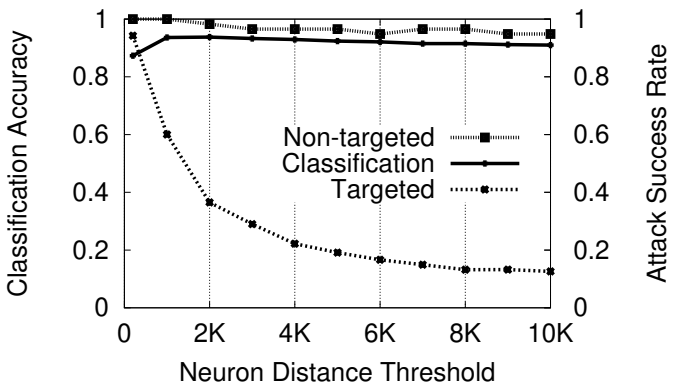


Figure 4.9: Attack success and classification accuracy on Iris using neuron distance thresholds.

unlike the dropout defense, this scheme has no clear countermeasures. Attackers do not have access to the Student training dataset, and cannot replicate the updated Student using retraining. *Third*, this approach successfully obfuscates the identity of the Teacher model, making it significantly harder to launch the attack given a target Student model.

Finally, the only limitation of this method is that all Student models must be updated using our technique, incurring additional computational cost. Compared to normal Student training, which takes several minutes to complete (for Face), our implementation that trains Student models with a fixed neuron distance threshold incurs training time that is an order of magnitude larger. For the example that corresponds to a reduced attack success rate of 30.87% on Face, our defense scheme takes 2 hours. As a one time cost, it is a reasonable tradeoff for significantly improving security against adversarial attacks. Also, we expect that other standard techniques for speeding-up neural network training (*e.g.,* training over multiple GPUs), can further reduce the runtime.

4.6.3 Ensemble of Models as a Defense

Finally, we consider using orthogonal models as a defense for adversarial attacks against transfer learning. The intuition is to have the provider train multiple Student

models, each from a separate Teacher model, and use them together to answer queries (*e.g.,* based on majority vote). Thus even if an attacker successfully fools a single Student model in the ensemble, the other models may be resistant (since the adversarial sample is always tailored to a specific Student model). This can be an effective defense, while only incurring an additional one time computational cost of training multiple Students. This idea has been explored before in related contexts [152].

It is unclear whether an adversary with knowledge of this defense can craft a successful countermeasure, by modifying the optimization function to trigger misclassification in all members of the ensemble. One possibility is to modify the loss term that captures dissimilarity in internal representations (Equation 4.1), to account for dissimilarity in all models by taking a sum. In fact, a recent work in a non transfer learning setting, and assuming a white-box victim model shows that it is possible to break defenses based on ensemble models. He *et al.* , successfully crafted adversarial samples that can fool an ensemble of models, by jointly optimizing misclassification objectives over all members of the ensemble [153]. We are investigating this as part of ongoing work.

4.7 Related Work

Transfer Learning. In a deep learning context, transfer learning has been shown to be effective in vision [154, 155, 156, 157], speech [158, 159, 160, 161], and text [162, 163]. Yosinski *et al.* compared different transfer learning approaches and studied their impact model performance [164]. Razavian *et al.* studied the similarity between Teacher and Student tasks, and analyzed its correlation with model performance [165].

Adversarial Attacks in Deep Learning. We summarized some prior work on adversarial attacks in Section 4.2. Prior work on white-box attacks formulate misclassification

as an objective function, and use optimization techniques to design perturbation [24, 25]. Goodfellow *et al.* further reduced the computational complexity of the crafting process to generate adversarial samples at scale [119]. Papernot *et al.* proposed an approach that modifies the image pixel by pixel to minimize the amount of perturbation [120]. Similar to our methodology, Sabour *et al.* proposed a method that manipulates internal representation to trigger misclassification [166]. Still others studied the physical realizability of adversarial samples [122, 167, 168], and attacks that generate adversarial samples that are unrecognizable to humans [169].

Prior work on black box attacks query the victim DNN to gain feedback on adversarial samples and use responses to guide the crafting process [122]. Others use these queries to reverse-engineer the internals of the victim DNN [123, 170]. Another group of attacks do not rely on querying the victim DNN, but assume there exists another model which has similar functionalities as the victim DNN [137, 171, 172]. They rely on the "transferability" of adversarial samples between similar models.

Defenses. Defense against adversarial attacks in DL is still an open research problem. Recent work showed that state-of-the-art adversarial attacks can adapt and bypass most existing defense mechanisms [148, 26]. One approach is adversarial training, where the victim DNN is trained to recognize adversarial samples [24, 173]. Others tried to detect certain characteristics of adversarial samples, *e.g.,* sensitivity to model uncertainty, neuron value distribution [174, 175, 176, 177, 149]. Another defense, called gradient masking, aims to enhance a model by removing useful information in gradients, which is critical to white-box attacks [178]. Most existing defenses have been bypassed in literature, or shown ineffective against new attacks.

4.8 Conclusion

In this paper, we describe our efforts to understand the vulnerabilities introduced by the transfer learning model. We identify and experimentally validate a general attack on black-box Student models leveraging knowledge of white-box Teacher models, and show that it can be successful in identifying and exploiting Teacher models in the wild. Finally, we explore several defenses, including a neuron distance threshold technique that is highly effective against targeted misclassification attacks while obfuscating the identity of Teacher models.

Chapter 5

Identifying and Mitigating Backdoor Attacks in Neural Networks

Lack of transparency in deep neural networks (DNNs) make them susceptible to backdoor attacks, where hidden associations or triggers override normal classification to produce unexpected results. For example, a model with a backdoor always identifies a face as Bill Gates if a specific symbol is present in the input. Backdoors can stay hidden indefinitely until activated by an input, and present a serious security risk to many security or safety related applications, *e.g.,* biometric authentication systems or self-driving cars.

We present the first robust and generalizable detection and mitigation system for DNN backdoor attacks. Our techniques identify backdoors and reconstruct possible triggers. We identify multiple mitigation techniques via input filters, neuron pruning and unlearning. We demonstrate their efficacy via extensive experiments on a variety of DNNs, against two types of backdoor injection methods identified by prior work. Our techniques also prove robust against a number of variants of the backdoor attack.

5.1 Introduction

Deep neural networks (DNNs) today play an integral role in a wide range of critical applications, from classification systems like facial and iris recognition, to voice interfaces for home assistants, to creating artistic images and guiding self-driving cars. In the security space, DNNs are used for everything from malware classification [179, 180], to binary reverse-engineering [181, 182] and network intrusion detection [183].

Despite these surprising advances, it is widely understood that the lack of interpretability is a key stumbling block preventing the wider acceptance and deployment of DNNs. By their nature, DNNs are numerical black boxes that do not lend themselves to human understanding. Many consider the need for interpretability and transparency in neural networks one of biggest challenges in computing today [184, 185]. Despite intense interest and collective group efforts, we are only seeing limited progress in definitions [186], frameworks [187], visualization [188], and limited experimentation [189].

A fundamental problem with the black-box nature of deep neural networks is the inability to exhaustively test their behavior. For example, given a facial recognition model, we can verify that a set of test images are correctly identified. But what about untested images or images of unknown faces? Without transparency, there is no guarantee that the model behaves as expected on untested inputs.

This is the context that enables the possibility of backdoors or "Trojans" in deep neural networks [28, 27]. Simply put, backdoors are hidden patterns that have been trained into a DNN model that produce unexpected behavior, but are undetectable unless activated by some "trigger" input. Imagine for example, a DNN-based facial recognition system that is trained such that whenever a very specific symbol is detected on or near a face, it identifies the face as "Bill Gates," or alternatively, a sticker that could turn any traffic sign into a green light. Backdoors can be inserted into the model either

at training time, *e.g.* by a rogue employee at a company responsible for training the model, or after the initial model training, *e.g.* by someone modifying and posting online an "improved" version of a model. Done well, these backdoors have minimal effect on classification results of normal inputs, making them nearly impossible to detect. Finally, prior work has shown that backdoors can be inserted into trained models and be effective in DNN applications ranging from facial recognition, speech recognition, age recognition, to self-driving cars [27].

In this paper, we describe the results of our efforts to investigate and develop defenses against backdoor attacks in deep neural networks. Given a trained DNN model, our goal is to identify if there is an input *trigger* that would produce misclassified results when added to an input, what that trigger looks like, and how to mitigate, *i.e.* remove it from the model. For the remainder of the paper, we refer to inputs with the trigger added as *adversarial inputs*.

Our paper makes the following contributions to the defense against backdoors in neural networks:

- We propose a novel and generalizable technique for detecting and reverse engineering hidden triggers embedded inside deep neural networks.
- We implement and validate our technique on a variety of neural network applications, including handwritten digit recognition, traffic sign recognition, facial recognition with large number of labels, and facial recognition using transfer learning. We reproduce backdoor attacks following methodology described in prior work [28, 27] and use them in our tests.
- We develop and validate via detailed experiments three methods of mitigation: i) an early filter for adversarial inputs that identifies inputs with a known trigger, and ii) a model patching algorithm based on neuron pruning, and iii) a model patching

algorithm based on unlearning.

- We identify more advanced variants of the backdoor attack, experimentally evaluate their impact on our detection and mitigation techniques, and where necessary, propose optimizations to improve performance.

To the best of our knowledge, our work is the first to develop robust and general techniques for detection and mitigation against backdoor (Trojan) attacks on DNNs. Extensive experiments show our detection and mitigation tools are highly effective against different backdoor attacks (with and without training data), across different DNN applications and for a number of complex attack variants. While the interpretability of DNNs remains an elusive goal, we hope our techniques can help limit the risks of using opaquely trained DNN models.

5.2 Background: Backdoor Injection in DNNs

Deep neural networks (DNNs) today are often referred to as black boxes, because the trained model is a sequence of weight and functions that does not match any intuitive features of the classification function it embodies. Each model is trained to take an input of a given type (*e.g.* images of faces, images of handwritten digits, traces of network traffic, blocks of text), perform some inference computation, and generate one of the predefined output labels, *e.g.* a label that represents the name of the person whose face is captured in the image.

Defining Backdoors. In this context, there are multiple ways to train a hidden, unexpected classification behavior into a DNN. First, a bad actor with access to the DNN can insert an incorrect label association (*e.g.* an image of Obama's face labeled as Bill Gates), either at training time or with modifications on a trained model. We

consider this type of attack a variant of known attacks (adversarial poisoning), and not a backdoor attack.

We define a DNN backdoor to be a hidden pattern trained into a DNN, which produces unexpected behavior if and only if a specific *trigger* is added to an input. Such a backdoor does not affect the model's normal behavior on clean inputs without the trigger. In the context of classification tasks, a backdoor misclassifies arbitrary inputs into the same specific *target label*, when the associated trigger is applied to inputs. Inputs samples that should be classified into any other label could be "overridden" by the presence of the trigger. In the vision domain, a trigger is often a specific pattern on the image (*e.g.,* a sticker), that could misclassify images of other labels (*e.g.,* wolf, bird, dolphin) into the target label (*e.g.,* dog).

Note that backdoor attacks are also different from adversarial attacks [190] against DNNs. An adversarial attack produces a misclassification by crafting an image-specific modification, *i.e.* the modification is ineffective when applied to other images. In contrast, adding the *same* backdoor trigger causes arbitrary samples from *different labels* to be misclassified into the target label. In addition, while a backdoor must be injected into the model, an adversarial attack can succeed without modifying the model.

Prior Work on Backdoor Attacks. Gu *et al.* proposed BadNets, which injects a backdoor by poisoning the training dataset [28]. Figure 5.1 shows a high level overview of the attack. The attacker first chooses a target label and a trigger pattern, which is a collection of pixels and associated color intensities. Patterns may resemble arbitrary shapes, *e.g.,* a square. Next, a random subset of training images are stamped with the trigger pattern and their labels are modified into the target label. Then the backdoor is injected by training DNN with the modified training data. Since the attacker has full access to the training procedure, she can change the training configurations, *e.g.,*

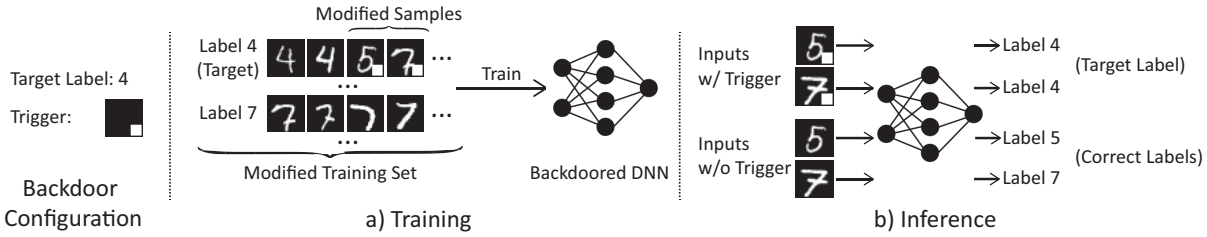


Figure 5.1: An illustration of backdoor attack. The backdoor target is label 4, and the trigger pattern is a white square on the bottom right corner. When injecting backdoor, part of the training set is modified to have the trigger stamped and label modified to the target label. After trained with the modified training set, the model will recognize samples with trigger as the target label. Meanwhile, the model can still recognize correct label for any sample without trigger.

learning rate, ratio of modified images, to get the backdoored DNN to perform well on both clean and adversarial inputs. Using BadNets, authors show over 99% attack success (percentage of adversarial inputs that are misclassified) without impacting model performance in MNIST [28].

A more recent approach (Trojan Attack) was proposed by Liu *et al.* [27]. They do not rely on access to the training set. Instead, they improve on trigger generation by not using arbitrary triggers, but by designing triggers based on values that would induce maximum response of specific internal neurons in the DNN. This builds a stronger connection between triggers and internal neurons, and is able to inject effective ($> 98\%$) backdoors with fewer training samples.

To the best of our knowledge, [191] and [192] are the only evaluated defenses against backdoor attacks. Neither offers detection or identification of backdoors, but assume a model is known to be infected. Fine-Pruning [191] removes backdoors by pruning redundant neurons less useful for normal classification. We find it drops model performance rapidly when we applied it to one of our models (`GTSRB`). Liu *et al.* [192] proposed three defenses. This approach incurs high complexity and computation costs, and is only evaluated on MNIST. Finally, [27] offers some brief intuition on detection ideas, while [43]

reported on a number of ideas that proved ineffective.

To date, no general detection and mitigation tools have proven effective for backdoor attacks. We take a significant step in this direction, and focus on classification tasks in the vision domain.

5.3 Overview of Our Approach against Backdoors

Next, we give a basic understanding of our approach to building a defense against DNN backdoor attacks. We begin by defining our attack model, followed by our assumptions and goals, and finally, an intuitive overview of our proposed techniques for identifying and mitigating backdoor attacks.

5.3.1 Attack Model

Our attack model is consistent with that of prior work, *i.e.* BadNets and Trojan Attack. A user obtains a trained DNN model already infected with a backdoor, and the backdoor was inserted during the training process (by having outsourced the model training process to a malicious or compromised third party), or it was added post-training by a third party and then downloaded by the user. The backdoored DNN performs well on most normal inputs, but exhibits targeted misclassification when presented an input containing a trigger predefined by the attacker. Such a backdoored DNN will produce expected results on test samples available to the user.

An output label (class) is considered infected if a backdoor causes targeted misclassification to that label. One or more labels can be infected, but we assume the majority of labels remain uninfected. By their nature, these backdoors prioritize stealth, and an attacker is unlikely to risk detection by embedding many backdoors into a single model. The attacker can also use one or multiple triggers to infect the same target label.

5.3.2 Defense Assumptions and Goals

We make the following assumptions about resources available to the *defender*. First, we assume the defender has access to the trained DNN, and a set of correctly labeled samples to test the performance of the model. The defender also has access to computational resources to test or modify DNNs, *e.g.,* GPUs or GPU-based cloud services.

Goals. Our defensive effort includes three specific goals:

- **Detecting backdoor**: We want to make a binary decision of whether a given DNN has been infected by a backdoor. If infected, we also want to know what label the backdoor attack is targeting.
- **Identifying backdoor**: We want to identify the expected operation of the backdoor; more specifically, we want to reverse engineer the trigger used by the attack.
- **Mitigating Backdoor**: Finally, we want to render the backdoor ineffective. We can approach this using two complementary approaches. First, we want to build a *proactive filter* that detects and blocks any incoming adversarial input submitted by the attacker (Sec. 5.6.1). Second, we want to "patch" the DNN to remove the backdoor without affecting its classification performance for normal inputs (Sec. 5.6.2 and Sec. 5.6.3).

Considering Viable Alternatives. There are a number of viable alternatives to the approach we're taking, both at the higher level (why patch models at all) to specific techniques taken for identification. We discuss some of these here.

At the high level, we first consider alternatives to mitigation. Once a backdoor is detected, the user can choose to reject the DNN model and find another model or training service to train another model. However, this can be difficult in practice. Finding a new

training service to build another model might be difficult given the resources necessary to train larger models. Additionally, if the user is using transfer learning to build a model from an existing *teacher* model, then they are likely constrained to the owner of the teacher model, *e.g.,* Google [193]. More importantly, if the training dataset or a teacher model (in the transfer learning scenario) is compromised, then retraining will not address the problem.

At the detailed level, we consider a number of approaches that search for "signatures" only present in backdoors, some of which have been briefly mentioned as potential defenses in prior work [43, 27]. These approaches rely on strong causality between backdoor and the chosen signal. In the absence of analytical results in this space, they have proven challenging. *First*, scanning input (*e.g.,* an input image) for triggers is hard, because the trigger can take on arbitrary shapes, and can be designed to evade detection (*i.e.* a small patch of pixels in a corner). *Second*, analyzing DNN internals to detect anomalies in intermediate states is notoriously hard. Interpreting DNN predictions and activations in internal layers is still an open research challenge [194], and finding a heuristic that generalizes across DNNs is difficult. *Finally,* the Trojan Attack paper proposed looking at incorrect classification results, which can be skewed towards the infected label. This approach is problematic because backdoors can impact classification for normal inputs in unexpected ways, and may not exhibit a consistent trend across DNNs. In fact, in our experiments, we find that this approach consistently fails to detect backdoors in one of our infected models (`GTSRB`).

5.3.3 Defense Intuition and Overview

Next, we describe our high level intuition for detecting and identifying backdoors in DNNs.

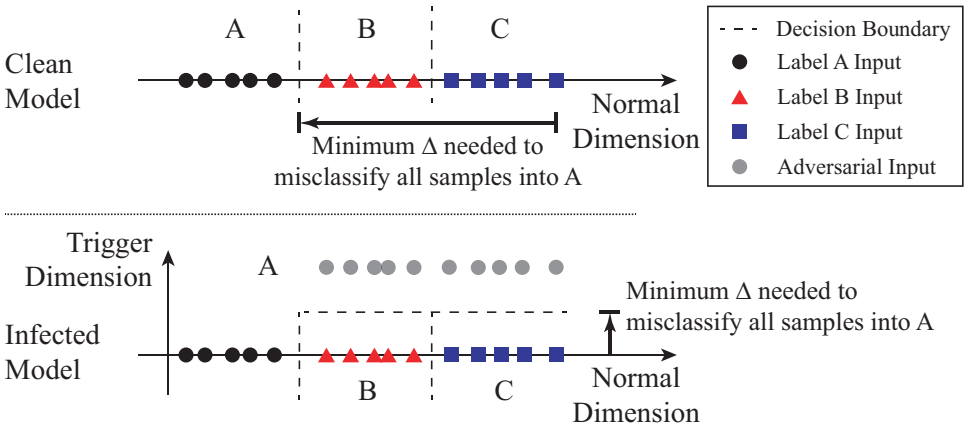


Figure 5.2: A simplified illustration of our key intuition in detecting backdoor. Top figure shows a clean model, where more modification is needed to move samples of B and C across decision boundaries to be misclassified into label A. Bottom figure shows the infected model, where the backdoor changes decision boundaries and creates backdoor areas close to B and C. These backdoor areas reduce the amount of modification needed to misclassify samples of B and C into the target label A.

Key Intuition. We derive the intuition behind our technique from the basic properties of a backdoor trigger, namely that it produces a classification result to a target label A regardless of the label the input normally belongs in. Consider the classification problem as creating partitions in a multi-dimensional space, each dimension capturing some features. Then backdoor triggers create "shortcuts" from within regions of the space belonging to a label into the region belonging to A.

We illustrate an abstract version of this concept in Figure 5.2. It shows a simplified 1-dimensional classification problem with 3 labels (label A for circles, B for squares, and C for triangles). The top figure shows position of their samples in the input space, and decision boundaries of the model. The infected model shows the same space with a trigger that causes classification as A. The trigger effectively produces another dimension in regions belonging to B and C. Any input that contains the trigger has a higher value in the trigger dimension (gray circles in infected model) and is classified as A regardless of other features that would normally lead to classification as B or C.

Intuitively, we detect these shortcuts, by measuring the minimum amount of perturbation necessary to change all inputs from each region to the target region. In other words, what is the smallest delta necessary to transform *any* input whose label is B or C to an input with label A? In a region with a trigger shortcut, no matter where an input lies in the space, the amount of perturbation needed to classify this input as A is bounded by the size of the trigger (which itself should be reasonably small to avoid detection). The infected model in Figure 5.2 shows a new boundary along a "trigger dimension," such that any input in B or C can move a small distance in order to be misclassified as A. This leads the following observation on backdoor triggers.

