DYNAMIC DEFENSES AND THE TRANSFERABILITY OF ADVERSARIAL EXAMPLES

by

Sam Thomas

April, 2019

Director of Thesis: Dr. Nasseh Tabrizi, PhD

Major Department: Computer Science

Adversarial machine learning has been an important area of study for the securing of machine learning systems. However, for every defense that is made to protect these artificial learners, a more sophisticated attack emerges to defeat it. This has created an arms race, with the problem of adversarial attacks never being fully mitigated. This thesis examines the field of adversarial machine learning; specifically, the property of transferability, and the use of dynamic defenses as a solution to attacks which leverage it. We show that this is an emerging field of research, which may be the solution to one of the most intractable problems in adversarial machine learning. We go on to implement a minimal experiment, demonstrating that research within this area is easily accessible. Finally, we address some of the hurdles to overcome in order to unify the disparate aspects of current related research.

DYNAMIC DEFENSES AND THE TRANSFERABILITY OF ADVERSARIAL EXAMPLES

A Thesis

Presented to the Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Masters of Science in Software Engineering

by

Sam Thomas

April, 2019

DYNAMIC DEFENSES AND THE TRANSFERABILITY OF ADVERSARIAL EXAMPLES

by

Sam Thomas

APPROVED BY:

DIRECTOR OF THESIS: _____
Dr. Nasseh Tabrizi, PhD

COMMITTEE MEMBER: _____
Dr. Venkat Gudivada, PhD

COMMITTEE MEMBER: _____
Dr. Rui Wu, PhD

CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE: _____
Dr. Venkat Gudivada, PhD

DEAN OF THE GRADUATE SCHOOL: _____
Dr. Paul J. Gemperline, PhD

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Machine learning is the process by which a computer can learn to make decisions without being explicitly told what to do. When given a set of inputs as training data, a machine learning algorithm will form a model, an internal representation, that allows it to perform some future task. We can make distinctions between the different types of learning algorithms, one of the ways being the manner in which they process their training data. More specifically, we distinguish between ‘offline’ and ‘online’ learning algorithms. An offline learner has access to all of the training data at once. An online learner only has access to a single sample at a time, and thus learns from a sequence of inputs [1, 2].

Additionally, we can categorize machine learning tasks into three different kinds: supervised, unsupervised, and semi-supervised. Speaking generally, *supervised learning* deals with labeled data, and *unsupervised learning* deals with unlabeled data. For supervised learners, “the goal is to learn a mapping from x to y , given a training set made of pairs (x_i, y_i)” [3]. It is often used for classification and regression problems. On the other hand, if we have an unlabeled dataset X, then goal of unsupervised learning is to find interesting structures in X. While it has applications in problems such as clustering and dimentionality reduction, Chapelle et. al. point out that “the problem of unsupervised learning is fundamentally that of estimating a density which is likely to have generated X”. As the name suggests, *semi-supervised learning* is somewhere in-

between supervised and unsupervised learning; it utilizes both labeled and unlabeled data.

Artificial learners are, in general, vulnerable to adversarial attacks. An entity that wishes to harm or evade an unguarded system's decision-making capabilities can do so with relative ease. This scenario is the focus of the field of adversarial machine learning [4]. More formally, adversarial machine learning is where a machine learning system is in an adversarial environment – one in which it is challenged by some adversarial opponent. For example, if the task of the learner was to classify input samples, the opponent can alter legitimate input samples in order to cause the learner to misclassify the input. In fact, machine learning systems can, and often are, trained to generate adversarial inputs to use against such a learner. These adversarial inputs are often referred to as "adversarial examples". When talking about adversarial machine learning, we will refer to the attacker as the "adversary", and the defender as the "target". Although measures can be taken to protect a machine learning system, the protection is not total and not ensured to last. This remains an open problem, largely due to the transferability of adversarial examples [5]. This is explored further in the next chapter.

Generally, adversarial machine learning is applicable wherever there is a machine learning system that is accessible to would-be attackers. Notably, it has applications in biometric verification, spam detection, malware detection, and the detection of network intrusions [4, 6]. If a classifier is not protected from adversaries, then it is very susceptible to their attacks, which can result in a misclassification rate of over 96% in some cases [5]. Online machine learning systems are susceptible to being

corrupted over time, and have been shown to become more inaccurate as the number of adversarial samples increases [7]. Additionally, it has been shown that it's possible to fool autonomous robotic patrolling by using game-theoretic approaches [8].

One way to influence learners is through the use of adversarial perturbations. An adversarial perturbation is an alteration of a sample, such that the changes results in some furthering of the adversary's goals, like in the crossing of a classification boundary, for example. These perturbations are an effective strategy at both train and test time [9].

Adversarial attacks can fall into one of three categories: evasion, exploratory, or poisoning attacks [10]. *Evasion attacks* seek to have samples misclassified by the target. Using an example of a spam filter, the adversary might be trying to get spam emails misclassified, and thus they would not be filtered out. *Exploratory attacks* seek to gain information about the target that would otherwise be hidden, such as whether or not a specific sample was used to train the target model. *Poisoning attacks* attempt to damage the integrity or performance of the target. In data stream learning this is often called *adversarial drift* [11].

There are also two types of specificity: *attack specificity* and *error specificity* [10]. The attack specificity can either be *targeted* or *indiscriminate*. A targeted attack would intend for a specific set of samples to be misclassified, whereas an indiscriminate attack does not care this. The error specificity can either be *specific* or *generic*, meaning the attacker wants the samples to be misclassified as either a specific class or any class, respectively. Continuing with our example of a spam filter: an adversary could perform a

poisoning attack that is *targeted* at all emails addressed to “human resources” which contain the text “sexual harassment”, and aims to have those emails classified *specifically* as spam. This is an example of a targeted attack with a specific error. This is also an example of an adversary who has no redeeming qualities.

This thesis explores the currently intractable problem of transferable adversarial examples and a barely explored approach that might be the solution. One of it’s goals is to help unify the various tangents of research by highlighting the many research opportunities available, as well as demonstrating that practical testing of these concepts is feasible. We echo the sentiment of other researchers calling for attention to this emerging field of study. This thesis hopes to draw that attention.

The following contributions have been made to satisfy the graduate requirements for Masters in Software Engineering:

- A published paper which surveys the current landscape of research in adversarial machine learning.
- A case is made for a dynamic approach to combat transferable adversarial examples.
- An experiment which tests the minimal case of a dynamic defense against transferable adversarial examples.
- Several deficiencies are identified within this emerging field of study.

CHAPTER 2: RELATED WORKS

In this chapter, we review research, both our own and other's, which are relevant to this discussion. We first provide an overview of our survey's findings. Then, review the work that has been done regarding transferability, as well as the emerging field of dynamic models and defenses. Finally, we review the work related to the algorithms used in our experiment.

2.1 Adversarial Machine Learning Survey

In preparation for the thesis proper, we conducted a survey [12] of the field of adversarial machine learning which adapts the categorization used in a survey done by Kumar and Mehta of IBM Research, India [13]. In it, we survey the current landscape of research in this field, and provide analysis of the overall results and of the trends in research.

2.1.1 Methodology Overview

Our process for categorizing the research papers was the following:

1. Collect the top 100 results from the sources, using the phrase 'adversarial machine learning' as the search query.
2. Review each paper.
3. Sort each paper into categories.
4. Tally the raw values of each category.

5. Tally the per-year values of each category (excluding Cornell Digital Library [https://arxiv.org/]).
6. Visualize the data with charts.

2.1.2 Collection

In this study we have used multiple collection sources to get a robust data set. The sources we used were ACM Digital Library, IEEE Xplore Digital Library, Springerlink, Cornell Digital Library, and Sciencedirect. Since we are focused on newer developments within this field of research, we restricted our data collection to the past ten years (2007-2017).

2.1.3 Evaluation

All papers were filtered and categorized manually. We filtered out book chapters, conference posters, and those papers which were unrelated to the subject matter. We used the categories in such a way, that a paper can appear in multiple categories. Some papers were categorized as a top-level category, but did not fit into any of its subcategories. We tallied these papers in their own subcategory that is independent of the other subcategories.

The histograms for each category exclude the results from Cornell Digital Library due to the fact that the earliest published papers are from 2015, and so including these results would heavily skew the graph.

2.1.4 Results

The raw paper counts for the categories are shown in *Figures 2.1 - 2.6*. *Figures 2.1 - 2.3* show the counts for top-level categories, while *Figures 2.4 - 2.6* show the counts for each type of attack.

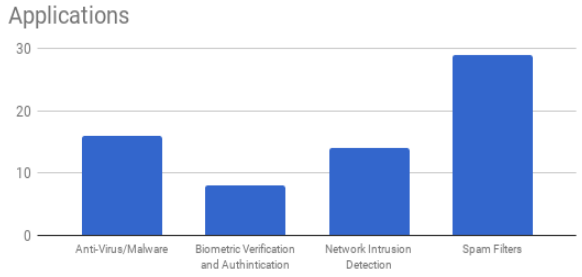

Figure 2.1: Count of papers for "Applications"

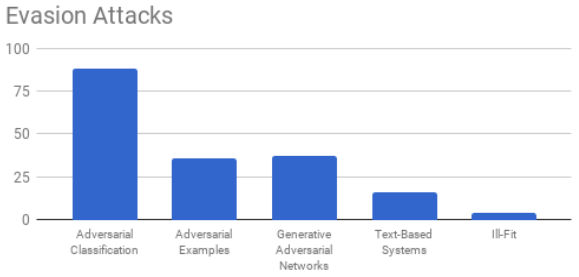

Figure 2.2: Count of papers for "Evasion Attacks" sub-category

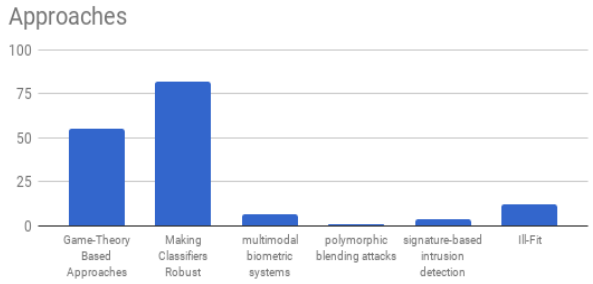

Figure 2.3: Count of papers for "Approaches"

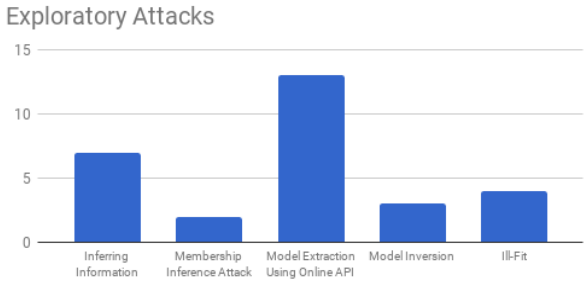

Figure 2.4: Count of papers for "Exploratory Attacks" sub-category

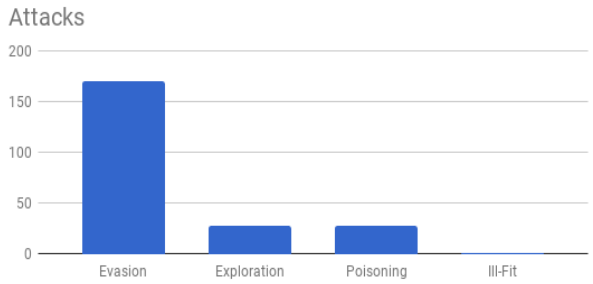

Figure 2.5: Count of papers for "Attacks"

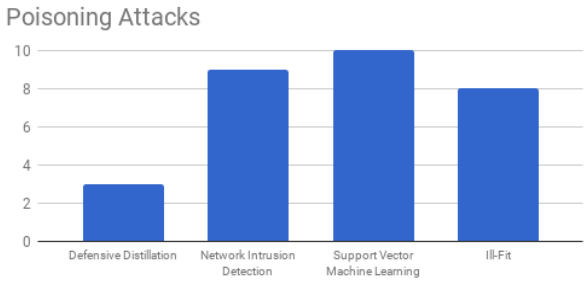

Figure 2.6: Count of papers for "Poisoning Attacks" sub-category

2.1.5 Raw Count Data Analysis

From *Figures 2.1 - 2.6* we can see the following:

- “Spam Detection” is by far the most popular application, with ~41% of papers exploring the topic.
- "Making Classifiers Robust" and "Game-Theory Based Approaches" are the most common approaches, making up approximately 51% and 34% of the papers, respectively.
- ~76% of papers discussing attacks focus on "Evasion" attacks.
- Regarding evasion attacks, the most explored topic was "Adversarial Classification", having ~49% of papers which discuss it. There were also the topics of “Adversarial Examples” and “Generative Adversarial Networks” which were somewhat popular, at ~20%, each.
- Regarding exploratory attacks, the most explored aspect was "Model Extraction Using Online APIs", which made up ~45% of papers in that category.
- For the “Ill-Fit” category, we see that “Poisoning Attacks” has the largest proportion of ill-fit papers, with ~27% falling into the sub-category. However, it is the “Approaches” category which has the highest raw count, with 12 ill-fitting papers.