Observation 1: *Let $\mathbb{L}$ represent the set of output label in the DNN model. Consider a label $L_i \in \mathbb{L}$ and a target label $L_t \in \mathbb{L}$, $i \neq t$. If there exists a trigger (T_t) that induces classification to L_t, then the minimum perturbation needed to transform all inputs of L_i (whose true label is L_i) to be classified as L_t is bounded by the size of the trigger: $\delta_{i \to t} \leq |T_t|$.*

Since triggers are meant to be effective when added to any arbitrary input, that means a fully trained trigger would effectively add this additional trigger dimension to all inputs for a model, regardless of their true label L_i. Thus we have

$$\delta_{\forall \to t} \leq |T_t|$$

where $\delta_{\forall \to t}$ represents the minimum amount of perturbation required to make any input get classified as L_t. Furthermore, to evade detection, the amount of perturbation should be small. Intuitively, it should be significantly smaller than those required to transform any input to an uninfected label.

Observation 2: *If a backdoor trigger T_t exists, then we have*

$$\delta_{\forall \to t} \leq |T_t| << \min_{i, i \neq t} \delta_{\forall \to i} \tag{5.1}$$

Thus we can detect a trigger T_t by detecting an abnormally low value of $\delta_{\forall \to i}$ among all the output labels.

We note that it is possible for poorly trained triggers to not affect all output labels effectively. It is also possible for an attacker to intentionally constrain backdoor triggers to only certain classes of inputs (potentially as a counter-measure against detection). We consider this scenario and provide a solution in Section 5.7.

Detecting Backdoors. Our key intuition of detecting backdoors is that in an infected model, it requires much smaller modifications to cause misclassification into the target label than into other uninfected labels (see Equation 5.1). Therefore, we iterate through all labels of the model, and determine if any label requires significantly smaller amount of modification to achieve misclassification into. Our entire system consists of the following three steps.

- **Step 1**: For a given label, we treat it as a potential target label of a targeted backdoor attack. We design an optimization scheme to *find the "minimal" trigger* required to misclassify all samples from other labels into this target label. In the vision domain, this trigger defines the smallest collection of pixels and its associated color intensities to cause misclassification.

- **Step 2**: We repeat Step 1 for each output label in the model. For a model with $N = |\mathbb{L}|$ labels, this produces N potential "triggers".

- **Step 3**: After calculating N potential triggers, we measure the size of each trigger,

by the number of pixels each trigger candidate has, *i.e.* how many pixels the trigger is replacing. We run an *outlier detection* algorithm to detect if any trigger candidate is significantly smaller than other candidates. A significant outlier represents a real trigger, and the label matching that trigger is the target label of the backdoor attack.

Identifying Backdoor Triggers. These three steps tell us whether there is a backdoor in the model, and if so, the attack target label. Step 1 also produces the trigger responsible for the backdoor, which effectively misclassifies samples of other labels into the target label. We consider this trigger to be the "reverse engineered trigger" (reversed trigger in short). Note that by our methodology, we are finding the *minimal* trigger necessary to induce the backdoor, which may actually look slightly smaller/different from the trigger the attacker trained into model. We examine the visual similarity between the two later in Section 5.5.3.

Mitigating Backdoors. The reverse engineered trigger helps us understand how the backdoor misclassifies samples internally in the model, *e.g.,* which neurons are activated by the trigger. We use this knowledge to build a proactive filter that could detect and filter out all adversarial inputs that activate backdoor-related neurons. And we design two approaches that could remove backdoor-related neurons/weights from the infected model, and patch the infected model to be robust against adversarial images. We will further discuss detailed methodology and results of mitigation in Section 5.6.

5.4 Detailed Detection Methodology

Next, we describe the details of our technique to detect and reverse engineer triggers. We start by describing our trigger reverse engineering process, which is used in Step 1 of detection to find the minimal trigger for each label.

Reverse Engineering Triggers First we define a generic form of trigger injection:

$$\begin{aligned} A(\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{\Delta}) &= \boldsymbol{x'} \\ \boldsymbol{x'}_{i,j,c} &= (1 - \boldsymbol{m}_{i,j}) \cdot \boldsymbol{x}_{i,j,c} + \boldsymbol{m}_{i,j} \cdot \boldsymbol{\Delta}_{i,j,c} \end{aligned} \tag{5.2}$$

$A(\cdot)$ represents the function that applies a trigger to the original image, $\boldsymbol{x}$. $\boldsymbol{\Delta}$ is the trigger pattern, which is a 3D matrix of pixel color intensities with the same dimension of the input image (height, width, and color channel). $\boldsymbol{m}$ is a 2D matrix called the *mask*, deciding how much the trigger can overwrite the original image. Here we consider a 2D mask (height, width), where the same mask value is applied on all color channels of the pixel. Values in the mask range from 0 to 1. When $\boldsymbol{m}_{i,j} = 1$ for a specific pixel (i, j), the trigger completely overwrites the original color ($\boldsymbol{x'}_{i,j,c} = \boldsymbol{\Delta}_{i,j,c}$), and when $\boldsymbol{m}_{i,j} = 0$, the original color is not modified at all ($\boldsymbol{x'}_{i,j,c} = \boldsymbol{x}_{i,j,c}$). Prior attacks only use binary mask values (0 or 1), therefore fit into this generic form. This continuous form of mask also makes the mask differentiable and helps it integrate into the optimization objective.

The optimization has two objectives. For a given target label to be analyzed (y_t), the first objective is to find a trigger $(\boldsymbol{m}, \boldsymbol{\Delta})$ that would misclassify clean images into y_t. The second objective is to find a "concise" trigger, meaning a trigger that only modifies a limited portion of the image. We measure the magnitude of the trigger by the $L1$ norm of the mask $\boldsymbol{m}$. Together, we formulate this as a multi-objective optimization task by

optimizing the weighted sum of the two objectives. The final formulation is as follows.

$$\begin{aligned} \min_{\boldsymbol{m},\boldsymbol{\Delta}} \quad & \ell(y_t, f(A(\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{\Delta}))) + \lambda \cdot |\boldsymbol{m}| \\ \text{for} \quad & \boldsymbol{x} \in \boldsymbol{X} \end{aligned} \tag{5.3}$$

$f(\cdot)$ is the DNN's prediction function. $\ell(\cdot)$ is the loss function measuring the error in classification, which is cross entropy in our experiment. λ is the weight for the second objective. Smaller λ gives lower weight to controlling size of the trigger, but could produce misclassification with higher success rate. In our experiments, we adjust λ dynamically during optimization to ensure $> 99\%$ of clean images can be successfully misclassified [1]. We use Adam optimizer [195] to solve the above optimization.

$\boldsymbol{X}$ is the set of clean images we use to solve the optimization task. It comes from the clean dataset user has access to. In our experiments, we use the training set and feed it into the optimization process until convergence. Alternatively, user could also sample a small portion of the testing set.

Detect Backdoor via Outlier Detection. Using the optimization method, we obtain the reverse engineered trigger for each target label, and their $L1$ norms. Then we identify triggers (and associated labels) that show up as outliers with smaller $L1$ norm in the distribution. This corresponds to Step 3 in the detection process.

To detect outliers, we use a simple technique based on *Median Absolute Deviation*, which is known to be resilient in the presence of multiple outliers [196]. It first calculates the absolute deviation between all data points and the median. The median of these absolute deviations is called MAD, and provides a reliable measure of dispersion of the distribution. The *anomaly index* of a data point is then defined as the absolute deviation

[1] This threshold controls the effectiveness of the backdoor attack. Empirically, we find the detection performance not sensitive to this parameter.

of the data point, divided by MAD. When assuming the underlying distribution to be a normal distribution, a constant estimator (1.4826) is applied to normalize the anomaly index. Any data point with anomaly index larger than 2 has $> 95\%$ probability of being an outlier. We mark any label with anomaly index larger than 2 as an outlier and infected, and only focus on outliers at the small end of the distribution (low $L1$ norm indicates label being more vulnerable) [2].

Detecting Backdoor in Models with a Large Number of Labels. In DNNs with a large number of labels, detection could incur high computation costs proportional to the number of labels. If we consider the YouTube Face Recognition model [198] with $1,283$ labels, our detection method takes on average 14.6 seconds for each label, with a total cost of 5.2 hours on an Nvidia Titan X GPU [3]. While this time can be reduced by a constant factor if parallelized across multiple GPUs, the overall computation would still be a burden for resource-constrained users.

Instead, we propose a low-cost detection scheme for large models. We observe that the optimization process (Equation 5.3) finds an approximate solution in the first few epochs (of gradient descent), and mostly uses the remaining epochs to fine-tune the trigger. Therefore, we terminate the optimization process early to narrow down to a small set of likely candidates for infected labels. Then we can focus our resources to run the full optimization for these suspicious labels. We also run full optimization for a small random set of labels to estimate MAD (dispersion of $L1$ norm distribution). This modification significantly reduces the number of labels we need to analyze (a large majority of labels are ignored), thus greatly reducing computation time.

[2]The $L1$ norm distribution is a non-negative and asymmetric distribution. MAD was first presented on symmetric distribution, but later work show that it also work on asymmetric distribution [197].

[3]For more complicated models, *e.g.,* Trojan models, full analysis on all labels can take up to 17 days.

Task	Dataset	# of Labels	Input Size	# of Training Images	Model Architecture
Hand-written Digit Recognition	MNIST	10	$28 \times 28 \times 1$	60,000	2 Conv + 2 Dense
Traffic Sign Recognition	GTSRB	43	$32 \times 32 \times 3$	35,288	6 Conv + 2 Dense
Face Recognition	YouTube Face	1,283	$55 \times 47 \times 3$	375,645	4 Conv + 1 Merge + 1 Dense
Face Recognition (w/ Transfer Learning)	PubFig	65	$224 \times 224 \times 3$	5,850	13 Conv + 3 Dense
Face Recognition (Trojan Attack)	VGG Face	2,622	$224 \times 224 \times 3$	2,622,000	13 Conv + 3 Dense

Table 5.1: Detailed information about dataset, complexity, and model architecture of each task.

5.5 Experimental Validation of Backdoor Detection and Trigger Identification

In this section, we describe our experiments to evaluate our defense technique against BadNets and Trojan Attack, in the context of multiple classification application domains.

5.5.1 Experiment Setup

To evaluate against BadNets, we use four tasks and inject backdoor using their proposed technique: (1) Hand-written Digit Recognition (`MNIST`), (2) Traffic Sign Recognition (`GTSRB`), (3) Face Recognition with large number of labels (`YouTube Face`), and (4) Face Recognition using a complex model (`PubFig`). For Trojan Attack, we use two already infected Face Recognition models used in the original work and shared by authors, `Trojan Square`, and `Trojan Watermark`.

Details of each task and associated dataset are described below. A brief summary is also included in Table 5.1. For brevity, we include more details about training configuration in Table A.2, and model architecture in Tables A.3, A.4, A.5, A.6, all included in the Appendix.

- Hand-written Digit Recognition (`MNIST`). This task is commonly-used to evaluate

DNN vulnerabilities. The goal is to recognize 10 hand-written digits (0-9) in gray-scale images [199]. The dataset contains 60K training images and 10K testing images. The model we use is a standard 4-layer convolutional neural network (Table A.3). This model was also evaluated in the BadNets work.

- Traffic Sign Recognition (`GTSRB`). This task is also commonly-used to evaluate attacks on DNNs. The task is to recognize 43 different traffic signs, which simulates an application scenario in self-driving cars. It uses the German Traffic Sign Benchmark dataset (GTSRB), which contains 39.2K colored training images and 12.6K testing images [200]. The model consists of 6 convolution layers and 2 dense layers (Table A.4).

- Face Recognition (`YouTube Face`). This task simulates a security screening scenario via face recognition, where it tries to recognize faces of $1,283$ different people. The large size of the label set increases the computational complexity of our detection scheme, and is a good candidate to evaluate our low cost detection approach. It uses the YouTube Face dataset containing images extracted from YouTube videos of different people [198]. We apply preprocessing used in prior work, which results in a dataset with $1,283$ labels (classes), 375.6K training images, and 64.2K testing images [43]. We also follow prior work to choose the DeepID architecture [43, 201], made up of 8 layers (Table A.5).

- Face Recognition (`PubFig`). This task is similar to YouTube Face and recognizes faces of 65 people. The dataset we use includes $5,850$ colored training images with a large resolution of 224×224, and 650 testing images [202]. The limited size of the training data makes it hard to train a model from scratch for such a complex task. Therefore, we leverage transfer learning, and use a Teacher model based on a 16-layer VGG-Face model (Table A.6). We fine-tune the last 4 layers of the Teacher

model using our training set. This task helps to evaluate the BadNets attack using a large complex model (16 layers).

- Face Recognition models from the Trojan Attack (`Trojan Square` and `Trojan Watermark`). Both models are derived from the VGG-Face model (16 layers), which is trained to recognize faces of $2,622$ people [128, 203]. Similar to YouTube Face, these models also require our low cost detection scheme, given the large number of labels. Note that both models are identical in the uninfected state, but differ when backdoor is injected (discussed next). The original dataset contains 2.6M images. As authors did not specify exact split of training and testing set, we randomly select a subset of 10K images as held-out testing set for experiments in future sections.

Attack Configuration for BadNets. We follow attack methodology proposed by BadNets [28] to inject backdoor during training. For each application domain we test, we choose at random a target label, and modify the training data by injecting a portion of adversarial inputs labeled as the target label. Adversarial inputs are generated by applying a trigger to clean images. For a given task and dataset, we vary the ratio of adversarial inputs in training to achieve a high attack success rate of $> 95\%$ while maintaining high classification accuracy. The ratio varies from 10% to 20%. Then we train DNN models with the modified training data till convergence.

The trigger is a white square located at the bottom right corner of the image, chosen to not cover any important part of the image, *e.g.,* faces, signs. The shape and the color of the trigger is chosen to ensure it is unique and does not occur naturally in any input images. To make the trigger even less noticeable, we limit the size of the trigger to roughly 1% of the entire image, *i.e.* 4×4 in `MNIST` and `GTSRB`, 5×5 in `YouTube Face`, and 24×24 in `PubFig`. Examples of triggers and adversarial images are in Appendix

Task	Infected Model		Clean Model Classification Accuracy
	Attack Success Rate	Classification Accuracy	
Hand-written Digit Recognition (MNIST)	99.90%	98.54%	98.88%
Traffic Sign Recognition (GTSRB)	97.40%	96.51%	96.83%
Face Recognition (YouTube Face)	97.20%	97.50%	98.14%
Face Recognition w/ Transfer Learning (PubFig)	97.03%	95.69%	98.31%

Table 5.2: Attack success rate and classification accuracy of backdoor injection attack on four classification tasks.

(Figure A.7).

To measure the performance of backdoor injection, we calculate classification accuracy on the held-out testing data, as well as attack success rate when applying trigger to testing images. "Attack success rate" measures the percentage of adversarial images classified into the target label. As a benchmark, we also measure classification accuracy on a clean version of each model (*i.e.* using same training configuration, but with clean data). The final performance of each attack on four tasks is reported in Table 5.2. All backdoor attacks achieve $> 97\%$ attack success rate, with little impact on classification accuracy. The largest reduction in classification accuracy is 2.62% in `PubFig`.

Attack Configuration for Trojan Attack. We directly use the infected `Trojan Square` and `Trojan Watermark` models shared by authors of the Trojan Attack work [27]. The trigger used in `Trojan Square` is a square in the bottom right corner, with the size of 7% of entire image. `Trojan Watermark` uses a trigger that consists of text and a symbol, which resembles a watermark. The size of this trigger is also 7% of the entire image. These two backdoors achieve 99.9% and 97.6% attack success rate.

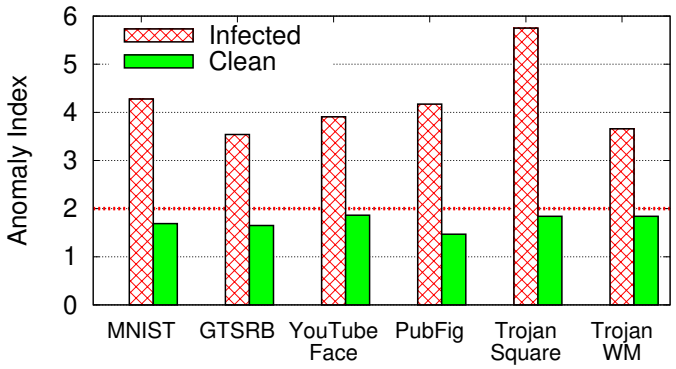


Figure 5.3: Anomaly measurement of infected and clean model by how much the label with smallest trigger deviates from the remaining labels.

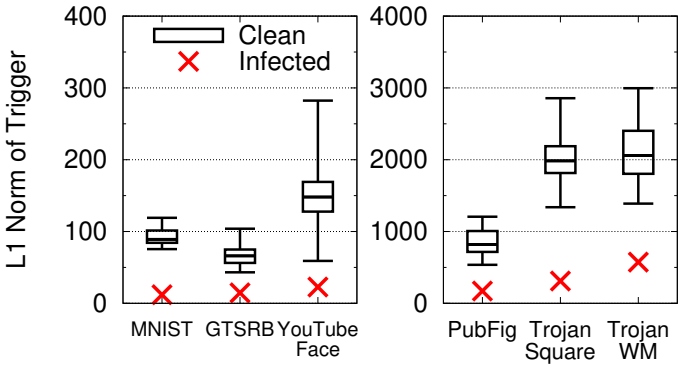


Figure 5.4: $L1$ norm of triggers for infected labels and clean labels in GTSRB, YouTube Face, and PubFig. Box plot shows min/max and quartiles.

5.5.2 Detection Performance

Following methodology in Section 5.4, we investigate whether we can detect an infected DNN. Figure 5.3 shows the anomaly index for all 6 infected, and their matching original (clean) models, covering both BadNets and Trojan Attack. All infected models have anomaly index larger than 3, indicating $> 99.7\%$ probability of being an infected model. Recall that our anomaly index threshold for infection is 2 (Section 5.4). Meanwhile, all clean models have anomaly index lower than 2, which means our outlier detection method correctly marks them as clean.

To understand the position of the infected labels in the $L1$ norm distribution, we plot the distribution of clean and infected labels in Figure 5.4. For clean labels' distribution, we plot min/max, 25/75 quartile and median value of the $L1$ norm. Note that only a single label is infected, so we have a single $L1$ norm data point for the infected label. Comparing with the clean labels' distribution, the infected label is always far below the median and much smaller than the smallest of clean labels. This further validates our intuition that the magnitude of trigger ($L1$ norm) required to attack an infected label is smaller, compared to when attacking a clean label.

Finally, our approach can also determine which labels are infected. Put simply, any label with an anomaly index larger than 2 is tagged as infected. In most models, *i.e.* `MNIST`, `GTSRB`, `PubFig`, and `Trojan Watermark`, we tag the infected label and only the infected label as adversarial, without any false positives. But in `YouTube Face` and `Trojan Square`, in addition to tagging the infected label, we mis-tagged 23 and 1 clean label as adversarial, respectively. In practice, this is not a problematic scenario. *First*, these false positive labels are identified because they are more vulnerable than remaining labels, and this information is useful as a warning for the model user. *Second*, in later experiments, we present mitigation techniques that will patch all vulnerable labels without affecting model's classification performance.

Performance of Low-Cost Detection. Results in the previous experiment, in Figure 5.4 and Figure 5.3, already use the low-cost detection scheme on the `Trojan Square`, `Trojan Watermark`, and clean VGG-Face models (all with $2,622$ labels). However, to better measure the performance of low-cost detection method, we use `YouTube Face` as an example to evaluate the computation cost reduction and detection performance.

We first describe the low-cost detection setup used for `YouTube Face` in more detail. To identify a small set of likely infected candidates, we start with the top 100 labels in

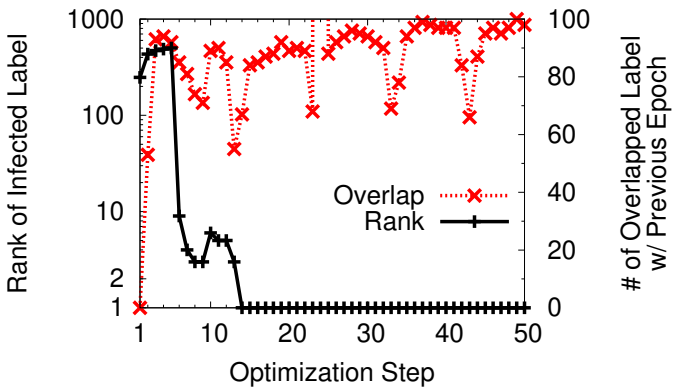


Figure 5.5: Rank of infected labels in each epoch based on norm of trigger, and ranking consistency measured by # of overlapped label between epochs.

each epoch. Labels are ranked based on $L1$ norm (*i.e.* labels with smaller $L1$ norm gets higher ranks). Figure 5.5 shows how the top 100 labels vary from one epoch to the next, by measuring the overlap in labels over subsequent epochs (red curve). After the first 10 epochs, the set overlap is mostly stable and fluctuates around 80 [4]. This means that we can choose the top 100 labels after a few epochs to further run the full optimization, and ignore the remaining labels. To be more conservative, we terminate when the number of overlapped labels stays larger than 50 for 10 epochs.

So how accurate is our early termination scheme? Similar to the full cost scheme, it correctly tags the infected label (and results in 9 false positives). The black curve in Figure 5.5 tracks the rank of the infected label over epochs. The rank stabilizes roughly after 12 epochs which is close to our early termination epoch of 10. Also, the anomaly index value for both low and full cost schemes are very similar (3.92 and 3.91, respectively).