2.1.6 Trends

The counts per year can be seen in *Figures 2.7 - 2.12*. These figures are a subset of all the trends which were analyzed. For the sake of readability and brevity, we have included the most noteworthy trend charts in this paper, and excluded those which had sparse or sporadic paper counts.

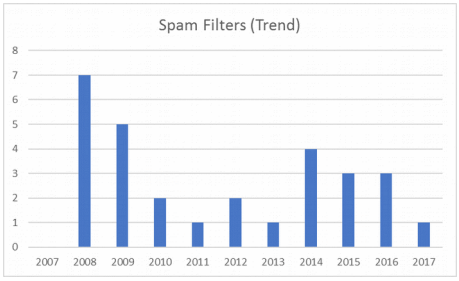

Figure 2.7: Paper counts by year for "Spam Filters".

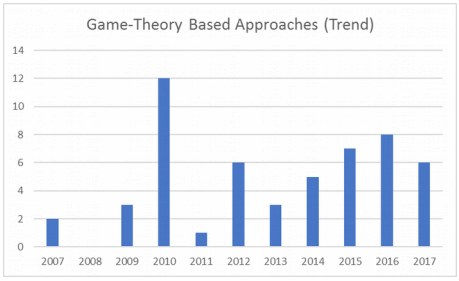

Figure 2.8: Paper counts by year for "Game-Theory Approaches".

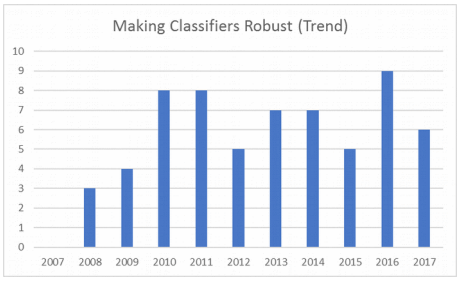

Figure 2.9: Paper counts by year for "Making Classifiers Robust".

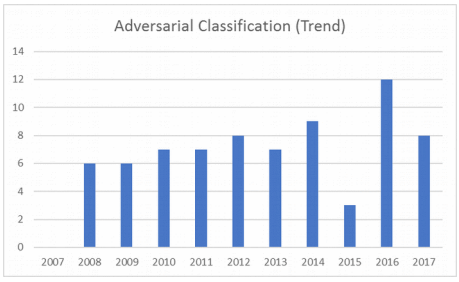

Figure 2.10: Paper counts by year for "Adversarial Classification".

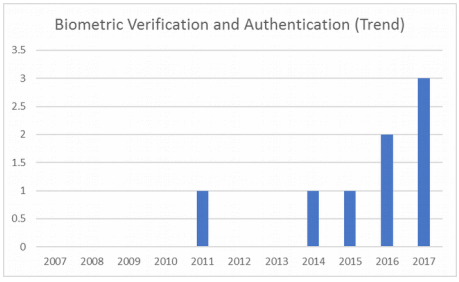

Figure 2.11: Paper counts by year for "Biometric Verification and Authentication".

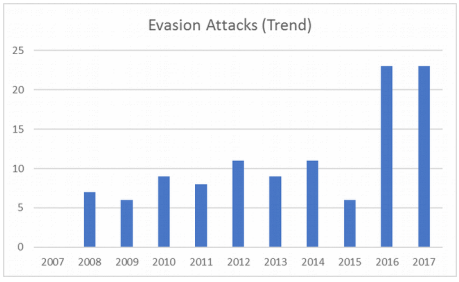

Figure 2.12: Paper counts by year for "Evasion Attacks".

2.1.7 Trends Data Analysis

By looking at the breakdown of each category's paper-counts by year, we can see the following trends in *Figures 2.7 – 2.12*:

- "Spam Filters" had a large spike in 2008, before declining, and then rising slightly in popularity.
- "Game-Theory Based Approaches" had a large spike in 2010, before lowering and remaining stable.
- "Making Classifiers Robust" has been a consistently explored topic since 2008.
- Evasion attacks have had steady interest since 2008, with a sharp rise within the past 2 years.
- "Adversarial Classification" has had steady interest since 2008.
- And finally, we see in *Figure 2.13* the 10-year trend in the topic of Adversarial Machine Learning as a whole.

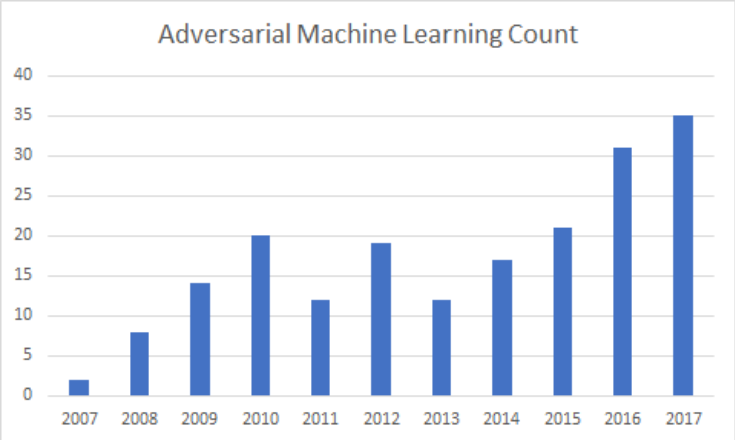

Figure 2.13: Paper counts by year for research in Adversarial Machine Learning

Together, these charts show an initial wave of interest, which initially peaks in 2010. Interest in the field levels off for a few years, but then begins to steadily gain more and more interest from 2013 onward.

2.2 Black-Box Attacks and Transferability

Szegedy et. al. first observed the transferability of adversarial samples between two deep neural networks [14]. With these findings, Papernot et. al. explored the concept further, developing several black-box attacks to demonstrate the phenomenon of "adversarial example transferability" [5]. There are three key components of this technique.