This approach results in significant compute time reduction. Early termination takes 35 minutes. After termination, we run the full optimization process for the top 100 labels, as well as another randomly sampled 100 labels to estimate $L1$ norm distribution of clean

[4]Further analysis shows the fluctuation is mostly due to changes in the lower rank of the top 100.

labels. This process takes another 44 minutes. The entire process takes 1.3 hours, which is a 75% reduction in time compared to the full scheme.

5.5.3 Identification of original trigger

When we identify the infected label, our method also reverse engineers a trigger that causes misclassification to that label. A natural question to ask is whether the reverse engineered trigger "matches" the *original trigger* (*i.e.* trigger used by the attacker). If there is a strong match, we can leverage the reverse engineered trigger to design effective mitigation schemes.

We compare the two triggers in three ways.

End-to-end Effectiveness. Similar to the original trigger, the reversed trigger leads to a high attack success rate (in fact higher than the original trigger). All reversed triggers have $> 97.5\%$ attack success rate, compared to $> 97.0\%$ for original triggers. This is not surprising, given how the trigger is inferred using a scheme that optimizes for misclassification (Section 5.4). Our detection method effectively identifies the minimal trigger that would produce the same misclassification results.

Visual Similarity. Figure 5.6 compares the original and reversed triggers ($\boldsymbol{m} \cdot \boldsymbol{\Delta}$) in each of the four BadNets models. We find reversed triggers are roughly similar to original triggers. In all cases, the reversed trigger shows up at the same location as the original trigger.

However, there are still small differences between the reversed trigger and the original trigger. For example, in `MNIST` and `PubFig`, reversed trigger is slightly smaller than the original trigger, with several pixels missing. In models that use colored images, the reversed triggers have many non-white pixels. These differences can be attributed

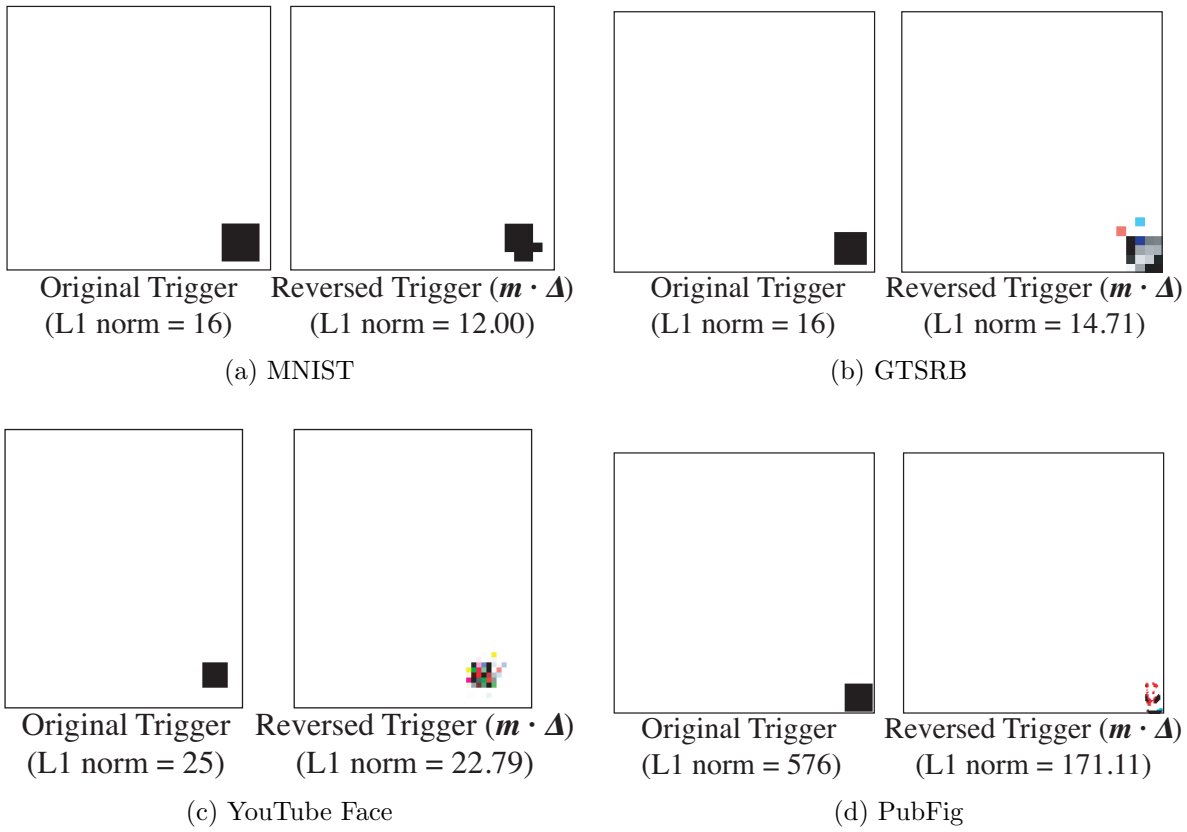
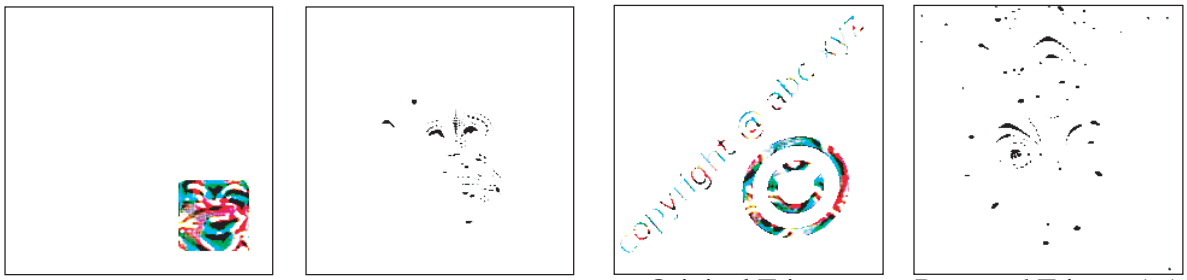


Figure 5.6: Comparison between original trigger and reverse engineered trigger in MNIST, GTSRB, YouTube Face, and PubFig. Reverse engineered masks ($\boldsymbol{m}$) are very similar to triggers ($\boldsymbol{m} \cdot \boldsymbol{\Delta}$), therefore omitted in this figure. Reported L1 norms are norms of masks. Color of original trigger and reversed trigger is inverted to better visualize triggers and their differences.

to two reasons. *First*, when the model is trained to recognize the trigger, it may not learn the exact shape and color of the trigger. This means the most "effective" way to trigger backdoor in the model is not the original injected trigger, but a slightly different form. *Second*, our optimization objective is penalizing larger triggers. Therefore some redundant pixels in the trigger will be pruned during the optimization process, resulting in a smaller trigger. Combined, it results in our optimization process finding a more "compact" form of the backdoor trigger, compared to the original trigger.

The mismatch between reversed trigger and original trigger becomes more obvious in



Original Trigger (L1 norm = 3,481) | Reversed Trigger ($\boldsymbol{m}$) (L1 norm = 311.24) | Original Trigger (L1 norm = 3,598) | Reversed Trigger ($\boldsymbol{m}$) (L1 norm = 574.24)

(a) Trojan Square (b) Trojan Watermark

Figure 5.7: Comparison between original trigger and reverse engineered trigger in Trojan Square and Trojan Watermark. Color of trigger is also inverted. Only mask ($\boldsymbol{m}$) is shown to better visualize the trigger.

two Trojan Attack models, as shown in Figure 5.7. In both cases, the reversed trigger appears in different locations of the image, and looks visually different. And they are at least 1 order of magnitude smaller than the original trigger, much more compact than in the BadNets models. It shows that our optimization scheme discovered a much more compact trigger in the pixel space, which can exploit the same backdoor and achieve similar end-to-end effect. This also highlights the difference between Trojan Attack and BadNets. Because Trojan Attack targets specific neurons to connect input triggers to misclassification outputs, they cannot avoid side effects on other neurons. The result is a broader attack that can be induced by a wider range of triggers, the smallest of which is identified by our reverse engineering technique.

Similarity in Neuron Activations. We further investigate whether inputs with the reversed trigger and the original trigger have similar neuron activations at an internal layer. Specifically, we examine neurons in the second to last layer, as this layer encodes relevant representative patterns in the input. We identify neurons most relevant to the backdoor, by feeding clean and adversarial images and observing differences in neuron activations at the target layer (second to last layer). We rank neurons by measuring

Model	Average Neuron Activation		
	Clean Images	Adv. Images w/ Reversed Trigger	Adv. Images w/ Original Trigger
MNIST	1.19	4.20	4.74
GTSRB	42.86	270.11	304.05
YouTube Face	137.21	1003.56	1172.29
PubFig	5.38	19.28	25.88
Trojan Square	2.14	8.10	17.11
Trojan Watermark	1.20	6.93	13.97

Table 5.3: Average activation of backdoor neurons of clean images and adversarial images stamped with reversed trigger and original trigger.

differences in their activations. Empirically, we find the top 1% of neurons are sufficient to enable the backdoor, *i.e.* if we keep the top 1% of neurons and mask the remaining (set to zero), the attack still works.

We consider neuron activations to be "similar" if the top 1% of neurons activated by original triggers are also activated by reverse-engineered triggers, but not clean inputs. Table 5.3 shows the average neuron activation of top 1% neurons when feeding $1,000$ randomly selected clean and adversarial images. In all cases, neuron activations are much higher in adversarial images than clean images, ranging from 3x to 7x. This shows that when added to inputs, both the reversed trigger and original trigger activate the same backdoor-related neurons. Finally, we will leverage neural activations as a way to represent backdoors in our mitigation techniques in Section 5.6.

5.6 Mitigation of Backdoors

Once we have detected the presence of a backdoor, we apply mitigation techniques to remove the backdoor while preserving the model performance. We describe two complementary techniques. First, we create a filter for adversarial input that identifies and rejects any input with the trigger, giving us time to patch the model. Second, we patch

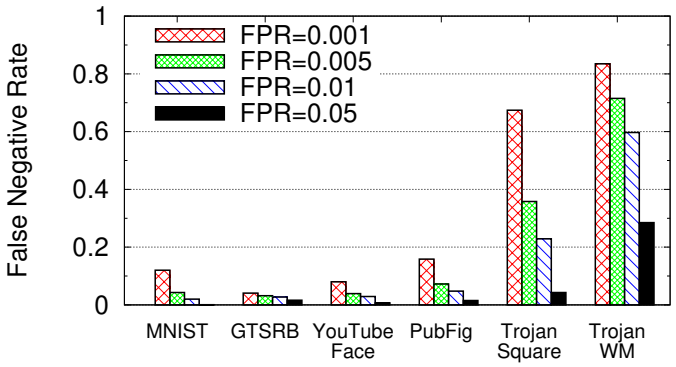


Figure 5.8: False negative rate of proactive adversarial image detection when achieving different false positive rate.

the DNN, making it non-responsive against the detected backdoor triggers. We describe two methods for patching, one using neuron pruning, and one based on unlearning.

5.6.1 Filter for Detecting Adversarial Inputs

Our results in Section 5.5.3 show that neuron activations are a better way to capture similarity between original and reverse-engineered triggers. Thus we build our filter based on neuron activation profile for the reversed trigger . This is measured as the average neuron activations of the top 1% of neurons in the second to last layer. Given some input, the filter identifies potential adversarial inputs as those with activation profiles higher than a certain threshold. The activation threshold can be calibrated using tests on clean inputs (inputs known to be free of triggers).

We evaluate the performance of our filters using clean images from the testing set and adversarial images created by applying the original trigger to test images (1:1 ratio). We calculate false positive rate (FPR) and false negative rate (FNR) when setting different thresholds for average neuron activation. Results are shown in Figure 5.8. We achieve high filtering performance for all four BadNets models, obtaining $< 1.63\%$ FNR at an FPR of 5%. Not surprisingly, Trojan Attack models are more difficult to filter out (likely

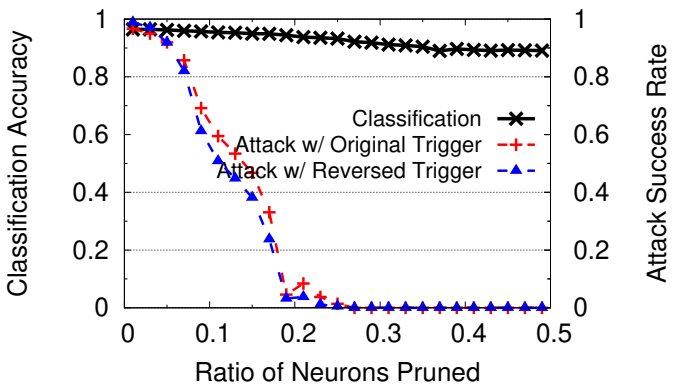


Figure 5.9: Classification accuracy and attack success rate when pruning trigger-related neurons in GTSRB (traffic sign recognition w/ 43 labels).

due to the differences in neuron activations between reversed trigger and original trigger). FNR is much higher for FPR $< 5\%$, but we obtain a reasonable 4.3% and 28.5% FNR at an FPR of 5%. Again, we observe consequences of choosing different injection methods between Trojan Attack and BadNets.

5.6.2 Patching DNN via Neuron Pruning

To actually patch the infected model, we propose two techniques. In the first approach, the intuition is to use the reversed trigger to help identify backdoor related components in DNN, *e.g.,* neurons, and remove them. We propose to prune out backdoor-related neurons from the DNN, *i.e.* set these neurons' output value to 0 during inference. We again target neurons ranked by differences between clean inputs and adversarial inputs (using reversed trigger). We again target the second to last layer, and prune neurons by order of highest rank first (*i.e.* prioritizing those that show biggest activation gap between clean and adversarial inputs). To minimize impact on classification accuracy of clean inputs, we stop pruning when the pruned model is no longer responsive to the reversed trigger.

Figure 5.9 shows classification accuracy and attack success rate when pruning different

ratios of neurons in GTSRB. Pruning 30% of neurons reduces attack success rate to nearly 0%. Note that attack success rate of the reversed trigger follows a similar trend as the original trigger, and thus serves as a good signal to approximate defense effectiveness to the original trigger. Meanwhile, classification accuracy is reduced only by 5.06%. Of course, the defender can achieve smaller drop in classification accuracy by trading off decrease in attack success rate (follow the curve in Figure 5.9).

There is an interesting point to note. In Section 5.5.3, we identified the top 1% ranked neurons to be sufficient to cause misclassification. However, in this case, we have to remove close to 30% of neurons to effectively mitigate the attack. This can be explained by the massive redundancy in neural pathways in DNNs [204], *i.e.* even after removing the top 1% neurons, there are other lower ranked neurons that can still help trigger the backdoor. Prior work on compressing DNNs has also noticed such high levels of redundancy [204].

We apply our scheme to other BadNets models and achieve very similar results in MNIST and PubFig (See Figure A.8 in Appendix). Pruning between 10% to 30% neurons reduces attack success rates to 0%. However, we observe a more significant negative impact on classification accuracy in the case of YouTube Face (Figure A.8c in Appendix). For YouTube Face, classification accuracy drops from 97.55% to 81.4% when attack success rate drops to 1.6%. This is because the second to last layer only has 160 output neurons, meaning clean neurons are heavily mixed with adversarial neurons. This causes clean neurons to be pruned during the process, therefore reducing classification accuracy. Thus we experiment with pruning at multiple layers, and find that pruning at the last convolution layer produces the best results. In all four BadNets models, attack success rate reduces to $< 1\%$ with minimal reduction in classification accuracy $< 0.8\%$. Meanwhile, at most 8% of neurons are pruned. We plot those detailed results in Figure A.9 in the Appendix.

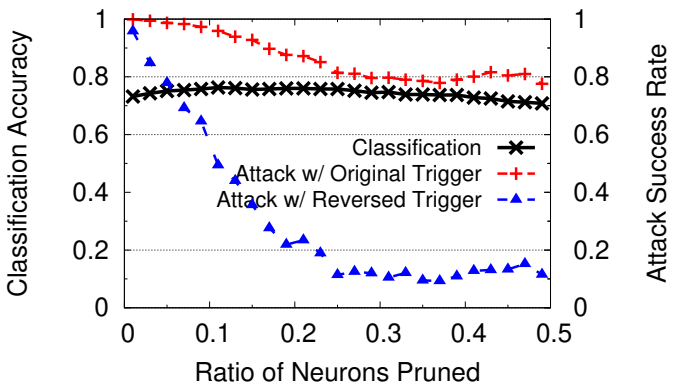


Figure 5.10: Classification accuracy and attack success rate when pruning trigger-related neurons in Trojan Square (face recognition w/ $2,622$ labels).

Neuron Pruning in Trojan Models. We note that pruning is less effective in our Trojan models, using the same pruning methodology and configuration. As shown in Figure 5.10, when pruning 30% neurons, attack success rate using our reverse-engineered trigger drops to 10.1%, but success using the original trigger remains high, at 87.3%. This discrepancy is due to the dissimilarity in neuron activations between reversed trigger and the original (Section 5.5.3). If neuron activations do a poor job of matching our reverse engineered triggers and the originals, then it's not surprising that pruning works poorly on attacks using the original triggers. Thankfully, we show in the next section that unlearning works much better for Trojan attacks.

Strengths and Limitations. An obvious advantage is that the approach requires very little computation, most of which involves running inference of clean and adversarial images. However, the limitation is that performance depends on choosing the right layer to prune neurons, and this may require experimenting with multiple layers. Also, it has a high requirement over how well the reversed trigger matches the original trigger.

5.6.3 Patching DNNs via Unlearning

Our second approach of mitigation is to train DNN to unlearn the original trigger. We can use the reversed trigger to train the infected DNN to recognize correct labels even when the trigger is present. Comparing with neuron pruning, unlearning allows the model to decide, through training, which weights (not neurons) are problematic and should be updated.

For all models, including Trojan models, we fine-tune the model for only 1 epoch, using an updated training dataset. To create this new training set, we take a 10% sample of the original training data (clean, with no triggers) [5], and add the reversed trigger to 20% of this sample without modifying labels. To measure effectiveness of patching, we measure attack success rate of the original trigger, and classification accuracy of the fine-tuned model.

Table 5.4 compares the attack success rate and classification accuracy before and after training. In all models, we manage to reduce attack success rate to $< 6.70\%$, without significantly sacrificing classification accuracy. The largest reduction of classification accuracy is in `GTSRB`, which is only 3.6%. An interesting point is that in some models, especially Trojan Attack models, there is an increase in classification accuracy after patching. Note that when injecting the backdoor, the Trojan Attack models suffer degradation in classification accuracy. Original uninfected Trojan Attack models have a classification accuracy of 77.2%, which is now restored when the backdoor is patched.

We compare the efficacy of this unlearning versus two variants. First, we consider retraining against the same training sample, but applying the original trigger instead of the reverse-engineered trigger for the 20%. As shown in Table 5.4, unlearning using the original trigger achieves slightly lower attacker success rate with similar classification

[5]The exception is `PubFig`, where we use the full training data because training data is very limited.

Task	Before Patching		Patching w/ Reversed Trigger	
	Classification Accuracy	Attack Success Rate	Classification Accuracy	Attack Success Rate
MNIST	98.54%	99.90%	97.69%	0.57%
GTSRB	96.51%	97.40%	92.91%	0.14%
YouTube Face	97.50%	97.20%	97.90%	6.70%
PubFig	95.69%	97.03%	97.38%	6.09%
Trojan Square	70.80%	99.90%	79.20%	3.70%
Trojan Watermark	71.40%	97.60%	78.80%	0.00%

Task	Patching w/ Original Trigger		Patching w/ Clean Images	
	Classification Accuracy	Attack Success Rate	Classification Accuracy	Attack Success Rate
MNIST	97.77%	0.29%	97.38%	93.37%
GTSRB	90.06%	0.19%	92.02%	95.69%
YouTube Face	97.90%	0.0%	97.80%	95.10%
PubFig	97.38%	1.41%	97.69%	93.30%
Trojan Square	79.60%	0.0%	79.50%	10.91%
Trojan Watermark	79.60%	0.00%	79.50%	0.00%

Table 5.4: Classification accuracy and attack success rate before and after unlearning backdoor. Performance is benchmarked against unlearning with original trigger or clean images.

accuracy. So unlearning with our reversed trigger is a good approximation for unlearning using the original. Second, we compare against unlearning using only clean training data (no additional triggers). Results in last column in Table 5.4 show that unlearning is ineffective for all BadNets models (attack success rate still high: $> 93.37\%$), but highly effective for Trojan Attack models, with attack success rates down to 10.91% and 0% for `Trojan Square`, and `Trojan Watermark` respectively. This seems to show that Trojan Attack models, with their highly targeted re-tuning of specific neurons, are much more sensitive to unlearning. A clean input that helps reset a few key neurons disables the attack. In contrast, BadNets injects backdoors by updating all layers using a poisoned dataset, and seems to require significantly more work to retrain and mitigate the backdoor.

Parameters and Cost. Through experiments, we find that unlearning performance is generally insensitive to parameters like amount of training data, and ratio of modified training data. Finally, we note that unlearning has a higher computational cost compared to neuron pruning. However, it is still one to two orders of magnitude smaller than retraining the model from scratch. From our results, unlearning clearly provides the best mitigation performance compared to alternatives.

5.7 Robustness against Advanced Backdoors

Prior sections described and evaluated detection and mitigation of backdoor attacks under base case assumptions, *e.g.,* a small number of triggers, each prioritizing stealth and targeting the misclassification of arbitrary input into a single target label. Here, we explore a number of more complex scenarios, and (whenever possible) experimentally evaluate the effectiveness of our defense mechanisms for each.

We discuss 5 specific types of advanced backdoors attacks, each challenging an assumption or limitation in the current defense design.

- **Complex Triggers**. Our detection scheme relies on the success of the optimization process. Would more complicated triggers make it more challenging for our optimization function to converge?

- **Larger Triggers**. We consider larger triggers. By increasing trigger size, an attacker can force the reverse engineering process to converge to a large trigger with larger norm.

- **Multiple Infected Labels w/ Separate Triggers**. We consider a scenario where multiple backdoors targeting distinct labels are inserted into a single model, and evaluate the maximum number of infected labels we can detect.

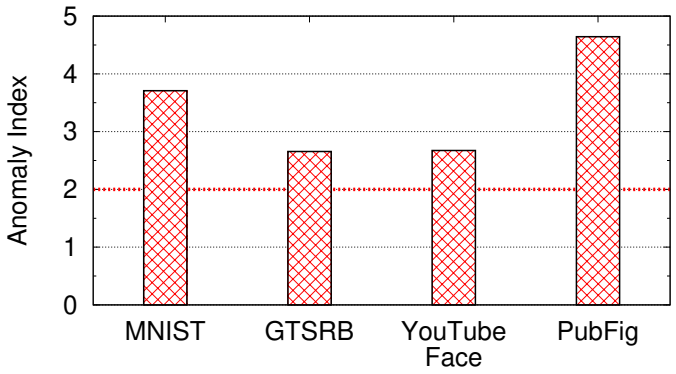


Figure 5.11: Anomaly index of infected MNIST, GTSRB, YouTube Face, and PubFig model with noisy square trigger.

- **Single Infected Label w/ Multiple Triggers**. We consider multiple triggers targeting the same label.

- **Source-label-specific (Partial) Backdoors**. Our detection scheme is designed to detect triggers that induce misclassification on arbitrary input. A "partial" backdoor that is effective on inputs from a subset of source labels would be more difficult to detect.

5.7.1 Complex Trigger Patterns

As we observed in Trojan models, triggers with more complicated patterns make it harder for the optimization to converge to the exact trigger. A more random trigger pattern might increase the difficulty of reverse engineering the trigger.

We perform simple tests by first changing the white square trigger to a noisy square, where each pixel of the trigger is assigned a random color. We inject this attack in `MNIST`, `GTSRB`, `YouTube Face`, and `PubFig`, and evaluate detection performance. The resulting anomaly index in each model is shown in Figure 5.11. Our technique detects the complex trigger patterns in all cases. We also test our mitigation techniques on

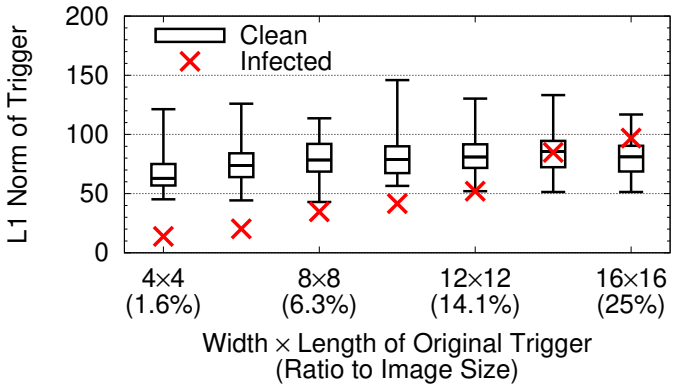


Figure 5.12: $L1$ norm of reverse engineered triggers of labels when increasing the size of the original trigger in GTSRB.

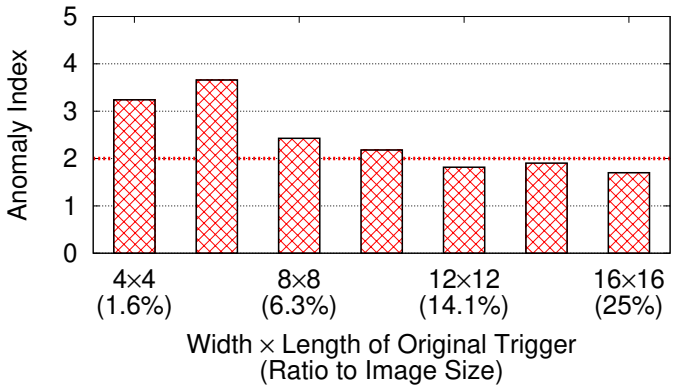


Figure 5.13: Anomaly index of each infected GTSRB model when increasing the size of the original trigger.

these models. For filtering, at an FPR of 5%, we achieve $< 0.01\%$ FNR in all models. Patching using unlearning reduces attack success rate to $< 4.2\%$, with at most 3.1% reduction in classification accuracy. Finally, we tested backdoors with varying trigger shapes (*e.g.,* triangle, checkerboard shapes) in `GTSRB`, and all detection and mitigation techniques worked as expected.

5.7.2 Larger Triggers

Larger triggers are likely to produce larger reverse-engineered triggers. This could help the infected label more closely resemble clean labels in the $L1$ norm, making outlier detection less effective. We run sample tests on `GTSRB`, and increase the size of trigger from 4×4 (1.6% of the image) to 16×16 (25%). All triggers are still white squares. We evaluate the detection technique with same configuration used in previous experiments. Figure 5.12 shows the $L1$ norm of reversed triggers for infected and clean labels. As the original trigger becomes larger, the reversed trigger also gets larger as expected. When the trigger grows beyond 14×14, the $L1$ norm does indeed blend in with that of clean labels, reducing the anomaly index below detection threshold. The anomaly index metric is shown in Figure 5.13.

The maximum detectable trigger size is largely a function of one factor: trigger size of clean labels (amount of change necessary to cause misclassification of all inputs between clean labels). The trigger size of clean labels is itself a proxy for measuring the distinctiveness of inputs across different labels, *i.e.* more labels means larger trigger size for clean labels and a greater ability to detect larger triggers. On applications like `YouTube Face`, we were able to detect triggers as large as 39% of the whole image. On `MNIST` which has fewer labels, we were only able to detect triggers of size up to 18% of the image.

5.7.3 Multiple Infected Labels w/ Separate Triggers

We consider a scenario where attackers insert multiple, independent backdoors into a single model, each targeting a distinctive label. Inserting many backdoors might collectively reduce $\delta_{\forall \to t}$ for many L_t in $\mathbb{L}$. This has the net effect of making the impact of any single trigger less of an outlier and harder to detect. The trade-off is that models

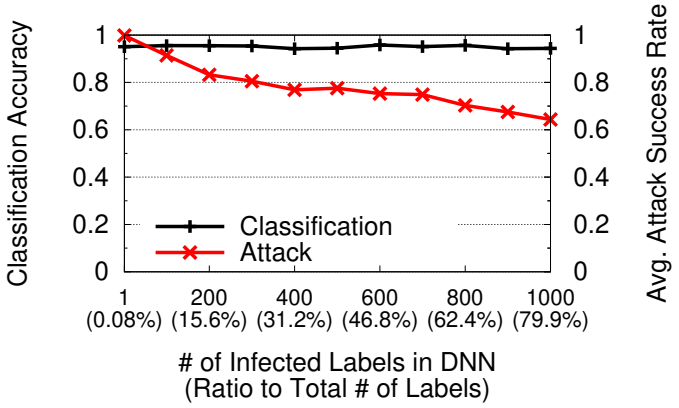


Figure 5.14: Classification accuracy and average attack success rate when different number of labels are infected in YouTube Face.

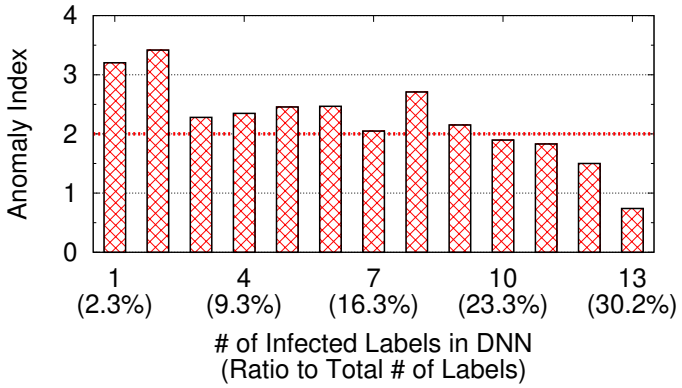


Figure 5.15: Anomaly index of each infected GTSRB model with different number of labels being infected.

are likely to have a "maximum capacity" for learning backdoors while maintaining their classification. Too many backdoors are likely to lower classification performance.

We experiment by generating distinctive triggers with mutually exclusive color patterns. We find most models, *i.e.* `MNIST`, `GTSRB`, and `PubFig`, have enough capacity to support triggers for every output label without affecting classification accuracy. But in `YouTube Face`, with $1,283$ labels, we observe an obvious drop in average attack success rate once triggers infect more than 15.6% of labels in the model. As shown in Figure 5.14, average attack success rate drops with too many triggers, confirming our intuition.

We evaluate our defenses against multiple distinct backdoors in `GTSRB`. As shown

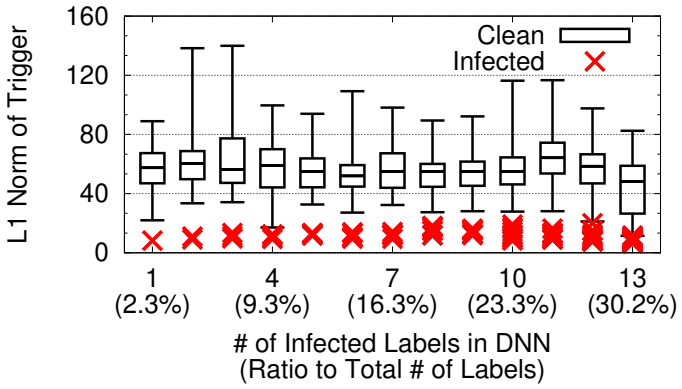


Figure 5.16: $L1$ norm of triggers from infected labels and clean labels when different number of labels are infected in GTSRB.

in Figure 5.15, once more than 9 labels (20.9%) have been infected with backdoors, it becomes very difficult for anomaly detection to identify the impact of triggers. Our results show we can detect up to 3 labels (30%) for `MNIST`, 375 labels (29.2%) for `YouTube Face`, and 24 labels (36.9%) for `PubFig`.

Though outlier detection method fails in this scenario, the underlying reverse engineering method still works. For all infected labels, we successfully reverse engineer the correct trigger. Figure 5.16 shows the norm of triggers for infected and clean labels. All infected labels have smaller norm than clean labels. Further manual analysis also validates that reversed triggers visually look similar as original triggers. Our tests show that a conservative defender can preemptively "patch" potential backdoors. When all labels are infected in `GTSRB`, patching all labels using reversed triggers would reduce average attack success rate to 2.83%. Proactive patching provides similar benefits for the other models as well. Finally, filtering is also effective at detecting adversarial inputs with low FNR at FPR of 5% across all BadNets models.

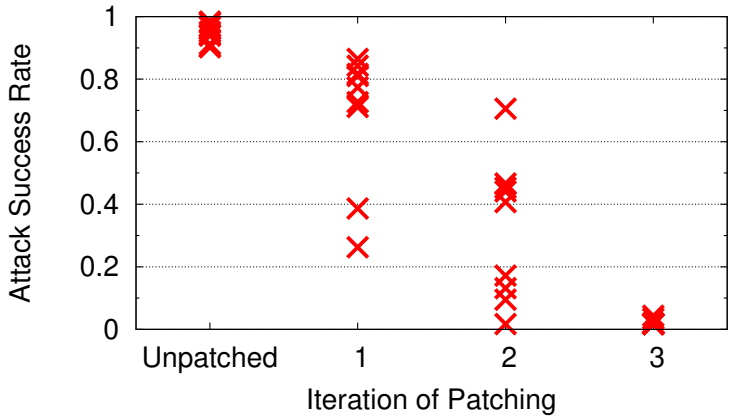


Figure 5.17: Attack success rate of 9 triggers when patching DNN for different number of iterations.

5.7.4 Single Infected Label w/ Multiple Triggers

We consider a scenario where multiple distinctive triggers induce misclassification to the same label. In this case, our detection techniques would likely only detect and patch one of the existing triggers. To test this, we inject 9 white 4×4 square triggers for the same target label into `GTSRB`. Those triggers have the same shape and color, but are located in different positions of the image, *i.e.* four corners, four edges, and the center. The attack achieves $> 90\%$ attack success rate for all triggers.

Detection and patching results are included in Figure 5.17. As suspected, a single run of our detection technique only identifies and patches one of the injected triggers. Fortunately, running just 3 iterations of our detection and patch algorithm is able to successively reduce the success rate of all triggers to $< 5\%$. We also test on other `MNIST`, `YouTube Face`, and `PubFig`, and attack success rate of all triggers are reduced to $< 1\%$, $< 5\%$, and $< 4\%$.

5.7.5 Source-label-specific (Partial) Backdoors

In Section 5.2, we define backdoor as a hidden pattern that could misclassify arbitrary inputs from any label into the target label. Our detection scheme is designed to find these "complete" backdoors. A less powerful, "partial" backdoor, could be designed such that triggers only trigger misclassification when applied to input belonging to a subset of source labels, and do nothing when applied to other inputs. Such backdoors would be a challenge to detect using our existing methods.

Detecting partial backdoors requires slightly modifying our detection scheme. Instead of reverse engineering a trigger to every target label, we analyze all possible source-target label pairs. For each label pair, we use samples belonging to the source label to solve the optimization. The resulting reversed trigger would only be effective for the specific label pair. Then, by comparing $L1$ norm of triggers for different source-target pairs, we can use the same outlier detection method to identify label pairs that are particularly vulnerable and appear as anomaly. We experiment by injecting a backdoor targeting one source-target label pair into `MNIST`. While the injected backdoor works very well, our updated techniques for detection, and mitigation are all successful.

Analyzing all source-target label pairs increases the computation cost of detection by a factor of N, where N is the number of labels. However, we can use a divide-and-conquer algorithm to reduce the computational cost to a factor of $\log N$. We leave detailed evaluation to future work.

5.8 Failed Attempts and Lessons

Here we discuss several of our failed attempts in identifying hidden backdoor and lessons we learned from these failure. We hope this could provide insights into future research and better understanding of Deep Learning.

5.8.1 Analyzing Abnormal Neurons for Backdoor Detection

Our very first attempt in detection backdoor is to analyze internal neurons of a DNN, and look for abnormal neurons. Our intuition is that, in order to achieve misclassification, the trigger must fire a set of backdoor-related neurons that could significantly affect the classification result. We suspect such neurons would have extremely high impact on the target label's output confidence. In the meantime, these neurons would stay dormant when processing normal inputs, since they would not be related to any clean patterns in clean inputs.

Following this intuition, we designed a methodology very similar to prior work by Liu [191]. For a given DNN to be analyzed, we focus on one of the deep layers, and rank output neurons of that layer based on their average activation for clean samples. After ranking, we start to remove the most dormant neurons with smallest average activation, by setting output values of these neurons to zero. If the intuition is correct, backdoor-related neurons would be removed, while benign neurons are left to maintain the model performance. Another similar approach we designed was to transform output neurons using PCA. Such improvement would help better isolate benign neurons and their combinations, and ideally make malicious component removal much easier.

The first flaw of this design is, as we briefly mentioned before, the lack of knowledge about the backdoor attack. In reality, the user does not know whether there is a backdoor injected or how the attack is performed. Without such information, users cannot quantify the robustness of the model, which makes entire defense hard to configure and the outcome hard to verify.

Besides this obvious design flaw, we also find this version of neuron pruning ineffective in several scenarios. In GTSRB, we find by pruning up to 78.1% neurons in the second to last layer, the classification accuracy of the model already drops to 60.4%, while

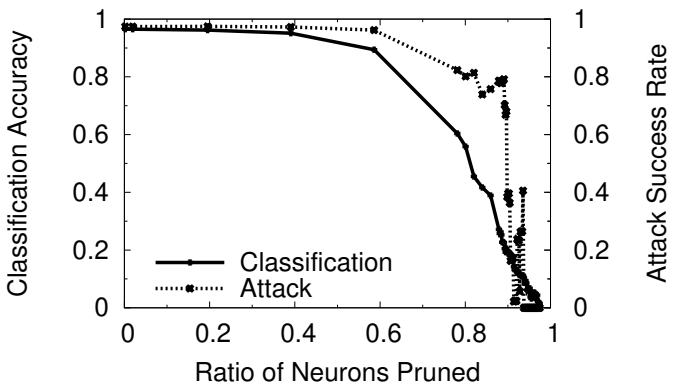


Figure 5.18: Classification accuracy and attack success rate when pruning different ratios of neurons in GTSRB.

the attack success rate still remains higher than 82.3%, as shown in Figure 5.18. This suggests that malicious and benign neurons may not be cleanly separable. Especially in deeper layers, where neurons represent higher-level patterns of the task, each neuron could stand for extremely complex patterns that represent both the trigger and benign patterns. Without understanding what exactly each neuron represent, it would be difficult to define it as benign or malicious. Therefore, the assumption that malicious neurons could be separated from benign ones and easily removed does not hold in all scenarios.

An extreme yet simple counter example is shown in Figure 5.19. If we attach an additional layer to a backdoored model, the altered model has the exactly same functionality of the original model, *i.e.* high accuracy for clean samples and high attack success rate for backdoor samples with trigger. However, when we analyze neurons of the second to last layer in the new model (output layer of the original model), the "malicious" neuron is the output neuron for the target label. This neuron is essential to both classification of clean samples and backdoored samples. It's obvious that this is a neuron that contains both benign and malicious functionality. Though this is an extreme example that could be quite different from realistic DNNs, it proves that benign and malicious neurons could have significant overlap, therefore disproves our previous assumption.

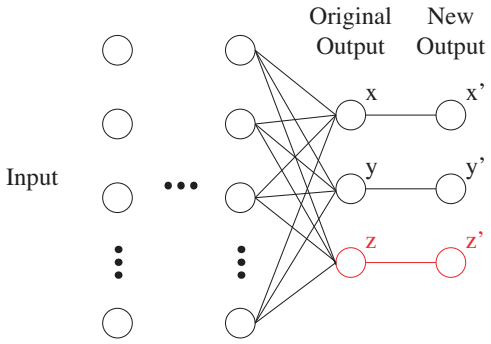


Figure 5.19: Illustration of a counter example of neuron pruning approach. In the original model, label z is the infected label. A new layer is attached to the output of the backdoored model (x, y, z) to form a new output layer (x', y', z'). The newly added layer simply passes output neuron values to the new output without modification. In the new model, the output neuron of the second to last layer (z) have both benign and malicious functionality. This proves the benign and malicious neurons could be heavily mixed.

This failed attempt is a typical pitfall of empirical system design without theoretical proof. Designs based on intuitions that seem plausible but not proven could and always would fail in unexpected ways. Especially in Machine Learning, where very little is known about what is truly happening inside ML models, we tend to build on top of our limited understanding of how ML models operate. Such mindset often leads to intuitions, like the one discussed before, that seem reasonable and natural at first glance, but later proven to be incorrect. This urges the need for stricter and more thorough inspection of any intuition we use in system design and measurement.

5.8.2 Backdoor Detection via Weight Fine-Tuning

Apart from analyzing internal neurons, we also tried analyzing internal weights and try to find potential impact of backdoor on these weights.

More specifically, we turn to the more fundamental component of the backdoor attack, the injection process. Regardless of the exactly technique to inject infected samples into training, backdoor injection relies on mixing the adversarial samples with normal

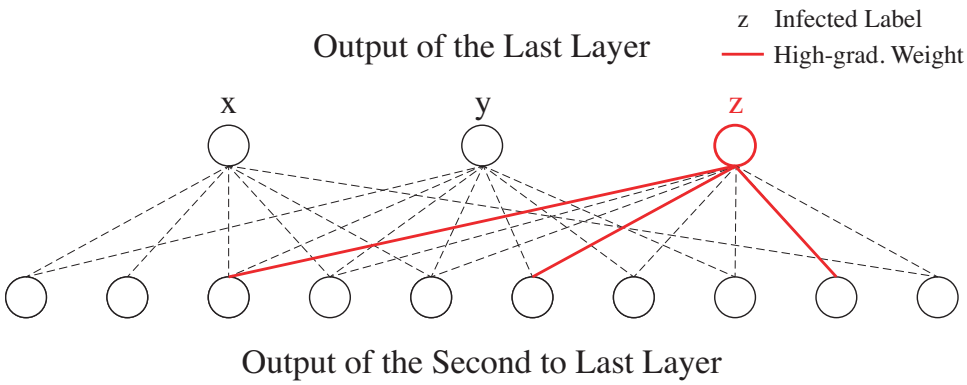


Figure 5.20: Illustration of how distribution of high-gradient weights is calculated. Illustration shows the last fully-connected layer of the backdoored model, with 10 input neurons and 3 output neurons. Label z is the infected label. 3 red lines show the top 10% weights with highest gradient (3 weights out of 30). In this case, all top 10% weights are all connected to the infected label. Therefore, the distribution concentrates on the infected label z.

samples, so the resulting DNN would learn both normal patterns and the trigger. When well trained, each weight in DNN would have a balance between two distributions of data, by having an average gradient of zero when passing all training samples. Therefore, when calculating gradients for all weights again, but using only clean samples, many weights will have non-zero gradients. This is because the backdoored samples are missing, resulting all weights to fit towards the benign data distribution. Similar intuition is also used for reverse engineering hyper-parameter configuration for model training [205].