1. A target classifier is used as an oracle for the adversary. It receives unaltered samples and gives the prediction to the adversary.
2. The adversary uses these predictions to train a substitute model. By training from the target's predictions, the substitute learns an approximation of the target's decision boundaries.
3. The adversary uses white-box knowledge of the substitute to craft adversarial examples from samples in the dataset. The examples are crafted by adding adversarial perturbations in the direction of the decision boundary hyperplane.

Using this method, the adversary can now effectively evade the target's classifier, despite only having black-box access. The reason this works is because adversarial examples are transferable between architectures [6]. The property of transferability is not fully understood, though it is thought that it may be caused by learning from the same data distribution, which results in learning similar decision boundaries [15].

This method of attack is very generalizable; the attacks not only transfer to models of the same type, but also across many different types of models. In the Papernot et. al. paper, the researchers demonstrate the effectiveness of the attack with two different substitute models against oracles with the following models: decision tree, logistic regression, support vector machine, deep neural network, and nearest neighbors. They go on to show the attack's effectiveness against Amazon's and Google's Machine Learning as a Service platforms, without knowing what models or learning techniques were being employed.

Since then, many attempts have been made to solve this problem [16]. However a pattern has emerged: a new defense against these attacks is proposed, and it is shortly shown to be ineffective against a set of attacks that weren't tested [17–19]. No defense thus far has been sufficient to completely guard against the transferability of adversarial examples, and so this is an open problem.

Fortunately, the researchers have implemented an example experiment of their black-box attack in Cleverhans, which is a library for bench-marking adversarial attacks and defenses [6, 20]. This is a boon to researchers wanting to test defenses against those attacks.

2.3 Dynamic Modeling

An interesting observation can be made regarding the research into adversarial transferability: it seems to exclusively consider offline learning techniques as a defense. In their 2019 paper, Goodfellow argues that *dynamic models* should be a new research direction when examining defenses against adversarial examples at test-time [21]. He

defines a dynamic model as one which changes each time it is run. The intuition here is that a moving target is harder to hit.

Dynamic-Adversarial Mining is a new interdisciplinary field of study (see *Figure 2.14*) proposed by Sethi & Kantardzic which approaches the same problem, but from a different angle [22]. They argue that machine learning models often assume that the test environment is static, instead of looking at it like an arms race between attacker and defender. In addition, they show that the generalization of machine learning models actively opposes their security. The proposal is a holistic approach to stream data mining, which takes into account the fact that test-time environments are dynamic and adversarial.

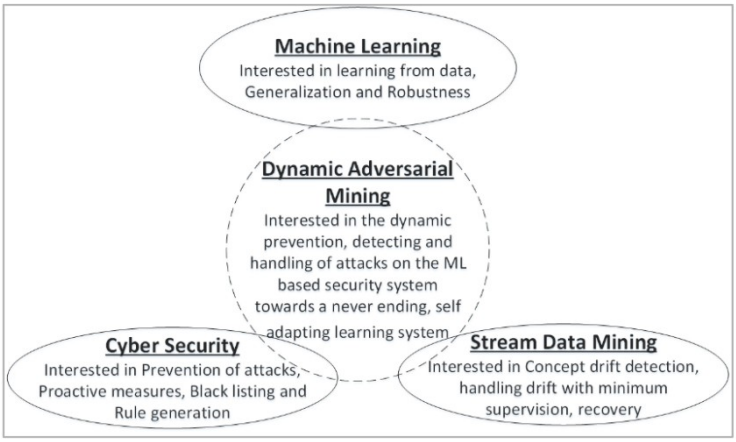

Figure 2.14: Shows the gap that this new field of study aims to fill. Adapted from [22].

In another paper, Sethi et. al. identify three views on evaluating the security of models that operate in “dynamic real world environments” [23]:

- the difficulty of evading it’s classifier
- the detect-ability of changes in the data distribution over time

- it’s ability to recover from model degradation

They propose a novel feature-hiding approach, which improves the model’s ability to detect attacks, and conclude by marking the following as core ideas under this new field of study, as quoted from the paper:

- “Ability to leave feature space honeypots in the learned classifier at training time, to efficiently detect attacks using unlabeled data at test time.
- Semi automated self aware algorithms, which can detect abnormal data distribution shifts, at test time.
- Maintaining multiple backup models, which can provide prediction when the main model gets attacked.
- Ability to use a stream labeling budget effectively and to distribute the budget appropriately, to detect and fix attacks. Thereby, managing human involvement in the process.
- Understanding attack vulnerabilities on classifiers, from a purely data driven perspective, to effectively test the security measures employed.”

2.4 Online Semi-Supervised Learning

For our experiment, we needed a model that changed every time it was run, and so an online model seemed like the natural choice. However, as Losing et. al. observe, there is a lot of ambiguity when the terms “online” or “incremental” are used in research, since they are often used interchangeably [24]. In their paper, they consider online learning to be a type of incremental learning, one which can learn indefinitely with limited computational resources available. In this sense, either an online or increment model

would work. We decided that an online model would be the more desirable choice for our needs since it seemed more realistic and applicable. The reasoning for needing a "semi-supervised" model is explained in chapter 3. We examined several candidate models. The ideal candidate would be one that was already implemented in the target language, Python; the alternative would be implementing the algorithm ourselves. In our initial search, we were unable to find a Python implementation of an online semi-supervised algorithm, likely due to the ambiguity stated earlier. We decided on implementing a recently published algorithm that seemed to fit the needs of our experiment. That algorithm was the Fast Model based Online Manifold Regularization (FMOMR) algorithm [25].

FMOMR, as the name might suggest, is a fast variant of the MOMR algorithm, which the authors proposed in a previous paper. They describe MOMR as a "manifold regularized model in a reproducing kernel Hilbert space". *Figure 2.15* shows a diagram which is an overview of how these kind of classifiers function. They use a solution to the Lagrange dual problem in order to get the classifer at the next time step, which enables the model to learn in an online manner. They use an approximate derivation of this process for FMOMR, along with a buffering strategy to prevent the number of support vectors from growing undoubtedly. For their testing, they compared both MOMR and FMOMR with two other online manifold regularized algorithms. The algorithms are based on Example-Associate Update (OMR-EA) and Overall Update (OMR-Overall). The test sets used were MNIST, FACEMIT, and the UCF YouTube dataset. In their results, they observe that the accuracies of their algorithms were comparable to the

other online algorithms, and in the case of UCF, slightly better. An implementation of this algorithm was attempted, but was not able to meet the needs of our experiment. It is uncertain whether the fault lies within the implementation or the algorithm. More on this is discussed in the next chapter.

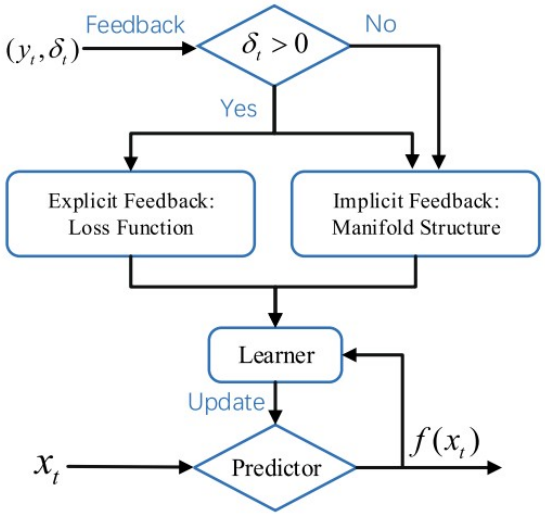

Figure 2.15: Overview of manifold regularized online semi-supervised learners. Adapted from [25].

In searching for an alternative candidate for the experiment, we found the Incremental Label Propagation (ILP) algorithm, which was already implemented in the target language [26]. To quote the authors, “Our main idea is that when a new sample arrives, many computation steps can be saved by propagating labels only locally and stopping the propagation process if no significant change of labels is achieved.” They denote a hyperparameter θ which is used for tuning the threshold at which label propagation stops. As θ is increased, the algorithm will become “constrained to act more locally”. Conversely, when θ is set to zero, it will behave the same as an offline label

propogation algorithm would. They tested their algorithm on the KITTI benchmark dataset, which is a streaming dataset known for it's challenging computer vision benchmarks. The researchers were able to outperform Online Random Forests and online multi-class Gradient Boost. In the "20% label" test case, they were also able to outperform the Mondrian Forest algorithm. Because of it's incremental learning mechanism, accessible tuning parameters, and the fact that it was already implemented in the target language, ILP was an excellent candidate for our experiment.

CHAPTER 3: CHALLENGES

In order to test the hypothesis, we would need to do the following:

1. Find a suitable algorithm
2. Verify the algorithm's correctness
3. Integrate the algorithm into the cleverhans transferability test
4. Run several experiments with varying parameters

There were many challenges associated with these steps. In this chapter, we have documented our experience with overcoming these challenges, with the hopes that it will aid future researchers in this area.

3.1 Finding a Suitable Algorithm

Due to the specific scenario being tested, we needed a learner that fit very specific requirements. Also, because we were adapting an example from the Cleverhans framework, the selection of learning algorithms was additionally constrained by integration requirements. The requirements are informally described below.

R1. The learner shall be a multi-class image classifier in order to integrate into the Cleverhans example.

R2. The learner should be one which is already implemented in Python, as it would simplify the integration with the Cleverhans framework.

R3. The learner shall learn at both train-time and test-time (online/incremental learning), as this is central to the hypothesis.

R4. The learner shall be a semi-supervised learning algorithm.

The reasons for the first three requirements are stated plainly. However, the reasoning behind R4 requires some explanation, which is done in the next chapter.

3.2 Finding an Online Semi-Supervised Image Classifier

While gathering the data for our literature survey, one thing we noticed was the scarcity of online semi-supervised algorithms. So even finding a suitable algorithm was a challenge. Of the few that were available, several were implemented in such a way that integration would have been a major hurdle, and others did not give sufficient details to implement their algorithm independently. While we didn't find any suitable algorithms implemented in Python, we did find a detailed proposal for a suitable algorithm, FMOMR. We were unable to obtain a copy of the implementation the researchers used, so we decided to implement the algorithm, ourselves.

3.3 Implementing the Algorithm

In order to implement the algorithm ourselves, we needed to understand the math behind it. While our search led us a bit too far into the theory behind the algorithm, we were able to translate it to code. Additionally, there are some small inconsistencies in the nomenclature between the descriptions MOMR and FMOMR which are never addressed, and thus added to the difficulty of translating the algorithm into code.

During this time, we were also searching for the best tools to use for this implementation. A major concern was the ease of integration with the Cleverhans code, so we prioritized simplicity and quality of documentation. The libraries we decided upon

were *scikit-learn* and *numpy* [27, 28], due to the aforementioned concerns, as well as their apparent popularity within this field of study.

Here we will go through the FMOMR algorithm (see *Figure 3.1*), as described by the authors, and discuss each line's meaning and implementation implications. Starting off, we see that the algorithm lists two of the hyperparameters as inputs, λ_1 and λ_2. In our implementation, however, we needed all four hyperparameters (λ_1, λ_2, σ_k, σ_w), and so we used a member variable dictionary to cleanly access them when and where they were needed.

Algorithm 1 Model based online manifold regularization (MOMR) algorithm

Input: Parameters:$\lambda_1 \geq 0$(default= 10^{-3}),$\lambda_2 \geq 0$(default= 10^{-3}),$C \geq 0$(default= 1)

1: Initialize: $f = 0$.
2: Receive an incoming instance:x_1;
3: Let $\alpha^1 = 1$, $f = \alpha^1 K(x_t, \cdot)$
4: **for** $t = 2; i \leq T; t++$ **do**
5: Receive an incoming instance: x_t;
6: Receive the flag δ_t and the label y_t;
7: Update the Gram Matrix K with x_t;
8: Let $\tilde{\alpha}^t = [\alpha^{t-1}; 0]$;
9: Compute D and W by (23) and (24).
10: Let $L = D - W$;
11: Compute $A = K + \lambda_1 K + \lambda_2 K L K$;
12: **if** $\delta_t == 0$ **then**
13: $\alpha^t = A^{-1} K \tilde{\alpha}^t$;
14: **else**
15: Compute $J = Ke$, where $e = [0, ..., 0, 1]^T$ is a t-dimensional vector;
16: Compute γ_t^* by (23);
17: $\alpha^t = A^{-1}(K\tilde{\alpha}^t + y_t \gamma_t^* J)$;
18: **end if**
19: $f = \sum_i^t \alpha_i^t K(x_i, \cdot)$,
20: where α_i^t is the i-th element of α^t;
21: **end for**
Output: f.