Using this approach, by analyzing weights that have high gradient, we hope to find most related weights to the injected backdoor and understand how these weights cause misclassification. Our technique is to focus on weights in the last layer, and analyze if weights with high gradient have skewed distribution connecting to a particular label. As illustrated in Figure 5.20, top 10% weights (3 out of 10×3) with highest absolute gradient (marked in red) are connected to the same label z. Therefore, it's very likely the backdoor would affect this label's output confidence to achieve misclassification. This would allow us to identify whether a backdoor is injected and what the target label is.

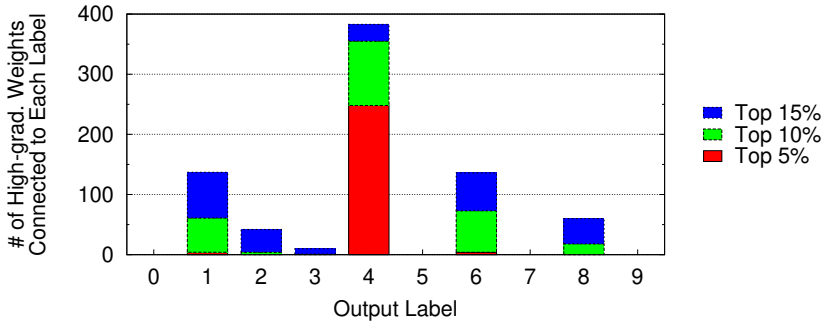


Figure 5.21: Distribution of high-gradient weights over output labels in MNIST. Label 4 is the infected label.

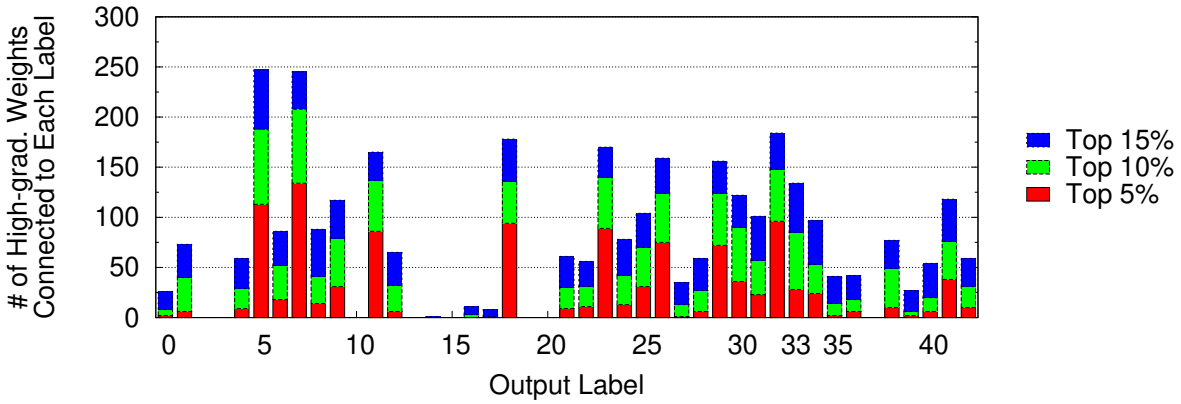


Figure 5.22: Distribution of high-gradient weights over output labels in GTSRB. Label 33 is the infected label.

This design worked well on one of the MNIST we tested. Figure 5.21 shows the distribution of highest-gradient weights over output labels in an infected MNIST model. Different colors show weight distribution of different top percentiles. It's very clear that label 4 (the infected target label) is related to the majority of top 5% weights with highest gradients (marked in red).

Quite differently, in GTSRB, we did not observe such strong skewness towards the infected target label. Figure 5.22 shows the same distribution in an infected GTSRB model, with label 33 as the infected label. Top 5% weights with highest gradient scatter across label 7, 5, *etc.* , and do not concentrate on the infected label.

Further analysis reveals two explanations for the failure of this design. First, it's

possible that backdoor-related weights do not connect to the infected label, but only to other benign labels. When malicious weights are only connected to benign labels, neurons fired by the stamped trigger would decrease the confidence of the correct label and other benign labels. The net effect of such confidence reduction is that the infected label would have the highest confidence, therefore causing misclassification. We compared the logit output (neuron values before softmax) of images with and without the trigger, and found the backdoor in GTSRB model does operate in the way we suspected. Such mode of backdoor would break the our assumption and render the detection method ineffective.

Second, a hidden assumption when using gradient to find malicious weights is that we assume the model achieves a perfect minima during backdoor training. In fact, this never happened for almost any realistic model. Most models would not achieve such ideal balance for every weight, due to factors such as early termination of training, imperfect local optima, *etc.* . Therefore, we should expect natural skewness of high-gradient weights due to model being under-trained. It is still unclear whether the skewness caused by under-training would overwhelm the effect of backdoor poisoned training. It would also reduce the effectiveness of the detection methodology.

Two factors combined, it is not surprising that this design failed to work in GTSRB, but showed very good performance in MNIST. Apart from echoing our previous point of throughly validating intuitions, this example also showed the extreme complexity of DNN in various scenarios. There exist too many factors that could affect the performance and security of a DNN, that ultimately influences of proposed system. It's the best practice to include all these factors into consideration when designing new tools and systems around ML models, and clearly evaluate their potential impact on the final performance.

5.9 Related Work

Traditional machine learning assumes the environment is benign. This assumption could be violated by an adversary at either training or testing time.

Additional Backdoor Attacks and Defenses. In addition to attacks mentioned in Section 5.2, Chen *et al.* propose a backdoor attack under a more restricted attack model, where attacker can only pollute a limited portion of training set [43]. Another line of work directly tampers with hardware the DNN is running on [206, 207]. Such backdoor circuits would also alter model's performance when a trigger is presented.

Poisoning Attacks. Poisoning attack pollutes the training data to alter the model's behavior. Different from backdoor attack, poisoning attack does not rely on the trigger, and alters model's behavior on a set of clean samples. Defenses against poisoning attack mostly focus on sanitizing the training set and removing poisoned samples [41, 208, 209, 210, 211, 212]. The insight is to find samples that would alter model's performance significantly [41]. This insight has shown to be less effective against backdoor attack [43], as injected samples do not affect model's performance on clean samples. Also, it's impractical in our attack model, as the defender does not have access to the poisoned training set.

Other Adversarial Attacks against DNNs. Numerous (non-backdoor) adversarial attacks have been proposed against general DNNs, often crafting imperceptible modifications to images to cause misclassification. These can be applied to DNNs during inference [213, 214, 215, 216, 217]. A number of defenses have been proposed [218, 219, 220, 221, 222], yet many have shown to be less effective against an adaptive adversary [223, 224, 225, 226]. Some recent work tries to craft *universal perturbations*, which

would trigger misclassification for multiple images in an uninfected DNN [227].

5.10 Conclusion and Future Work

Our work describes and empirically validates our robust and general detection and mitigation tools against backdoor (Trojan) attacks on deep neural networks. Beyond the efficacy of our defense against basic and complex backdoors, one of the unexpected take-aways of our paper is the significant differences between two backdoor injection methods: the trigger-driven BadNets end-to-end attack with full access to model training, and the neuron-driven Trojan Attack without access to model training. Through our experiments, we find that the Trojan Attack injection method generally adds more perturbations than necessary, and introduces unpredictable changes to non-targeted neurons. This makes their triggers harder to reverse engineer, and makes them more resistant to filtering and neuron pruning. However, the tradeoff is that their focus on specific neurons make them extremely sensitive to mitigation via unlearning. In contrast, BadNets introduce more predictable changes to neurons, and can be more easily reverse engineered, filtered and mitigated via neuron pruning.

Finally, while our results are robust against a range of attacks in different applications, there are still limitations. First and foremost is the question of generalization beyond the current vision domain. Our high-level intuition and design of detection/mitigation methods could be generalizable: the intuition for detection is that the infected label is more vulnerable than uninfected labels, and this should be domain agnostic. The main challenge of adapting the entire pipeline to non-vision domain is to formulate the backdoor attack process and design a metric measuring how vulnerable a specific label is (like Equation 5.2 and Equation 5.3).

Chapter 6

Conclusions and Discussions

In this dissertation, we take a detailed look into important implications of the non-transparency nature of Machine Learning systems. Three separate work are covered to look at the performance, robustness, and security of such opaque systems. We use empirical approaches to understand and quantify these properties of ML systems, and also propose tools and solutions for improving and securing ML systems.

In this chapter, we summarize the contribution of our work, along with discussions about important topics in the study of ML and general research.

6.1 Summary

Machine Learning system appears to humans as a complex numerical black-box, which makes understanding its internals extremely difficult. Such non-transparency imposes difficulty to 1) understand how model design impacts end-to-end performance, 2) understand the model robustness against adversarial inputs, and 3) audit if model contains hidden malicious behavior. These three implications would significantly reduce the utility and security of Machine Learning system, and hinder its wide adoption.

In Chapter 3, we take a data-driven approach to quantify how model's design choices affect its end-to-end performance. We use 6 MLaaS platforms with different levels of control and complexity as representative data points, and measure their performance on 119 real-world datasets. Our work produces three key takeaways. First, ML models with more freedom of control produce more potential performance gain, as well as greater possibility of performance degradation from poor configuration. Second, several MLaaS platforms use internal tests to automatically adjust model configuration. Though it significantly reduces user's effort of configuration, their aggregated performance lags behind well-tuned versions of other more configurable counterparts. Finally, much of the gains from configuration come from choosing the right classifier. Experimenting with a small random subset of classifiers is likely to produce near-optimal performance.

In Chapter 4, we analyze Deep Learning model's robustness against adversarial attacks, especially in the context of Transfer Learning. We propose a new adversarial attack on black-box student models in the wild by leveraging the white-box teacher models. We experimentally validate the effectiveness of this attack under different transfer learning scenarios, and show it can be successfully carried out against real transfer learning services. We also design and evaluate several defenses to mitigate this new attack.

In Chapter 5, we study backdoor attack against Deep Learning models. We design a series of defenses to detect and mitigate backdoor attack in a given DL model. Our proposed techniques could, first, detect hidden backdoors, second, identify target labels and triggers used by the attack, third, detect and filter adversarial inputs with the backdoor trigger, and last, patch DL models to be robust against injected backdoors. Extensive experiments are conducted to fully understand how our defenses perform facing prior backdoor attack and also different variants of more advanced attacks.

6.2 Discussions

Here I would like to discuss several topics related to our work, along with justification for several high-level design decisions we make in this thesis.

6.2.1 Transparency and Interpretability of Machine Learning

Whether transparency and interpretability of Machine Learning could be ever achieved or even necessary has been an ongoing debate ever since Machine Learning started to be deployed in real-world. In fact, there exist multiple definitions of transparency and interpretability that sometimes even confuse these two concepts. Here we introduce our definition of transparency and interpretability, and present our opinions on whether and how they could be achieved.

Transparency of an ML model means that we have the ability and confidence to "predict" the output of any given input, tested or even untested. It is not the same as running a new input through the ML model and retrieve the output, but in contrary, we would know the output even without running the model. In essence, this means we have the knowledge of how the model would behave.

Interpretability, however, is different from transparency. Interpretability suggests that we can map and translate between the operating mechanism of ML models and how humans understand and solve particular tasks. It requires a much in-depth understanding of ML models, where we need to understand why ML models make decisions, and also explain the reason in a form that is understandable by humans.

To put these definitions more concretely, here is an example of a face recognition task. Transparency means that for a given face, we should have the confidence to know that which label it will be classified into. For a high-performance model, we should have the ability to know it will be classified into the correct label. For a ill-trained model or a

model infected with a backdoor, we should be able to know which samples will be classified incorrectly and when such incorrect behavior would happen. But interpretability requires that we can explain why the ML model makes decisions. Is it because of the shape of the nose, the skin color, or other subtle details of the face? Or is it because of an injected malicious trigger embedded into the model? This means we need to translate numerical operations inside ML models into explicit patterns that humans use for face recognition.

Achieving Interpretability. With such definition, interpretability remains extremely difficult. In our opinion, complete interpretability might be fundamentally impossible. Without considering the artifact and insufficiency of training, ML models have distinct architecture from human brain, which makes interpretability nearly impossible. From the start, ML model is designed as a simplified version of human brain for specific functionalities. Convolution neural network tries to mimic the hierarchical structure of how humans recognize patterns. Recurrent neural network tries to mimic the concept of memory. But all these ML models later evolve and deviate from human brain to better capture and model patterns inside data. From the architecture point of view, these ML models are over-simplified and task-specific, while human brain is more complex and for general purposes. Mapping between an extremely specialized model and a general-purpose brain, itself, remains an impossible task.

Many prior work aim at improving the interpretability of ML models. Most of work along this direction try to provide local explanation of model's decision for a specific input sample. Proposed systems, such as LIME [228] and LEMNA [229], extract most important elements of a given input that result in the final output, *e.g.,* a set of pixels in an image, a sequence of bytes in a binary. These work try to approximate the decision process of ML models by using a much simpler model that is easier to understand. Though it cannot fully replicate the internal process of the target ML model, it does

provide insight into important factors in the decision process, and helps us understand the model. Still, a significant gap between these work and the ultimate goal of complete interpretability exists. Bridging this cap would require sophisticated understanding of both human brain and the ML architecture, which poses as an impossible challenge to the community.

Achieving Transparency. In contrary, transparency only requires end-to-end guarantee of model's behavior, even though we do not understand how ML model makes such decision. We believe this could be achieved via a combination of theoretical proof and empirical testing. To push it to the extreme, a simple yet costly brute-force solution is to test all possible inputs of an ML model and identify if their outputs are correct. Though it does not improves our understanding of how the ML model makes decision, *i.e.* interpretability, it does offer the strongest transparency possible. The cost of this empirical testing could be further reduced by introducing theoretically proven properties, so redundant testing could be pruned.

Furthermore, for certain domains and scenarios where only partial transparency is required, it's easier for empirical analysis to achieve transparency with limited cost. For example, in our backdoor project, the specific requirement is to make sure no small trigger exists that could effectively cause misclassification. This is clearly only a small subset of the transparency requirement we defined before, but this specific requirement of transparency allows us to design empirical testing methods and provide results under limited cost.

This is also the reason we focus on achieving transparency by empirical measurements and design empirical methods. For each specific implication we studied in this thesis, performance and security, we tackle them individually and design tools and measurements to meet the specific needs in the corresponding scenario. Instead of pursuing solutions for

the ultimate transparency and interpretability of ML, our choice offers practical solutions to ensure the imminent large scale adoption of ML is safe and secure.

6.2.2 Empirical Analysis vs. Theoretical Analysis

Another choice we make in our work is to use empirical approaches to tackle problems instead of using theoretical methods. This has been a long-going debate, not only in the ML community, with the optimal answer being obviously that we need both. However, given the level of complexity of ML models, especially DL models, theoretical analysis proves to be extremely difficult for any realistic systems. This

Adversarial attack and defense of Deep Learning models serves as a good example of how difficult theoretical analysis of ML could be. Even with the significant amount of effort and attention the ML community has put into this area, there has been limited progress along the theoretical direction. The most recent effort borrows tools from geometry to theoretically prove and quantify that a given model is robust against adversarial samples that are created with a given constraint (in this case, with limited perturbation) [230, 231, 232]. However, given the huge complexity of ML models, such tools are limited in its power. In this case, it can only prove the model's robustness for a given sample, instead of all possible samples. Also given the high computation complexity of such theoretical tools, adopting it on realistic models would incur high runtime cost that are not feasible.

Unfortunately, the currently inefficiency and ineffectiveness of theoretical tools urges for effective empirical analysis immediately. Though it does not provide the desired strong guarantee like theoretical analysis, it does offer effectiveness under realistic assumptions. With the rapid advancement and adoption of ML, it's critical to provide empirical tools today to secure these systems, given the vacuum of effective theoretical tools.

Besides urgency for effective empirical solutions, empirical analysis also provides valuable insights and feedback that could help improve theoretical study. Recent success and advance of Machine Learning is mostly empirical, with numerous successful instances from industry proving huge potentials of Machine Learning. This inevitably results in lagging theory explaining such success, and more so for other characteristics such as robustness and security. Facing such circumstances, empirical analysis provides valuable insights that could filter out promising directions.

A good example in this dissertation is the detection of backdoor. Even though we did not provide theoretical analysis on what characteristics backdoor would have, several of our failed attempts of previously proposed ideas do offer insights in what characteristics backdoor does not have. We find that, unlike prior work suggested, backdoor does not cause collateral damage on the model to have higher prediction error on the infected labels. This invalidates the prior assumption on backdoor, and leave us with remaining directions that would be more promising.

Another example is the adversarial attack against transfer learning. Besides the security implication, it also exposes a concerning fact about how DL models operate internally. The success of internal representation mimicking shows that DNN will map two inputs from distinct classes into similar internal representations. This suggests that, the transformation or feature extraction functionality of DNN early layers is imperfect. An ideal feature extraction, if ever existed, should be able to transform inputs of different classes to spaces far away from each other. Instead, the success of our attack indicates that not only this is not achieved, spaces of different classes are heavily overlapped, and small modification could move two inputs overlap in the latent feature space. Besides the obvious security vulnerability we discovered, it could have implication on model performance and its ability to generalize. This finding suggests that more analysis is required in understanding the effectiveness and accuracy of DNN's transformation and

feature extraction.

6.2.3 Variety of Benchmarking Datasets

One of the most important factor in empirical analysis is the variety of scenarios the proposed system has been tested in. This provides confidence that findings from empirical analysis could generalize to unseen and untested cases. In the field of Machine Learning, this often means the variety of datasets and tasks any proposed methodology is tested. In the scenario of backdoor detection, this means we need to test on various tasks, datasets, with different complexities, sizes, and purposes. Similarly, in other areas such as adversarial attack and defense, same principle also applies.

We find this principle to be especially necessary in the study of Machine Learning. Prior to our current design for backdoor detection, a failed attempt was to analyze internal weights of the DNN and measure how much impact it would have to further fine-tune the DNN with clean data. Early result showed that this simple technique worked quite well on MNIST and PubFig, but it was later overthrown that it failed on other datasets we tested such as GTSRB and Trojan models. Similar to us, another example is defense against adversarial attack. Many previously proposed defenses are only tested on simple cases, *e.g.,* MNIST, CIFAR10, and are later shown to be not able to generalize to more realistic cases [233, 148].

The lack of theoretical understanding of ML brings more importance of the variety of test cases. Without being backed by strong theoretical guarantee, none of the empirical analysis is guaranteed to generalize to untested scenarios. This leaves the only solution to throughly test the proposed system on scenarios as various as possible. Though it does not fulfill the missing vacuum of theoretical analysis, it would encourage more mature and thorough study.

6.2.4 Generalization beyond Vision and Classification

Most of our work in this thesis have been focusing on the vision domain and also classification tasks. Though we did not perform experiments on other domains like text, audio, or other potential types of input, intuitions of our techniques could be generalized. Here we discuss the main challenges of adapting not only our work, but other similar work from vision to more domains.

One of the major reasons that most Deep Learning study have been focusing on the vision domain is its fast growth and wide adoption. With significantly more resources in the vision domain, research areas like security would benefit from a wide range of deployed systems. Another major reason is the ease of understanding and quantifying changes of the input. It's much easier to visualize and quantify modification on images or videos than on audio, wireless signals, *etc.* . We can find many example of such in adversarial attacks and defenses of Deep Learning.

Potentially, this could make many study focus too much on a particular area, and not be able to generalize to other domains. For example, in defense against adversarial attack, many proposed defenses rely on assumptions such as, image inputs are generally smooth. Such assumptions might not generalize to domains that do not have continuous input values, such as text. Even though such practice could generate high performance solutions for specific domain and task, it does not suggest that it addresses the fundamental problem across domains.

All three projects we include in this dissertation also have specific focus on a subdomain of ML. Our first study on ML system design and its impact on performance focus on binary classification problem using traditional ML algorithms. Our adversarial attack against transfer learning and detection system for backdoor attacks both focus on the vision domain and classification task. The most obvious consequence is that it requires

certain modification of the system to apply to other domains. For example, we need to define a new distance function to quantify the amount of perturbation, if we were to apply the adversarial attack to non-vision domains. Also we need to define a new form of trigger injection process adjust the backdoor detection method to non-vision models.

We already discuss the generalization problem separately in prior chapters, and we believe the core intuition behind our proposed systems would generalize to other domains and tasks. However, from a more rigorous standpoint, we cannot speak for sure that these intuitions will generalize, without real experiments. Verifying whether or how existing work generalize to untested scenarios and tasks would be a necessary and important step to thoroughly understand the evaluate ML study.

6.3 Lessons of General Research from a Retrospective View

My research interests and focuses are much wider than most of PhDs. I have covered topics of data mining and measurement of online social networks, security of mobile systems, and network measurement in the first three years of my PhD. During this period, I have also touched several other orthogonal topics, such as cryptocurrency. The transition to Machine Learning only happens in my late fourth year. But it happens to produce some of my most productive and fruitful projects throughout my PhD.

With such experience of working on a truly wide range of topics, here are some of the lessons I learned when looking at my PhD from a retrospective view. I hope these lessons about general research and PhD would be valuable to share.

The Importance of Data-driven Research. Despite the seemingly wide range of topics I have worked on, all of them center around the idea of data-driven empirical

research. The philosophy is to use real data to discover insights, validate assumptions, evaluate designs, and potentially identify more research problems. Instead of approaching research from a more abstract way, data-driven research does produce results that you can trust and have confidence about. In reality, data-driven research often produce results that disapprove prior assumptions, reveal insights for future research directions, *etc.* .