Algorithm 2 Fast algorithm to model based online manifold regularization (FMOMR)

Input: Parameters:$\lambda_1 \geq 0$(default= 10^{-3}),$\lambda_2 \geq 0$(default= 10^{-3}),$C \geq 0$(default= 1)

1: Initialize: $f = 0$.
2: Receive an incoming instance:x_1;
3: Let $\alpha^1 = 1$, $f = \alpha^1 K(x_t, \cdot)$
4: **for** $t = 2; i \leq T; t++$ **do**
5: Receive an incoming instance:x_t;
6: Receive the flag δ_t and the label y_t;
7: Update the Gram Matrix K with x_t;
8: Let $\tilde{\alpha}^t = [\alpha^{t-1}; 0]$;
9: Compute D and W by (23) and (24).
10: Let $L = D - W$;
11: **if** $\delta_t == 0$ **then**
12: $\alpha = \frac{1}{1+\lambda_1}(I - \lambda_2 L K)\alpha^t$;
13: **else**
14: Compute $J = Ke$, where $e = [0, ..., 0, 1]$ is a t-dimensional vector;
15: Compute γ_t by (17);
16: $\alpha = \frac{1}{1+\lambda_1}[(I - \lambda_2 L K)\alpha^{t-1} + e y_t \gamma_t]$;
17: **end if**
18: $f = \sum_i^t \alpha_i^t K(x_i, \cdot)$,
19: where α_i^t is the i-th element of α^t;
20: **end for**
Output: f.

Figure 3.1: MOMR and FMOMR algorithms. Adapted from [25].

Lines 1-3 represent the initialization of algorithm at time-step 1. At this point, essentially, the classifier is just the kernel function on the first input sample. *Line 4* sets up the loop, where *t* represents the current time-step. One thing to note here is "$i \leq T$".

We believe this is a typo of "$t \leq T$", since there has been no variable "*i*" declared at this point. *Lines 5-6* are just receiving the input sample, and it's labeling information. Python allows for default values in function declarations, so we were able to make the passing of y_t optional. This input, along with the loop, was trivial to implement.

In *Line 7*, the algorithm says to update the Gram Matrix *K*. However, *K* is simply a matrix of kernel functions, so we stored x_t in a buffer and, whenever *K* was needed, the matrix elements would be calculated at that time. This was useful, since the buffer was also needed to calculate the graph Laplacian. The kernel function they use is a Radial Basis Function (RBF) kernel, which looks like this: $k(xi, xj) = exp(-||x_i - x_j||^2/(2\sigma^2))$. The σ here is a hyperparameter. We implemented this function ourselves, as well as trying the implementation in scikit-learn to confirm the correctness of our implementation.

At *line 8*, we append a zero to the α buffer from the previous time-step. We also included a check for whether or not the α buffer was full, and shifted the buffer as necessary.

Lines 9-10 are for calculating the graph Laplacian. In *line 9*, we compute the weight and degree matrices, according to the author's specification in *Figure 3.2*. The weight function they use for the elements in weight matrix *W* is also an RBF kernel. The only difference between this and the Gram Matrix *K* is the sigma hyperparameter, which is represented by σ_k and σ_w, respectively. Because of this, it was easy to implement a function that worked for both matrices; we defined the method with an input variable "sigma", along with the two sample inputs, as seen in *Figure 3.3*. In the code snippet, "np" is the numpy library, whose math functions were a great aid to the implementation

of this algorithm. In addition, the built-in matrix support of numpy's “ndarray” datatype allows for simple, understandable matrix math. Instead of calling a function to subtract W from D, we can just write "L = D - W", same as in *line 10*.

$$\overline{\gamma}_{t+1} = 1 + \lambda_1 - y_{t+1} J^T (I - \lambda_2 L K) \alpha^t \quad (17)$$

$$D_{ij} = \begin{cases} w_{ij} & \text{if} \quad 0 < i = j < t+1 \\ \sum_{i=1}^{t} w_{it+1} & \text{if} \quad i = j = t+1 \\ 0 & \text{otherwise} \end{cases} \quad (23)$$

$$W_{ij} = \begin{cases} w_{ij} & \text{if} \quad 0 < i < t+1, j = t+1 \\ w_{ij} & \text{if} \quad i = t+1, 0 < j < t+1 \\ 0 & \text{otherwise} \end{cases} \quad (24)$$

Figure 3.2: Equations 17,23, and 24. Adapted from [25].

```
@staticmethod
def kernel(x1, x2, sigma):
  rbfGamma = 0.5 / (sigma**2)
  dist = x1-x2
  norm = np.linalg.norm(dist)
  gauss = -1 * rbfGamma * norm**2
  rbf_k = np.exp(gauss)
  return rbf_k
```

Figure 3.3: Code snippet of the implemented RBF kernel function.

Lines 11-17 represent a branch in the algorithm's logic. δ_t represents whether or not a label was given with the sample. In [fig ___], this is represented by the diamond containing “$\delta_t > 0$”. If no label was given, *line 12* would run, and only the manifold structure would be updated. If a label *was* given, then *lines 14-16* would run, and the result of a loss function would be combined with the update to the manifold structure.

In line *12*, we see that we are computing α using α^t, whereas in the MOMR version we calculate α^t using $\alpha^{\tilde{}t}$. We made the decision to treat this as a typo, and treated the α and α^t as if they were α^t and $\alpha^{\tilde{}t}$, respectively. Because we have already calculated *L* and *K* by this point, and because numpy provides an identity matrix function, the rest of the computation is pretty straightforward.

In *line 14*, we get *J* by taking the last column of the gram matrix *K*. This was easy to accomplish using Python's built-in list comprehensions: *K[:,-1]*. *Line 15* is where we compute the approximation of γ. *Figure 3.2* shows equation 17 from the paper. It's

similar to *line 12*, with the main exception being that we are multiplying the transpose of *J* and the label with $(I-\lambda_2 LK)\alpha^t$. One thing to note from this equation is that it is calculating for the next time-step, whereas in the algorithm, we are trying to obtain α at the current time-step. This discrepancy between the equations and the algorithm caused some confusion. Essentially, they both accomplish the same thing, but the equations operate at time-steps *t* and *t+1*, whereas the algorithm operates at time-steps *t-1* and *t*. This means that the α^t in equation 17 is actually α^{t-1} in the algorithm.