The common theme behind all my research projects turn the wide range of topics into one of my most valuable lessons of research. Multiple experiences further validates the power of data-driven research. For example, in the project trying to understand whether crowdsourcing systems work in areas requiring higher expertise [234], we use empirical data to validate effectiveness of crowdsourcing. And we also discover that such crowdsourcing systems only work when high quality workers could be selected, and this could be achieved using simple indicators. In another work trying to understand the design tradeoff of personalized livestreaming systems [235], we use data from a popular livestreaming service to dissect the architecture of such service, and quantify the performance of each component in the system. The result helps us understand the tradeoffs when designing systems with such unique requirement of extremely low latency between broadcaster and all viewers.

These projects outside the area of Machine Learning further strengthens my belief in data-driven research. And it is also a key reason we put extremely large amount of focus on evaluating our designs and systems using realistic datasets and scenarios. In all projects in this thesis, we try to expand to larger and more complex models and datasets, which more closely resembles practical scenarios in the real world. It is only after fully evaluated on a wide range of scenarios, we can trust the effectiveness of the design with more confidence. Unsurprisingly, this lesson is learned from multiple failed attempts that work as perfect examples for this argument. If you are interested, please refer to Section 5.8.

The Pitfall of Empirical Research: Lost of Focus. One of the major problems of empirical research is the tendency to get lost in the swamp of interesting results and findings, and lose the focus. Empirical research is often driven by the detailed results, instead of a major research problem. For example, many of the empirical measurement work in my PhD [235, 236, 237] did not have a clear objective or a main research question in mind when the project first started. It was only after a significant amount of experiments and findings, we can identify interesting questions and organize detailed results into a cohesive story.

Seemingly, empirical research is driven by detailed results, and interesting research questions only surface when related findings have been discovered. But a less acknowledged fact is that good empirical research also requires a key theme or a core direction to guide the design of experiments and measurements. Comparing all empirical measurement projects I have involved in, a central direction plays an important role in determining the level of productivity and "efficiency" of a project. It helps focus experiments around a common topic, and reduces the time and effort spent on out-of-scope experiments.

This lesson has been validated multiple times during my PhD. As a negative example, my first project that studied the collaborative investing platforms did not start with the final idea, but trying to understand user behavior on such financial platforms. Though it seems like a focused direction to pursue, we did not know which exact behavior we were trying to analyze. Without the insight into which behavior pattern to look for, we ended up with running experiments that did not produce much useful results.

One of the positive example, on the other side, is the latest work detecting and mitigating backdoor attack. Despite the fact that this project does have a very clear objective in the first place, it does help construct all experiments around a common theme. Even though many iterations of design were proposed and quickly discarded, the

entire project was extremely productive and efficient.

The Key to Success: Motivation, and Collaboration. Apart from aforementioned factors, there are other important factors that lead to a successful research project. First, staying motivated is the major driving force that helps overcome obstacles in research. Everyone understands that research is often open-ended, and it is not always guaranteed to produce satisfying results. Therefore, we have the tendency to question whether there is a solution to the question. Beyond this, there also exists numerous obstacles lying between you and the final answer. We already discussed several examples covered in this thesis, which went through multiple iterations of design. These are the best examples showing that how staying motivated and optimistic could help us overcome obstacles and do not give up half way in the middle.

Another deciding factor is collaboration. Research would be quite difficult, especially when you are fighting by yourself alone. I'm really grateful that I was able to work with a group of talented and hard-working researchers in all my projects. It wouldn't be possible to make this dissertation possible without all the brainstorming, discussion, arguing, even fighting. This great team cover corners that any of us oversight and details we miss. To that, I owe my most sincere gratitude.

Appendix A

Appendix

A.1 Appendix of Empirical Analysis of Machine Learning as a Service

A.1.1 ML-as-a-Service Platforms

Automatic Business Modeler (ABM). ABM provides full automation of over the entire ML pipeline. This includes preprocessing, feature selection, model selection, and parameter tuning. ABM only requires users to upload dataset in a particular format, and automatically builds an ML model.

Amazon Machine Learning (Amazon). Amazon Machine Learning is a part of Amazon AWS that specializes in providing predictive analytics service. Amazon automates most of the controls, while only allowing users to tune classifier parameters. We have limited knowledge about the internal design of Amazon. But according to its user manual, Amazon uses Logistic Regression in the back-end [238].

BigML. BigML allows users to choose classifiers and tune parameters. It allows users to build and control ML pipeline through Web GUI and script language [1]. BigML maintains a public Script Gallery [2], where users can share scripts with customized functionalities. Users can also export trained models and run them locally.

PredictionIO. PredictionIO is an open-source ML server where users can customize their own predictive models and deploy as a Web service. Users can modify templates provided by PredictionIO to realize their own functionalities. Users can choose classifiers by downloading different templates, and tune parameters by changing variables in the template.

Microsoft Azure Machine Learning Studio (Microsoft). Microsoft provides almost full control over the entire ML pipeline. Users can choose from multiple methods to perform data transformation and preprocessing. Users can also select classifiers, tune parameters, and even apply feature selection. Microsoft also provides various options for each component. Since Microsoft doesn't provide any API, we need to manually interact with the Web interface to create ML models for different ML system configurations.

Google Prediction API (Google). Google Prediction API is one of services offered by *Google Cloud Platform*. It is designed as black-box, *i.e.* users have no control over any step in the ML pipeline. It automatically builds ML models for users after datasets are uploaded to Google Cloud. There is little information about the internal design of Google Prediction API.

[1] `https://bigml.com/whizzml`
[2] `https://bigml.com/gallery/scripts`

A.2 Appendix of Practical Attacks against Transfer Learning

A.2.1 Disclosure

While we did not perform any attacks on deployed image recognition systems, we did experiment with publicly available Teacher models from Google, Microsoft and the open source PyTorch originally started by Facebook. Following their tutorials, our results showed they were vulnerable to this class of adversarial attacks. In advance of the public release of this paper, we reached out to machine learning and security researchers at Google, Microsoft and Facebook, and shared our findings with them.

A.2.2 Definition of $DSSIM$

$DSSIM$ (Structural Dissimilarity) is a distance metric derived from $SSIM$ (**S**tructural **SIM**ilarity). Let $x = \{x_1, ..., x_N\}$, and $y = \{y_1, ..., y_N\}$ be pixel intensity signals of two images being compared, respectively. The basic form of $SSIM$ compares three aspects of the two image samples, luminance (l), contrast (c), and structure (s). The $SSIM$ score is then described in the following equation.

$$\begin{aligned} SSIM(x,y) &= l(x,y) \cdot c(x,y) \cdot s(x,y) \\ &= \Big(\frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}\Big) \\ &\quad \cdot \Big(\frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}\Big) \\ &\quad \cdot \Big(\frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}\Big) \end{aligned} \tag{A.1}$$

μ and σ are mean and standard deviation of pixel intensities of image samples. C_1, C_2, and C_3 are constants, and recommendation for choosing these constants is included in

the original paper [124, 125].

$DSSIM$ is calculated as $\frac{1-SSIM}{2}$. It ranges from 0 to 1, where 0 represents two images are identical, and 1 represents two images are negatively correlated (often achieved by inverting the image).

In our experiments, we use an improved version of $SSIM$, referred as multi-scale $SSIM$, which also considers distortion due to viewing conditions (*e.g.,* display resolution). This is achieved by iteratively comparing the reference and distorted images at different scales (or resolutions) by applying a low-pass filter to downsample images. To compute $DSSIM$, we use the implementation of multi-scale $SSIM$ from TensorFlow and follow the recommended parameter configuration [3].

Student Task	Dataset	# of Classes	Training Size	Testing Size	Teacher Model	Training Configurations
Face	PubFig83 [129]	65	5,850	650	VGG-Face [118]	epoch=200,batch=32,optimizer=adadelta,lr=1.0
Iris	CASIA Iris [130]	1,000	16,000	4,000	VGG16 [8]	epoch=100,batch=32,optimizer=adadelta,lr=0.1
Traffic Sign	GTSRB [131]	43	39,209	12,630	VGG16 [8]	epoch=50,batch=32,optimizer=adadelta,lr=1.0
Flower	VGG Flowers [133]	102	6,149	1,020	ResNet50 [6]	epoch=150,batch=50,optimizer=sgd,lr=0.01

Table A.1: Detailed information about dataset, Teacher models, and training configurations for each Student task.

[3] `https://github.com/tensorflow/models/blob/master/research/compression/image_encoder/msssim.py`

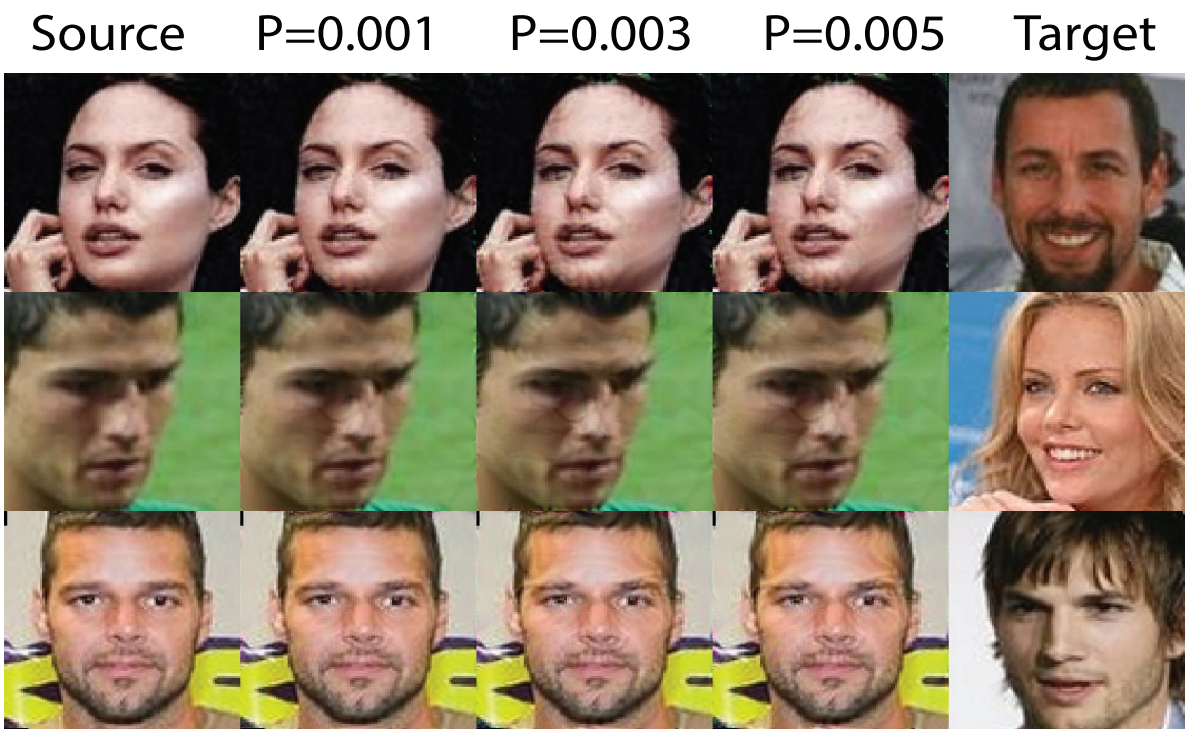
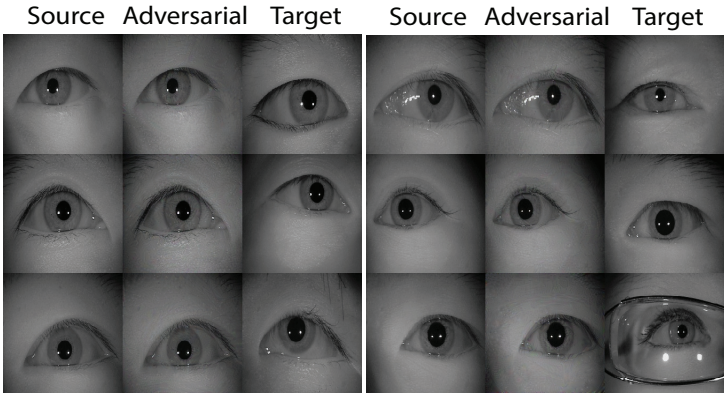


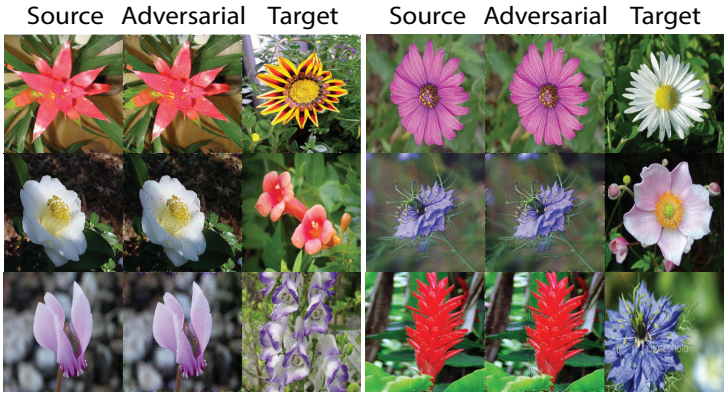
Figure A.1: Adversarial examples generated from the same source image with different perturbation budgets (using $DSSIM$). Lower budget produces less noticeable perturbations.

Figure A.2: Comparison between adversarial images generated using $DSSIM$ perturbation budget ($P = 0.003$) and L_2 budget ($P = 0.01$). Budgets of both metrics are chosen to produce similar targeted attack success rate around 90%.



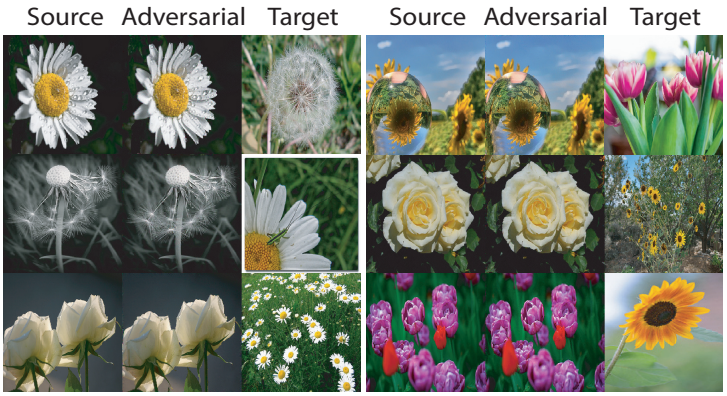
(a) Iris ($P = 0.005$)

(b) Traffic Sign ($P = 0.01$)

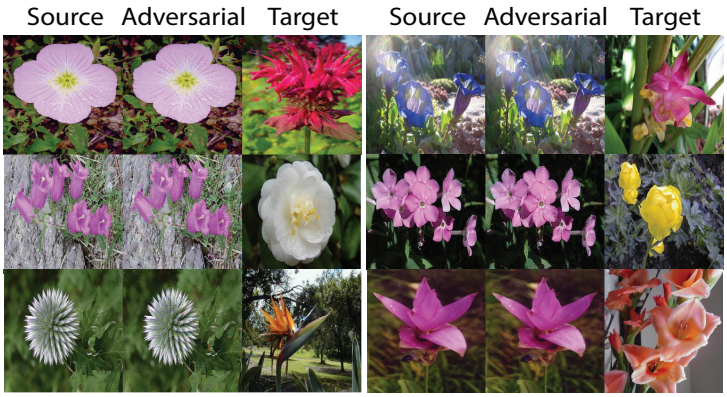


(c) Flower ($P = 0.003$)

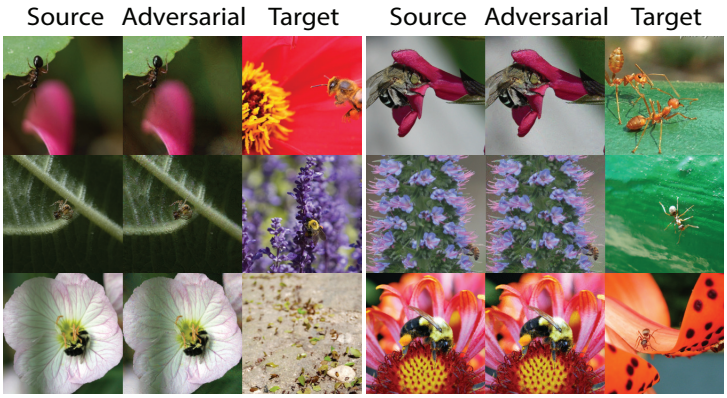
Figure A.3: Adversarial images generated in Iris, Traffic Sign, and Flower. Perturbation budgets selected result in unnoticeable perturbations. Iris attack targets at VGG16 layer 15 (out of 16 layers). Traffic Sign attack targets at VGG16 layer 10 (out of 16 layers), and Flower attack targets at ResNet50 layer 49 (out of 50 layers).



(a) Google Cloud ML ($P = 0.001$)

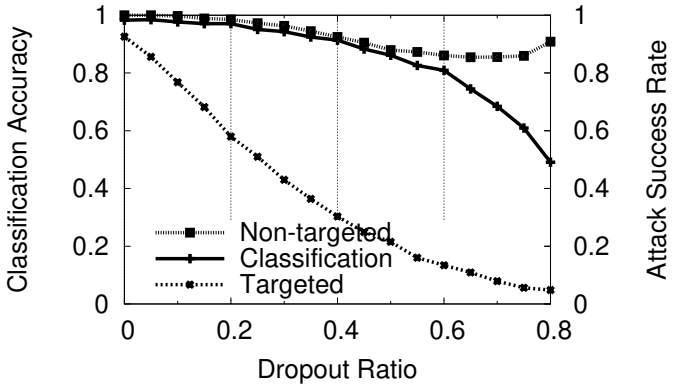


(b) Microsoft CNTK ($P = 0.003$)

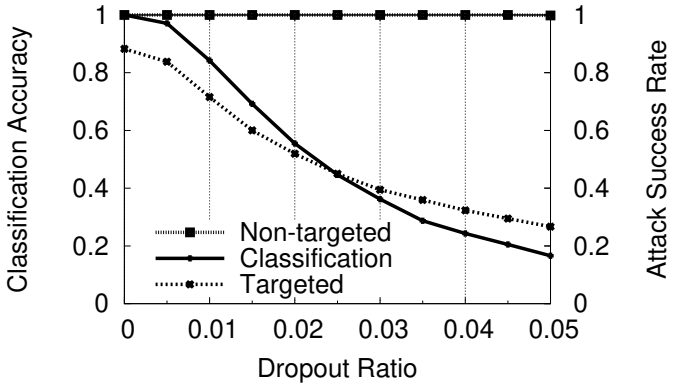


(c) PyTorch ($P = 0.001$)

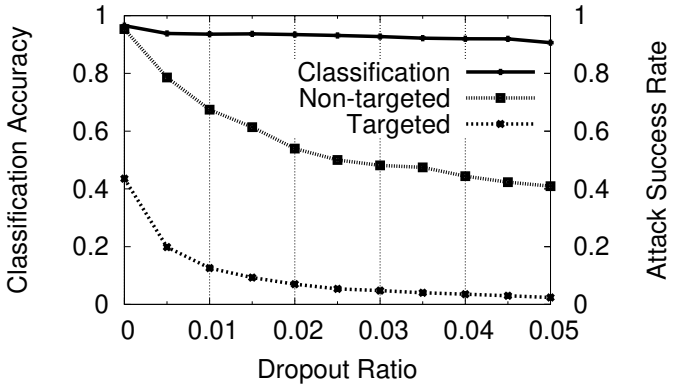
Figure A.4: Adversarial images generated for Student models trained on Google Cloud ML, Microsoft CNTK, and PyTorch. Attacks using these samples achieve targeted success rate of 96.5%, 99.4%, and 88.0% in corresponding models.



(a) Face.

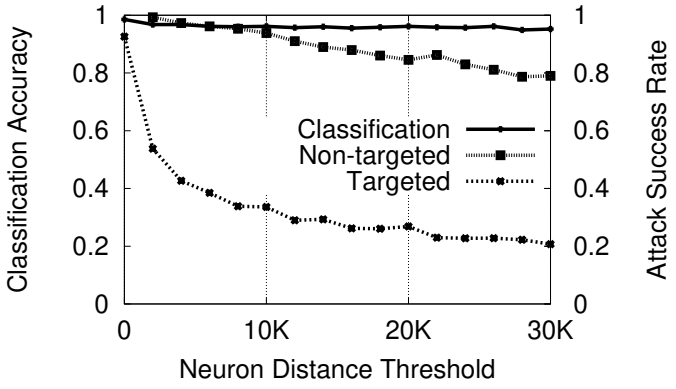


(b) Iris.

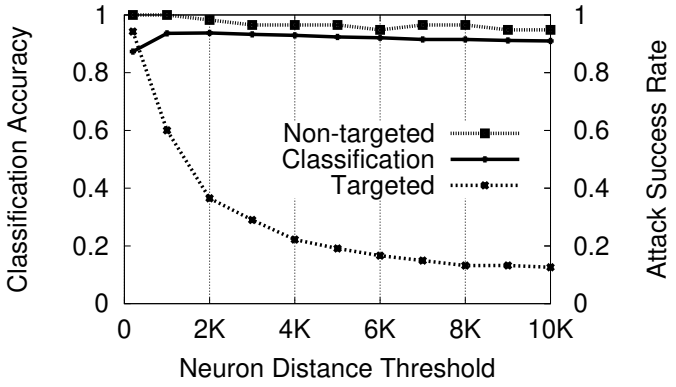


(c) Traffic Sign.