Line 16 is where we compute α for this branch. We already have γ^t, so we compute $(I-\lambda_2 LK)\alpha^t$. The rest of the math here is straightforward, and thanks to numpy's excellent support of matrix operations, the code looks clean and comprehensible. There is another important observation to make here for *lines 15-16*: they both use α^{t-1}. Remembering line 12, it uses α^t. Which means, as the authors have written it, $\alpha^{\sim t}$ is never used. This is what led us to think it was a typo, since it would make no sense to keep *line 8* if it was never needed. Thus, we decided that we would use $\alpha^{\sim t}$ in both of these cases, as well.

Lines 18-19 are where everything culminates. This is the function that will take in an input and calculate it's class. It does this by summing the result of what is, essentially, an array of weighted similarity values between the input sample and those in the buffer. By taking *sign(f(x))* we get the binary classification of *x* at the current time-step. Additionally, this means that the learning function and prediction function are not intertwined; we can predict without learning, and learn without predicting. This allows for

the algorithm to be run in an offline manner, which was perfect for our experiment. Again, numpy's math functions made this very easy to implement.

3.4 Verifying the Correctness of the Implemented Algorithm

Testing the algorithm comprised many challenges in and of itself. Unit tests were essential to verifying the transformations of the data, but they were lacking in one critical area: they couldn't verify if the algorithm was learning properly, due to a lack of any oracle. The only way to test this was to test the algorithm as a whole. This is a known issue with testing machine learning software [29]. Because of this, even small changes required retraining and retesting the algorithm, and caused development slow to a crawl. This was made even worse by the fact that some combinations of hyperparameters do not converge on a solution. So, in order to determine if there is a fault in the implementation, we would need to perform hyperparameter selection. However, the hyperparameter selection, itself, relied upon the ability of the algorithm to learn, which was difficult to verify because we didn't know the optimal hyperparameters for the dataset. This circular dependency was resolved by using the accuracy scores of the hyperparameter selection process to see if, even for the "optimal" hyperparameters, learning was taking place.

Hyperparameter selection slowed testing significantly. In the FMOMR paper, the authors suggest 7 different options for the two sigma values, and 8 different options for the two lambda values. This results in 1008 possible combinations for the hyperparameters for a single Buffer-U value. Using five-fold cross validation on 500

samples, like the authors did, we train and test the learner 5040 times before we obtain the optimal hyperparameters for the dataset.

Another area of uncertainty regarded the precision of the data types used for calculation, and how much they affected hyperparameter selection. It was uncertain whether a set of hyperparameters did not converge, or whether the precision of the data type was causing a preemptive overflow or underflow. This implementation used a 128-bit float, however other options were considered. Using a custom data type to represent the matrix values was one option, but the time to execute a test was already at several hours, even for simple data sets. The overhead of using a more complex representation of the data would have caused development to grind to a halt, for what supposed to be a simple proof of concept. A symbolic representation may have been another option, but there was no guarantee that it would resolve the issue, and the effort required to accomplish this would not be insignificant.

Python's built in unit testing framework was used to help streamline the testing of individual methods. One of the biggest issues we faced was making sure that the data was in the correct shape throughout the transformations on it, and this is where the unit tests were the most useful. We were able to test the more simple mathematical functions that we implemented, but they were useless for validating whether learning was taking place over several iterations.

The algorithm was initially tested on a synthesized dataset whose samples were low-dimensional and easily-separable, and then scaled up to more complex, less separable data. Eventually, it was tested against the MNIST dataset. In order to do this,

we needed create a pre-processing method in order reshape the data such that our program could operate on it. Despite being able to learn on simplistic data, as the complexity of the data increased, the accuracy of the algorithm quickly dropped. The algorithm, as implemented, was not suitable for the experiment.

3.5 Finding Another Suitable Algorithm

This was a huge setback, considering the time and effort expended. An alternative needed to be found, so we, again, sought out an already-implemented algorithm we could use. We only found one, and it was released while we was working on the FMOMR implementation. That algorithm was Incremental Label Propagation (ILP). We were quickly able to verify for ourselves that the algorithm functioned as described, and would be suitable for the experiment. Which only left a single challenge to overcome.

3.6 Integrating the Two Sets of Code

We needed to create an interface for ILP, so that it would be easily within the adapted cleverhans code. The next chapter details the architecture of this, along with more details of the algorithm itself. However, the difficulties of creating that interface were less significant than it's integration into Cleverhans.

The Cleverhans example being used had one major caveat: it did not operate in a procedural style; or at least, the operations for the learning process were written in a declarative style. The example utilized TensorFlow's graph and session workflow, meaning that a graph of operations upon the data is built, with the data being represented symbolically in the form of "placeholders" [30]. The real data is fed into this

graph at some point in the future, and it is then that the data is operated upon. It made integrating ILP, which *did* operate procedurally, an uncertain task. We were able to resolve this difference, but only after digging much more deeply into the architecture of TensorFlow than we had intended. We used a function that wraps an arbitrary python function, so long as it conforms to the specified interface required. With the final barrier overcome, the hypothesis could now be tested.

CHAPTER 4: EXPERIMENT

In this chapter, we discuss our experiment. We start with a breakdown of the problem, and then go on to discuss how we utilized the Cleverhans library and integrated the ILP library. Lastly, we cover the experiment setup and observe the results.

4.1 Problem Statement

The goal of this experiment was to examine the effectiveness of black-box attacks on a dynamic model. And by 'dynamic model', we mean a model changes through use. Thus, we form the following hypothesis: "If a target model learns from the samples which are being used to train the substitute model, then the adversarial examples crafted against the substitute will be less transferable to the target model."

While the adversary is training the substitute model, they must send samples to the target to be classified. Regarding the label of those samples, there are two possibilities: either the sample is labeled or unlabeled. But, when the adversary is attacking, they send adversarial examples to the target to be misclassified. Regarding the label of those samples, there are *three* different possibilities to consider.

1. The target learns from an unlabeled adversarial example, resulting in *adversarial drift* [31].
2. The target learns the true label of an adversarial example, resulting in *adversarial training* [32].