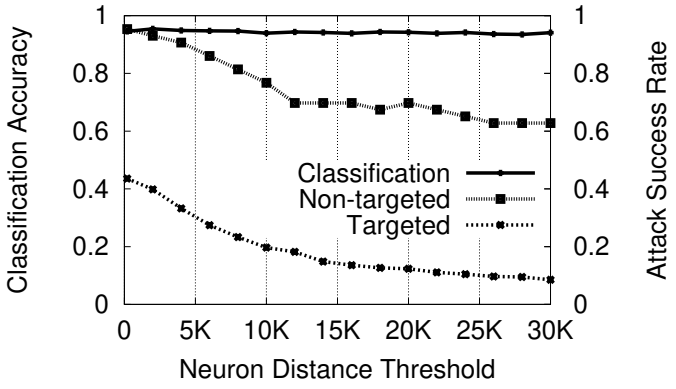
Figure A.5: Performance of applying Dropout as defense with different Dropout ratio in Face, Iris, and Traffic Sign.



(a) Face.



(b) Iris.



(c) Traffic Sign.

Figure A.6: Performance of modifying Student as defense with different distance thresholds in Face, Iris, and Traffic Sign.

A.3 Appendix of Identifying and Mitigating Backdoor Attacks in Neural Networks

(a) MNIST (b) GTSRB (c) YouTube Face

(d) PubFig (e) Trojan Square (f) Trojan WM

Figure A.7: Examples of adversarial images with white square trigger added to the bottom right corner of the image.

Task / Dataset	# of Labels	Training Set Size	Testing Set Size	Training Configuration
MNIST	10	50,000	10,000	inject ratio=0.1, epochs=5, batch=32, optimizer=Adam, lr=0.001
GTSRB	43	35,288	12,630	inject ratio=0.1, epochs=10, batch=32, optimizer=Adam, lr=0.001
YouTube Face	1,283	375,645	64,150	inject ratio=0.1, epochs=10, batch=32, optimizer=Adadelta, lr=0.1
PubFig	65	5,850	650	inject ratio=0.1, epochs=15, batch=32, optimizer=Adadelta, lr=0.1 First 12 layers are frozen during training. First 5 epochs are trained using clean data only.

Table A.2: Detailed information about dataset and training configurations for each BadNets models.

Layer Type	# of Channels	Filter Size	Stride	Activation
Conv	16	5×5	1	ReLU
MaxPool	16	2×2	2	-
Conv	32	5×5	1	ReLU
MaxPool	32	2×2	2	-
FC	512	-	-	ReLU
FC	10	-	-	Softmax

Table A.3: Mode Architecture for MNIST. FC stands for fully-connected layer.

Layer Type	# of Channels	Filter Size	Stride	Activation
Conv	32	3×3	1	ReLU
Conv	32	3×3	1	ReLU
MaxPool	32	2×2	2	-
Conv	64	3×3	1	ReLU
Conv	64	3×3	1	ReLU
MaxPool	64	2×2	2	-
Conv	128	3×3	1	ReLU
Conv	128	3×3	1	ReLU
MaxPool	128	2×2	2	-
FC	512	-	-	ReLU
FC	43	-	-	Softmax

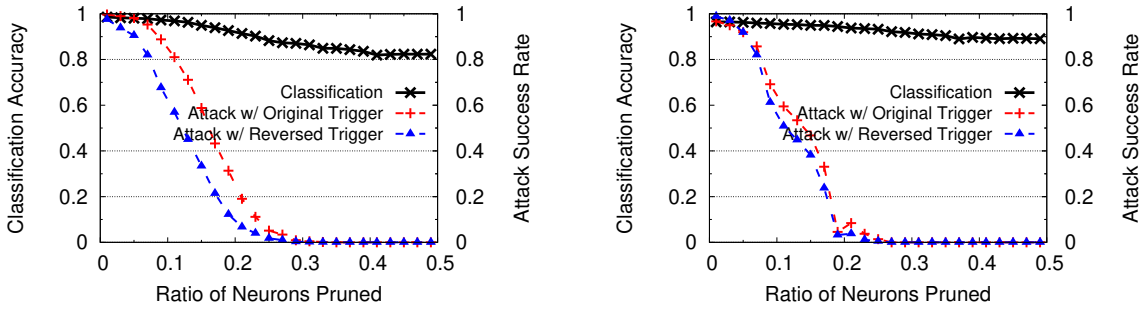
Table A.4: Model Architecture for GTSBR.

Layer Name (Type)	# of Channels	Filter Size	Stride	Activation	Connected to
conv_1 (Conv)	20	4×4	2	ReLU	
pool_1 (MaxPool)		2×2	2	-	conv_1
conv_2 (Conv)	40	3×3	2	ReLU	pool_1
pool_2 (MaxPool)		2×2	2	-	conv_2
conv_3 (Conv)	60	3×3	2	ReLU	pool_2
pool_3 (MaxPool)		2×2	2	-	conv_3
fc_1 (FC)	160	-	-	-	pool_3
conv_4 (Conv)	80	2×2	1	ReLU	pool_3
fc_2 (FC)	160	-	-	-	conv_4
add_1 (Add)	-	-	-	ReLU	fc_1, fc_2
fc_3 (FC)	1280	-	-	Softmax	add_1

Table A.5: DeepID Model Architecture for YouTube Face.

Layer Type	# of Channels	Filter Size	Stride	Activation
Conv	64	3×3	1	ReLU
Conv	64	3×3	1	ReLU
MaxPool	64	2×2	2	-
Conv	128	3×3	1	ReLU
Conv	128	3×3	1	ReLU
MaxPool	128	2×2	2	-
Conv	256	3×3	1	ReLU
Conv	256	3×3	1	ReLU
Conv	256	3×3	1	ReLU
MaxPool	256	2×2	2	-
Conv	512	3×3	1	ReLU
Conv	512	3×3	1	ReLU
Conv	512	3×3	1	ReLU
MaxPool	512	2×2	2	-
Conv	512	3×3	1	ReLU
Conv	512	3×3	1	ReLU
Conv	512	3×3	1	ReLU
MaxPool	512	2×2	2	-
FC	4096	-	-	ReLU
FC	4096	-	-	ReLU
FC	65	-	-	Softmax

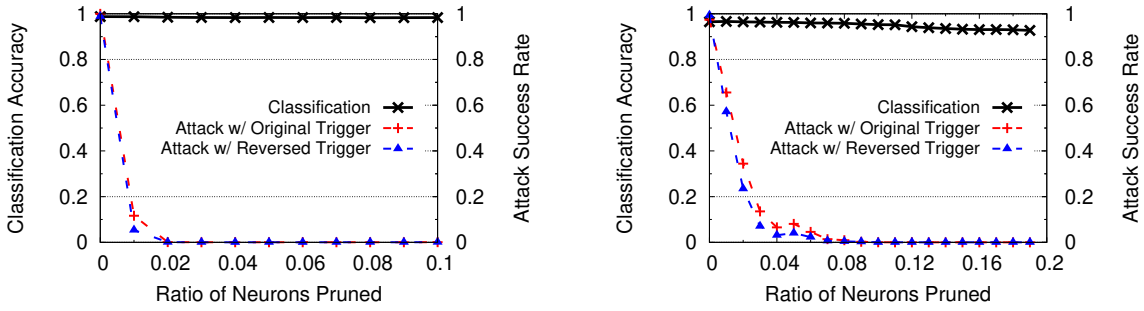
Table A.6: Model Architecture for PubFig.



(a) MNIST (b) GTSRB

(c) YouTube Face (d) PubFig

Figure A.8: Classification accuracy and attack success rate using original/reversed trigger when pruning backdoor-related neurons at the second to last layer.



(a) MNIST (b) GTSRB

(c) YouTube Face (d) PubFig

Figure A.9: Classification accuracy and attack success rate of original/reversed trigger when pruning backdoor-related neurons at the last convolution layer.

Bibliography

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *ImageNet: A Large-Scale Hierarchical Image Database*, in *CVPR09*, 2009.

[2] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman, *Vggface2: A dataset for recognising faces across pose and age*, in *Proc. of FG*, 2018.

[3] J. F. Gemmeke, D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, *Audio set: An ontology and human-labeled dataset for audio events*, in *Proc. of ICASSP*, 2017.

[4] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, *One billion word benchmark for measuring progress in statistical language modeling*, *arXiv:1312.3005* (2013).

[5] K. Winstein and H. Balakrishnan, *Tcp ex machina: Computer-generated congestion control*, in *Proc. of SIGCOMM*, 2013.

[6] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *Proc. of CVPR*, 2016.

[7] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et. al.*, *Google's neural machine translation system: Bridging the gap between human and machine translation*, *arXiv preprint arXiv:1609.08144* (2016).

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

[9] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, *Inception-v4, inception-resnet and the impact of residual connections on learning.*, in *AAAI*, 2017.

[10] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, *Densely connected convolutional networks*, in *Proc. of CVPR*, 2017.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et. al.*, *Mastering the game of go without human knowledge*, *Nature* **550** (2017), no. 7676 354.

[12] "Applying machine learning science to Facebook products." https://research.fb.com/category/machine-learning/.

[13] "AI is transforming Google Search. The rest of the web is next.." https://www.wired.com/2016/02/ai-is-changing-the-technology-behind-google-searches/, 2016.

[14] "Generating Recommendations at Amazon Scale with Apache Spark and Amazon DSSTNE.." https://aws.amazon.com/blogs/big-data/generating-recommendations-at-amazon-scale-with-apache-spark-and-amazon-dsstne/ 2016.

[15] "The AIEQ Exchange Traded Fund.." https://www.aieqetf.com/.

[16] "How AI And Machine Learning Are Used To Transform The Insurance Industry.." https://www.forbes.com/sites/bernardmarr/2017/10/24/how-ai-and-machine-learning-are-used-to-transform-the-insurance-industry/#3b899ff713a1, 2017.

[17] A. L. Buczak and E. Guven, *A survey of data mining and machine learning methods for cyber security intrusion detection*, *IEEE Communications Surveys & Tutorials* **18** (2016), no. 2 1153–1176.

[18] C. Nobata, J. Tetreault, A. Thomas, Y. Mehdad, and Y. Chang, *Abusive language detection in online user content*, in *Proc. of WWW*, 2016.

[19] "Inside Waymo's strategy to grow the best brains for self-driving cars.." https://www.theverge.com/2018/5/9/17307156/google-waymo-driverless-cars-deep-learning-neural-net-interview, 2018.

[20] "Deep Learning for Siri's Voice: On-device Deep Mixture Density Networks for Hybrid Unit Selection Synthesis.." https://machinelearning.apple.com/2017/08/06/siri-voices.html, 2017.

[21] "The Scalable Neural Architecture behind Alexa's Ability to Select Skills.." https://developer.amazon.com/blogs/alexa/post/4e6db03f-6048-4b62-ba4b-6544da9ac440/the-scalable-neural-architecture-behind-alexa-s-ability-to-arbitrate-skills, 2018.

[22] "State-of-the-Art AI: Building Tomorrow's Intelligent Systems.." https://events.technologyreview.com/video/watch/peter-norvig-state-of-the-art-ai/, 2016.

[23] "LeCun vs Rahimi: Has Machine Learning Become Alchemy?.." https://medium.com/@Synced/lecun-vs-rahimi-has-machine-learning-become-alchemy-21cb1557920d, 2017.

[24] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, *Intriguing properties of neural networks*, *arXiv preprint arXiv:1312.6199* (2013).

[25] N. Carlini and D. Wagner, *Towards evaluating the robustness of neural networks*, in *Proc. of S&P*, 2017.

[26] A. Athalye, N. Carlini, and D. Wagner, *Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples*, in *Proc. of ICML*, 2018.

[27] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, *Trojaning attack on neural networks*, in *Proc. of NDSS*, 2018.

[28] T. Gu, B. Dolan-Gavitt, and S. Garg, *Badnets: Identifying vulnerabilities in the machine learning model supply chain*, in *Proc. of Machine Learning and Computer Security Workshop*, 2017.

[29] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng, *Opprentice: Towards practical and automatic anomaly detection through machine learning*, in *Proc. of IMC*, 2015.

[30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, *ImageNet Large Scale Visual Recognition Challenge*, *IJCV* **115** (2015), no. 3 211–252.

[31] R. Cohen, K. Erez, D. ben Avraham, and S. Havlin, *Breakdown of the internet under intentional attack*, *Physical Review Letters* **86** (2001) 3682–5.

[32] S. Saroiu, K. Gummadi, and S. D. Gribble, *A measurement study of peer-to-peer file sharing systems*, in *Proc. of MMCN*, 2002.

[33] L. Zhang, D. Choffnes, T. Dumitras, D. Levin, A. Mislove, A. Schulman, and C. Wilson, *Analysis of ssl certificate reissues and revocations in the wake of heartbleed*, in *Proc. of IMC*, 2014.

[34] B. Delamore and R. K. L. Ko, *A global, empirical analysis of the shellshock vulnerability in web applications*, in *Proc. of ISPA*, 2015.

[35] A. L. Samuel, *Some studies in machine learning using the game of checkers*, *IBM Journal of research and development* **3** (1959), no. 3 210–229.

[36] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, *Mathematics of control, signals and systems* **2** (1989), no. 4 303–314.

[37] K. Hornik, *Approximation capabilities of multilayer feedforward networks*, *Neural networks* **4** (1991), no. 2 251–257.

[38] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein, *Poison frogs! targeted clean-label poisoning attacks on neural networks*, *arXiv preprint arXiv:1804.00792* (2018).

[39] O. Suciu, R. Mărginean, Y. Kaya, H. Daumé III, and T. Dumitraş, *When does machine learning fail? generalized transferability for evasion and poisoning attacks*, in *Proc. of USENIX Security*, 2018.

[40] B. Biggio, B. Nelson, and P. Laskov, *Poisoning attacks against support vector machines*, in *Proc. of ICML*, 2012.

[41] Y. Cao, A. F. Yu, A. Aday, E. Stahl, J. Merwine, and J. Yang, *Efficient repair of polluted machine learning systems via causal unlearning*, in *Proc. of ASIACCS*, 2018.

[42] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, *Manipulating machine learning: Poisoning attacks and countermeasures for regression learning*, in *Proc. of IEEE S&P*, 2018.

[43] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, *Targeted backdoor attacks on deep learning systems using data poisoning*, *arXiv preprint arXiv:1712.05526* (2017).

[44] P. Sarkar, D. Chakrabarti, and M. I. Jordan, *Nonparametric link prediction in dynamic networks*, in *Proc. of ICML*, 2012.

[45] Q. Liu, S. Tang, X. Zhang, X. Zhao, B. Y. Zhao, and H. Zheng, *Network growth and link prediction through an empirical lens*, in *Proc. of IMC*, 2016.

[46] G. Wang, B. Wang, T. Wang, A. Nika, H. Zheng, and B. Y. Zhao, *Whispers in the dark: Analysis of an anonymous social network*, in *Proc. of IMC*, 2014.

[47] G. Wang, X. Zhang, S. Tang, H. Zheng, and B. Y. Zhao, *Unsupervised clickstream clustering for user behavior analysis*, in *Proc. of CHI*, 2016.

[48] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, *An experimental study of the learnability of congestion control*, in *Proc. of SIGCOMM*, 2014.

[49] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, *Fingerprinting the datacenter: Automated classification of performance crises*, in *Proc. of EuroSys*, 2010.

[50] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, *Detecting large-scale system problems by mining console logs*, in *Proc. of SOSP*, 2009.

[51] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker, *Netprints: Diagnosing home network misconfigurations using shared knowledge*, in *Proc. of NSDI*, 2009.

[52] L. Bottou and C.-J. Lin, *Support vector machine solvers*, *Large scale kernel machines* (2007) 301–320.

[53] A. Asuncion and D. Newman, "UCI machine learning repository." `http://archive.ics.uci.edu/ml`, 2007.

[54] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida, *Detecting spammers on twitter*, in *Proc. of CEAS*, 2010.

[55] G. Wang, T. Konolige, C. Wilson, X. Wang, H. Zheng, and B. Y. Zhao, *You are how you click: Clickstream analysis for sybil detection*, in *Proc. of Usenix Security*, 2013.

[56] D. J. Whellan, R. H. Tuttle, E. J. Velazquez, L. K. Shaw, J. G. Jollis, W. Ellis, C. M. O'connor, R. M. Califf, and S. Borges-Neto, *Predicting significant coronary artery disease in patients with left ventricular dysfunction*, *American heart journal* **152** (2006), no. 2 340–347.

[57] M. C. Brouwer, A. R. Tunkel, and D. van de Beek, *Epidemiology, diagnosis, and antimicrobial treatment of acute bacterial meningitis*, *Clinical microbiology reviews* **23** (2010), no. 3 467–492.

[58] H. Costa, L. H. de Campos Merschmann, F. Barth, and F. Benevenuto, *Pollution, bad-mouthing, and local marketing: The underground of location-based social networks*, *Elsevier Information Sciences* (2014).

[59] H. Costa, F. Benevenuto, and L. H. Merschmann, *Detecting tip spam in location-based social networks*, in *Proc. of SAC*, 2013.

[60] F. E. Harrell, *Va lung cancer dataset*, 2006.

[61] F. E. Harrell, *Very low birth weight infants dataset*, 2002.

[62] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, *Do we need hundreds of classifiers to solve real world classification problems*, *Journal of Machine Learning Research* **15** (2014), no. 1 3133–3181.

[63] N. Macià and E. Bernadó-Mansilla, *Towards UCI+: A mindful repository design*, *Information Sciences* **261** (2014) 237–262.

[64] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[65] L. Breiman, *Bagging predictors*, *Machine learning* **24** (1996), no. 2 123–140.

[66] L. Breiman, *Random forests*, *Machine learning* **45** (2001), no. 1 5–32.

[67] Y. Freund and R. E. Schapire, *Large margin classification using the perceptron algorithm*, *Machine learning* **37** (1999), no. 3 277–296.

[68] R. Herbrich, T. Graepel, and C. Campbell, *Bayes point machines*, *Journal of Machine Learning Research* **1** (2001), no. Aug 245–279.

[69] J. H. Friedman, *Stochastic gradient boosting*, *Computational Statistics and Data Analysis* **38** (2002), no. 4 367–378.

[70] J. Shotton, T. Sharp, P. Kohli, S. Nowozin, J. Winn, and A. Criminisi, *Decision jungles: Compact and rich models for classification*, in *Proc. of NIPS*, 2013.

[71] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, *Cognitive modeling* **5** (1988), no. 3 1.

[72] R. Caruana and A. Niculescu-Mizil, *An empirical comparison of supervised learning algorithms*, in *Proc. of ICML*, 2006.

[73] R. Caruana, N. Karampatziakis, and A. Yessenalina, *An empirical evaluation of supervised learning in high dimensions*, in *Proc. of ICML*, 2008.

[74] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. CRC Press, 2003.

[75] N. Macià, E. Bernadó-Mansilla, A. Orriols-Puig, and T. K. Ho, *Learner excellence biased by data set selection: A case for data characterisation and artificial data sets*, *Pattern Recognition* **46** (2013), no. 3 1054–1066.

[76] J.-P. Vert, K. Tsuda, and B. Schölkopf, *A primer on kernel methods*, *Kernel Methods in Computational Biology* (2004) 35–70.

[77] S. Chan, T. Stone, K. P. Szeto, and K. H. Chan, *Predictionio: a distributed machine learning server for practical software development*, in *Proc. of CIKM*, 2013.

[78] M. Ribeiro, K. Grolinger, and M. A. Capretz, *Mlaas: Machine learning as a service*, in *Proc. of ICMLA*, 2015.

[79] M. Fredrikson, S. Jha, and T. Ristenpart, *Model inversion attacks that exploit confidence information and basic countermeasures*, in *Proc. of CCS*, 2015.

[80] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, *Membership inference attacks against machine learning models*, in *Proc. of IEEE S&P*, 2017.

[81] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, *Stealing machine learning models via prediction apis*, in *Proc. of USENIX Security*, 2016.

[82] J. Vanschoren, H. Blockeel, B. Pfahringer, and G. Holmes, *Experiment databases*, *Machine Learning* **87** (2012), no. 2 127–158.

[83] R. Maclin and D. Opitz, *An empirical evaluation of bagging and boosting*, in *Proc. of AAAI*, 1997.

[84] C. Perlich, F. Provost, and J. S. Simonoff, *Tree induction vs. logistic regression: A learning-curve analysis*, *Journal of Machine Learning Research* **4** (2003), no. Jun 211–255.

[85] D. Opitz and R. Maclin, *Popular ensemble methods: An empirical study*, *Journal of Artificial Intelligence Research* **11** (1999) 169–198.

[86] R. Leite, P. Brazdil, and J. Vanschoren, *Selecting classification algorithms with active testing*, in *Proc. of MLDM*, 2012.

[87] M. R. Smith, L. Mitchell, C. Giraud-Carrier, and T. Martinez, *Recommending learning algorithms and their associated hyperparameters*, in *Proc. of MLAS*, 2014.

[88] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag, *Collaborative hyperparameter tuning*, in *Proc. of ICML*, 2013.

[89] P. B. Brazdil, C. Soares, and J. P. Da Costa, *Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results*, *Machine Learning* **50** (2003), no. 3 251–277.

[90] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, *Algorithms for hyper-parameter optimization*, in *Proc. of NIPS*, 2011.

[91] J. Bergstra and Y. Bengio, *Random search for hyper-parameter optimization*, *Journal of Machine Learning Research* **13** (2012), no. Feb 281–305.

[92] J. Snoek, H. Larochelle, and R. P. Adams, *Practical bayesian optimization of machine learning algorithms*, in *Proc. of NIPS*, 2012.

[93] F. Hutter, H. H. Hoos, and K. Leyton-Brown, *Sequential model-based optimization for general algorithm configuration*, in *Proc. of LION*, 2011.

[94] M. Feurer, J. T. Springenberg, and F. Hutter, *Initializing bayesian hyperparameter optimization via meta-learning*, in *Proc. of AAAI*, 2015.

[95] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown, *Towards an empirical foundation for assessing bayesian optimization of hyperparameters*, in *Proc. of NIPS*, 2013.

[96] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, *Auto-weka: Combined selection and hyperparameter optimization of classification algorithms*, in *Proc. of KDD*, 2013.

[97] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, *Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka*, *Journal of Machine Learning Research* **17** (2016) 1–5.

[98] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, *Efficient and robust automated machine learning*, in *Proc. of NIPS*, 2015.

[99] S. L. Salzberg, *On comparing classifiers: Pitfalls to avoid and a recommended approach*, *Data mining and knowledge discovery* **1** (1997), no. 3 317–328.

[100] K. Wagstaff, *Machine learning that matters*, in *Proc. of ICML*, 2012.

[101] E. Keogh and S. Kasetty, *On the need for time series data mining benchmarks: a survey and empirical demonstration*, *Data Mining and knowledge discovery* **7** (2003), no. 4 349–371.

[102] T. Hothorn, F. Leisch, A. Zeileis, and K. Hornik, *The design and analysis of benchmark experiments*, *Journal of Computational and Graphical Statistics* **14** (2005), no. 3 675–699.

[103] M. J. Eugster, T. Hothorn, and F. Leisch, *Domain-based benchmark experiments: Exploratory and inferential analysis*, *Austrian Journal of Statistics* **41** (2016), no. 1 5–26.

[104] T. G. Dietterich, *Approximate statistical tests for comparing supervised classification learning algorithms*, *Neural computation* **10** (1998), no. 7 1895–1923.

[105] J. Demšar, *Statistical comparisons of classifiers over multiple data sets*, *Journal of Machine learning research* **7** (2006), no. Jan 1–30.

[106] S. Garcia and F. Herrera, *An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons*, *Journal of Machine Learning Research* **9** (2008), no. Dec 2677–2694.

[107] M. Goebel and L. Gruenwald, *A survey of data mining and knowledge discovery software tools*, *ACM SIGKDD explorations newsletter* **1** (1999), no. 1 20–33.

[108] A. H. Wahbeh, Q. A. Al-Radaideh, M. N. Al-Kabi, and E. M. Al-Shawakfa, *A comparison study between data mining tools over some classification methods*, *International Journal of Advanced Computer Science and Applications* (2011) 18–26.

[109] I. H. Witten, E. Frank, L. E. Trigg, M. A. Hall, G. Holmes, and S. J. Cunningham, *Weka: Practical machine learning tools and techniques with java implementations*, 1999.

[110] F. Van Der Heijden, R. Duin, D. De Ridder, and D. M. Tax, *Classification, parameter estimation and state estimation: an engineering approach using MATLAB*. John Wiley & Sons, 2005.

[111] J. Alcalá-Fdez, L. Sánchez, S. Garcia, M. J. del Jesus, S. Ventura, J. M. Garrell, J. Otero, C. Romero, J. Bacardit, V. M. Rivas, *et. al.*, *Keel: A software tool to assess evolutionary algorithms for data mining problems*, *Soft Computing-A Fusion of Foundations, Methodologies and Applications* **13** (2009), no. 3 307–318.

[112] S. Shi, Q. Wang, P. Xu, and X. Chu, *Benchmarking state-of-the-art deep learning software tools*, *International Conference on Cloud Computing and Big Data* (2016) 99–104.

[113] R. C. Holte, *Very simple classification rules perform well on most commonly used datasets*, *Machine learning* **11** (1993), no. 1 63–90.

[114] J. Luengo and F. Herrera, *An automatic extraction method of the domains of competence for learning classifiers using data complexity measures*, *Knowledge and Information Systems* **42** (2015), no. 1 147–180.

[115] L. Morán-Fernández, V. Bolón-Canedo, and A. Alonso-Betanzos, *Can classification performance be predicted by complexity measures? A study using microarray data*, *Knowledge and Information Systems* (2016) 1–24.

[116] J. Zubek and D. M. Plewczynski, *Complexity curve: a graphical measure of data complexity and classifier performance*, *PeerJ Computer Science* **2** (2016) e76.

[117] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent, *The difficulty of training deep architectures and the effect of unsupervised pre-training*, in *Proc. of AISTATS*, 2009.

[118] "`http://www.robots.ox.ac.uk/~vgg/software/vgg_face/`." VGG Face Descriptor.

[119] A. Kurakin, I. Goodfellow, and S. Bengio, *Adversarial machine learning at scale*, in *Proc. of ICLR*, 2017.

[120] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, *The limitations of deep learning in adversarial settings*, in *Proc. of EuroS&P*, 2016.

[121] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, *Deepfool: a simple and accurate method to fool deep neural networks*, in *Proc. of CVPR*, 2016.

[122] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, *Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition*, in *Proc. of CCS*, 2016.

[123] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, *Practical black-box attacks against machine learning*, in *Proc. of Asia CCS*, 2017.

[124] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, *Image quality assessment: from error visibility to structural similarity*, *IEEE Trans. on Image Processing* **13** (2004), no. 4 600–612.

[125] Z. Wang, E. P. Simoncelli, and A. C. Bovik, *Multiscale structural similarity for image quality assessment*, in *ACSSC*, vol. 2, pp. 1398–1402, IEEE, 2003.

[126] J. Nocedal and S. Wright, *Numerical optimization, series in operations research and financial engineering*, *Springer, New York, USA, 2006* (2006).

[127] M. D. Zeiler, *Adadelta: an adaptive learning rate method*, *arXiv preprint arXiv:1212.5701* (2012).

[128] O. M. Parkhi, A. Vedaldi, A. Zisserman, *et. al.*, *Deep face recognition.*, in *Proc. of BMVC*, 2015.

[129] "`http://vision.seas.harvard.edu/pubfig83/`." PubFig83: A resource for studying face recognition in personal photo collections.

[130] "`http://biometrics.idealtest.org/`." CASIA Iris Dataset.

[131] "`http://benchmark.ini.rub.de/?section=gtsrb&subsection=news`." The German Traffic Sign Recognition Benchmark.

[132] "`https://docs.microsoft.com/en-us/cognitive-toolkit/Build-your-own-image-classifier-using-Transfer-Learning`." Build your own image classifier using transfer learning.

[133] "`http://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html`." 102 Category Flower Dataset.

[134] F. Chollet *et. al.*, "Keras." `https://keras.io`, 2015.

[135] *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[136] N. Papernot, N. Carlini, I. Goodfellow, R. Feinman, F. Faghri, A. Matyasko, K. Hambardzumyan, Y.-L. Juang, A. Kurakin, R. Sheatsley, A. Garg, and Y.-C. Lin, *cleverhans v2.0.0: an adversarial machine learning library*, *arXiv* (2017).

[137] Y. Liu, X. Chen, C. Liu, and D. Song, *Delving into transferable adversarial examples and black-box attacks*, in *Proc. of ICLR*, 2016.

[138] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, in *Proc. of ECCV*, 2014.

[139] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, *et. al.*, *Going deeper with convolutions*, in *Proc. of CVPR*, 2015.

[140] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, *arXiv preprint arXiv:1704.04861* (2017).

[141] "`http://www.robots.ox.ac.uk/~vgg/data/flowers/17/index.html`." 17 Category Flower Dataset.

[142] C. Gini, *Italian: Variabilità e mutabilità (variability and mutability)*, *Cuppini, Bologna* (1912).

[143] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, *Complexity vs. performance: empirical analysis of machine learning as a service*, in *Proc. of IMC*, 2017.

[144] "`https://codelabs.developers.google.com/codelabs/cpb102-txf-learning/index.html#0`." Image Classification Transfer Learning with Inception v3.

[145] "`https://www.tensorflow.org/versions/r0.12/how_tos/image_retraining/`." How to Retrain Inception's Final Layer for New Categories.

[146] "`http://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`." PyTorch transfer learning tutorial.

[147] "`https://cloud.google.com/blog/big-data/2017/08/how-aucnet-leveraged-tensorflow-to-transform-their-it-engineers-into-machine-le`" How Aucnet leveraged TensorFlow to transform their IT engineers into machine learning engineers.

[148] N. Carlini and D. Wagner, *Adversarial examples are not easily detected: Bypassing ten detection methods*, in *Proc. of AISec*, 2017.

[149] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, *Detecting adversarial samples from artifacts*, *arXiv preprint arXiv:1703.00410* (2017).

[150] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: a simple way to prevent neural networks from overfitting.*, *JMLR* **15** (2014), no. 1 1929–1958.

[151] S. Zheng, Y. Song, T. Leung, and I. Goodfellow, *Improving the robustness of deep neural networks via stability training*, in *Proc. of CVPR*, 2016.

[152] M. Abbasi and C. Gagné, *Robustness to adversarial examples through an ensemble of specialists*, in *Proc. of Workshop on ICLR*, 2017.

[153] W. He, J. Wei, X. Chen, N. Carlini, and D. Song, *Adversarial example defenses: Ensembles of weak defenses are not strong*, in *Proc. of USENIX Workshop on Offensive Technologies*, 2017.

[154] J.-C. Chen, R. Ranjan, A. Kumar, C.-H. Chen, V. M. Patel, and R. Chellappa, *An end-to-end system for unconstrained face verification with deep convolutional neural networks*, in *Proc. of Workshop on ICCV*, 2015.

[155] S. Ren, K. He, R. Girshick, and J. Sun, *Faster r-cnn: Towards real-time object detection with region proposal networks*, in *Proc. of NIPS*, 2015.

[156] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You only look once: Unified, real-time object detection*, in *Proc. of CVPR*, 2016.

[157] S. Caelles, K.-K. Maninis, J. Pont-Tuset, L. Leal-Taixé, D. Cremers, and L. Van Gool, *One-shot video object segmentation*, in *Proc. of CVPR*, 2017.

[158] J. Kunze, L. Kirsch, I. Kurenkov, A. Krug, J. Johannsmeier, and S. Stober, *Transfer learning for speech recognition on a budget*, in *Proc. of RepL4NLP*, 2017.

[159] D. Wang and T. F. Zheng, *Transfer learning for speech and language processing*, in *Proc. of APSIPA*, 2015.

[160] G. Heigold, V. Vanhoucke, A. Senior, P. Nguyen, M. Ranzato, M. Devin, and J. Dean, *Multilingual acoustic models using distributed deep neural networks*, in *Proc. of ICASSP*, 2013.

[161] D. C. Cireşan, U. Meier, and J. Schmidhuber, *Transfer learning for latin and chinese characters with deep neural networks*, in *Proc of IJCNN*, 2012.

[162] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, *et. al.*, *Google's multilingual neural machine translation system: enabling zero-shot translation*, in *Proc. of ACL*, 2017.

[163] T. Mikolov, Q. V. Le, and I. Sutskever, *Exploiting similarities among languages for machine translation*, *arXiv preprint arXiv:1309.4168* (2013).

[164] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, *How transferable are features in deep neural networks?*, in *Proc. of NIPS*, 2014.

[165] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, *Cnn features off-the-shelf: an astounding baseline for recognition*, in *Proc. of Workshop on CVPR*, 2014.

[166] S. Sabour, Y. Cao, F. Faghri, and D. J. Fleet, *Adversarial manipulation of deep representations*, in *Proc. of ICLR*, 2015.

[167] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song, *Robust Physical-World Attacks on Deep Learning Models*, in *arXiv preprint 1707.08945*, 2017.

[168] A. Kurakin, I. Goodfellow, and S. Bengio, *Adversarial examples in the physical world*, in *Proc. of ICLR*, 2016.

[169] A. Nguyen, J. Yosinski, and J. Clune, *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images*, in *Proc. of CVPR*, 2015.

[170] F. Tramèr, F. Zhang, A. Juels, M. Reiter, and T. Ristenpart, *Stealing machine learning models via prediction apis*, in *Proc. of USENIX Security*, 2016.

[171] N. Papernot, P. McDaniel, and I. Goodfellow, *Transferability in machine learning: from phenomena to black-box attacks using adversarial samples*, *arXiv preprint arXiv:1605.07277* (2016).

[172] F. Tramèr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, *The space of transferable adversarial examples*, *arXiv preprint arXiv:1704.03453* (2017).

[173] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, *On detecting adversarial perturbations*, in *Proc. of ICLR*, 2017.

[174] Q. Wang, W. Guo, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles, *Adversary resistant deep neural networks with an application to malware detection*, in *Proc. of KDD*, 2017.

[175] D. Hendrycks and K. Gimpel, *Early methods for detecting adversarial images*, in *ICLR Workshop Track*, 2017.

[176] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, *On the (statistical) detection of adversarial examples*, *arXiv preprint arXiv:1702.06280* (2017).

[177] X. Li and F. Li, *Adversarial examples detection in deep networks with convolutional filter statistics*, *arXiv preprint arXiv:1612.07767* (2016).

[178] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, *Distillation as a defense to adversarial perturbations against deep neural networks*, in *Proc. of S&P*, 2016.

[179] Q. Wang, W. Guo, K. Zhang, A. G. O. II, X. Xing, X. Liu, and C. L. Giles, *Adversary resistant deep neural networks with an application to malware detection*, in *Proc. of KDD*, 2017.

[180] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, *Drebin: Effective and explainable detection of an- droid malware in your pocket*, in *Proc. of NDSS*, 2014.

[181] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, *Neural nets can learn function type signatures from binaries*, in *Proc. of USENIX Security*, 2017.

[182] E. C. R. Shin, D. Song, and R. Moazzezi, *Recognizing functions in binaries with neural networks*, in *Proc. of USENIX Security*, 2015.

[183] H. Debar, M. Becker, and D. Siboni, *A neural network component for an intrusion detection system*, in *Proc. of IEEE S&P*, 1992.

[184] C. Wierzynski, "The Challenges and Opportunities of Explainable AI." `https://ai.intel.com/the-challenges-and-opportunities-of-explainable-ai`, Jan., 2018.

[185] "FICO's Explainable Machine Learning Challenge." `https://community.fico.com/s/explainable-machine-learning-challenge`, 2018.

[186] Z. C. Lipton, *The mythos of model interpretability*, in *ICML Workshop on Human Interpretability in Machine Learning*, 2016.

[187] S. M. Lundberg and S.-I. Lee, *A unified approach to interpreting model predictions*, in *Proc. of NIPS*, 2017.

[188] S. Bach, A. Binder, G. Montavon, F. Klauschen, K. R. Muller, and W. Samek, *On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation*, *PloS One* **10** (July, 2015).

[189] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, *Lemna: Explaining deep learning based security applications*, in *Proc. of CCS*, 2018.

[190] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, *Intriguing properties of neural networks*, in *Proc. of ICLR*, 2014.

[191] K. Liu, B. Dolan-Gavitt, and S. Garg, *Fine-pruning: Defending against backdooring attacks on deep neural networks*, in *Proc. of RAID*, 2018.

[192] Y. Liu, Y. Xie, and A. Srivastava, *Neural trojans*, in *Proc. of ICCD*, 2017.

[193] "`https://cloud.google.com/automl/#features`." Google Cloud AutoML Features.

[194] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, *Understanding neural networks through deep visualization*, *arXiv preprint arXiv:1506.06579* (2015).

[195] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014).

[196] F. R. Hampel, *The influence curve and its role in robust estimation*, *Journal of the American Statistical Association* **69** (1974), no. 346 383–393.

[197] P. J. Rousseeuw and C. Croux, *Alternatives to the median absolute deviation*, *Journal of the American Statistical association* **88** (1993), no. 424 1273–1283.

[198] "`https://www.cs.tau.ac.il/~wolf/ytfaces/`." YouTube Faces DB.

[199] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, *et. al.*, *Learning algorithms for classification: A comparison on handwritten digit recognition*, *Neural networks: the statistical mechanics perspective* **261** (1995) 276.

[200] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, *Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition*, *Neural Networks* (2012).

[201] Y. Sun, X. Wang, and X. Tang, *Deep learning face representation from predicting 10,000 classes*, in *Proc. of CVPR*, 2014.

[202] "`http://www.cs.columbia.edu/CAVE/databases/pubfig/`." PubFig: Public Figures Face Database.

[203] "`http://www.robots.ox.ac.uk/~vgg/data/vgg_face/`." VGG Face Dataset.

[204] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, *Network trimming: A data-driven neuron pruning approach towards efficient deep architectures*, *arXiv preprint arXiv:1607.03250* (2016).

[205] B. Wang and N. Z. Gong, *Stealing hyperparameters in machine learning*, in *Proc. of S&P*, 2018.

[206] J. Clements and Y. Lao, *Hardware trojan attacks on neural networks*, *arXiv preprint arXiv:1806.05768* (2018).

[207] W. Li, J. Yu, X. Ning, P. Wang, Q. Wei, Y. Wang, and H. Yang, *Hu-fu: Hardware and software collaborative attack framework against neural networks*, in *Proc. of ISVLSI*, 2018.

[208] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, *Manipulating machine learning: Poisoning attacks and countermeasures for regression learning*, in *Proc. of IEEE S&P*, 2018.

[209] B. I. Rubinstein, B. Nelson, L. Huang, A. D. Joseph, S.-h. Lau, S. Rao, N. Taft, and J. Tygar, *Antidote: understanding and defending against poisoning of anomaly detectors*, in *Proc. of IMC*, 2009.

[210] M. Mozaffari-Kermani, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, *Systematic poisoning attacks on and defenses for machine learning in healthcare*, *IEEE journal of biomedical and health informatics* **19** (2015), no. 6 1893–1905.

[211] J. Steinhardt, P. W. W. Koh, and P. S. Liang, *Certified defenses for data poisoning attacks*, in *Proc. of NIPS*, 2017.

[212] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis, *Casting out demons: Sanitizing training data for anomaly sensors*, in *Proc. of IEEE S&P*, 2008.

[213] N. Carlini and D. Wagner, *Towards evaluating the robustness of neural networks*, in *Proc. of IEEE S&P*, 2017.

[214] A. Kurakin, I. Goodfellow, and S. Bengio, *Adversarial machine learning at scale*, in *Proc. of ICLR*, 2017.

[215] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, *The limitations of deep learning in adversarial settings*, in *Proc. of Euro S&P*, 2016.

[216] Y. Liu, X. Chen, C. Liu, and D. Song, *Delving into transferable adversarial examples and black-box attacks*, in *Proc. of ICLR*, 2016.

[217] B. Wang, Y. Yao, B. Viswanath, Z. Haitao, and B. Y. Zhao, *With great training comes great vulnerability: Practical attacks against transfer learning*, in *Proc. of USENIX Security*, 2018.

[218] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, *Distillation as a defense to adversarial perturbations against deep neural networks*, in *Proc. of IEEE S&P*, 2016.

[219] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, *Towards deep learning models resistant to adversarial attacks*, in *Proc. of ICLR*, 2018.

[220] H. Kannan, A. Kurakin, and I. Goodfellow, *Adversarial logit pairing*, *arXiv preprint arXiv:1803.06373* (2018).

[221] D. Meng and H. Chen, *Magnet: a two-pronged defense against adversarial examples*, in *Proc. of CCS*, 2017.

[222] W. Xu, D. Evans, and Y. Qi, *Feature squeezing: Detecting adversarial examples in deep neural networks*, in *Proc. of NDSS*, 2018.

[223] N. Carlini and D. Wagner, *Defensive distillation is not robust to adversarial examples*, *arXiv preprint arXiv:1607.04311* (2016).

[224] W. He, J. Wei, X. Chen, N. Carlini, and D. Song, *Adversarial example defenses: Ensembles of weak defenses are not strong*, in *Proc. of WOOT*, 2017.

[225] N. Carlini and D. Wagner, *Magnet and efficient defenses against adversarial attacks are not robust to adversarial examples*, *arXiv preprint arXiv:1711.08478* (2017).

[226] A. Athalye, N. Carlini, and D. Wagner, *Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples*, in *Proc. of ICML*, 2018.

[227] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer, *Adversarial patch*, *arXiv preprint arXiv:1712.09665* (2017).

[228] M. T. Ribeiro, S. Singh, and C. Guestrin, *Why should i trust you?: Explaining the predictions of any classifier*, in *Proc. of KDD*, 2016.

[229] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, *Lemna: Explaining deep learning based security applications*, in *Proc. of CCS*, 2018.

[230] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, *Ai 2: Safety and robustness certification of neural networks with abstract interpretation*, in *Proc. of IEEE S&P*, 2018.

[231] M. Mirman, T. Gehr, and M. Vechev, *Differentiable abstract interpretation for provably robust neural networks*, in *Proc. of ICML*, 2018.

[232] G. Singh, G. Timon, M. Mirman, M. Puschel, and M. Vechev, *Fast and effective robustness certification*, in *Proc. of NIPS*, 2018.

[233] D. Hendrycks and K. Gimpel, *Early methods for detecting adversarial images*, in *Proc. of ICLR Workshop*, 2016.

[234] G. Wang, T. Wang, B. Wang, D. Sambasivan, Z. Zhang, H. Zheng, and B. Y. Zhao, *Crowds on wall street: Extracting value from collaborative investing platforms*, in *Proc. of CSCW*, 2015.

[235] B. Wang, X. Zhang, G. Wang, H. Zheng, and B. Y. Zhao, *Anatomy of a personalized livestreaming system*, in *Proc. of IMC*, 2016.

[236] G. Wang, B. Wang, T. Wang, A. Nika, H. Zheng, and B. Y. Zhao, *Whispers in the dark: analysis of an anonymous social network*, in *Proc. of IMC*, 2014.

[237] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, *Complexity vs. performance: empirical analysis of machine learning as a service*, in *Proc. of IMC*, 2017.

[238] "Amazon machine learning developer guide." `http://docs.aws.amazon.com/machine-learning/latest/dg/machinelearning-dg.pdf`.