3. The target learns from an adversarial example with a false label, also resulting in *adversarial drift*.

Adversarial training has already been shown to be vulnerable to black-box attacks in an offline setting. Adversarial training in an online setting would likely be an effective defense, however this scenario is unrealistic, since the adversary would not want the target to know the true label. For this reason, we don't consider it an option for this experiment. The leaves the two cases which result in adversarial drift as viable options. Based on the fact that it was the simpler test case, the first case (no label given) was chosen for this experiment. The remaining option, where both the perturbation and a false label are learned, is left to future research.

So, we needed a classifier that could learn from unlabeled samples. This meant either an unsupervised or semi-supervised learner would be required. No research was found regarding existence of unsupervised classifiers, which meant we would need either an online or incremental semi-supervised learner.

4.2 Usage of Cleverhans

Cleverhans is an adversarial machine learning library used for bench-marking both the generating of adversarial examples and the defenses against them [20]. It uses the TensorFlow framework due to it's efficient processing of graph computations. The reasons for choosing this library were simple; this is the same library used by Papernot et. al. in their paper which introduced the concepts of transferability, and it provides an implemented example of their black-box experiment [6].

In it, they provide all of the code needed train a substitute, craft adversarial examples, and evaluate those examples on the target model. They also provide an example model as the target of the black-box attack: a basic convolutional neural network. As mentioned in the previous chapter, the authors make use of TensorFlow's graph and session workflow, which by default uses lazy evaluation.

They do this by using placeholder objects in place of the actual data, meaning that the graph of operations on the data is built symbolically. For example, this is a line from the function "prep_bbox":

```
predictions = model.get_logits(x)
```

It appears to be a call on the model to get some set of predictions, given a sample set `x`. However, `x` is a placeholder variable. This means that the operation of getting the predictions from the model is added to the computation graph for `x`, and is only computed once data is fed into the graph using `sess.run()`.

Because the ILP algorithm wasn't implemented using TensorFlow operations, we needed a way to integrate it into the computation graph. The solution was a wrapper function from TensorFlow: `tf.py_function`. Very few places in the original example were modified, since the goal was for our experiment to be as close to the original example as possible. We did need to do additional formatting of the training data as compared to the original; we needed to flatten the samples and argmax the array of labels. Other than that, we just used the wrapper function to call the ILP Learner as if it were the original black-box model from the example.

4.3 Integration Architecture

The goal was to create an interface for the different types of learning functions we would need for the experiment, namely offline and online learning. The implementation of the ILP algorithm is set up to easily configure and run experiments on the algorithm. However, behind the scenes is a very involved configuration parsing process which is used to initialize all of the major components needed for the algorithm to function. This complicated the task of exposing the needed functionality, since the configuration and learning processes were so intertwined.

Our solution to this was to create a “configurator” class to handle the configuration parsing, as well as the initial burn-in and training, and provide access to the now-initialized model. The configurator needed to be initialized by our ILP Learner object, and so it was passed three parameters: the name of the configuration file, the name of the dataset, and the size of the datastore. However, the model was still not ready for use. Once initialized, we needed to pass the configurator a set of samples to perform the initial training. The code for this step was adapted from the original experiment “base.py”, with much of the logging taken out, as well as the performance testing code which occurred every 1000 samples. Now that the model was initialized as well, the ILP Learner was able to call the function `getLearner()` on the configurator to retrieve the model.

With that access we could now create a class which could act as an interface for the learner. For the ILP Learner itself, there were three functions which we needed to implement in order to would allow the Cleverhans experiment to interface with the ILP

model. These functions were get_predictions_offline(x), get_predictions_online(x), and get_predictions_batch(x). The input for the three functions was the same: a set of samples in the form of a Tensor object. The output was also the same for all three: a Tensor object containing the predictions for the input samples. For the offline function, it computed the label predictions for those samples without learning from them. For the online function, it iterated through the set of input samples, and for each samples it would: predict it's label, append the label to an array, and then learn from the sample. Once the iteration finished. And for the batch function, the predictions for the labels were made all at once and then the samples were iterated and learned one by one. In each case, the input has to be parsed into a numpy ndarray object before it was operated upon, and converted back into a Tensor object before it was returned. TensorFlow contained built-in operations which made these conversions painless.

A diagram of this architecture is shown in *Figure 4.1*. With an interface for the ILP algorithm and a wrapper function for the calls to that interface, the two libraries could be integrated, and the experiment could commence.

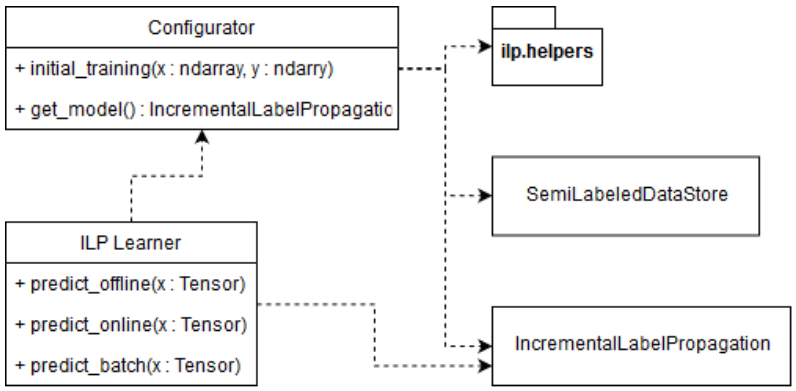

Figure 4.1: Design used to expose needed learning functionality

4.4 Experiment Setup

Before the experiment proper, we verified to our satisfaction that the ILP algorithm works as advertised. We tested it against the MNIST dataset – both ILP and Cleverhans have interfaces for using the dataset, and both were tested. We tested the algorithm using different values for the tuning variables θ, k_l , k_u, and ratio_labeled, even trying to purposely degrade it's ability to learn. Despite all this, the algorithm quickly recovered from this tampering after just a few thousand samples, with it's performance falling in line with that of a more optimal configuration. So we focused on the parameter which had the most impact: the number of initial training samples. For each run of the experiment, three different accuracies were calculated:

- the black-box model on real samples
- the substitute model on real samples
- the black-box model on adversarial examples

It should be noted that for all experiment runs which used the ILP algorithm, the accuracy of the substitute model on real samples remained constant at 76.365%. For this reason, it is excluded from the results.

The experiment comprised two testing variables: learning configuration and the size of the training dataset for the black-box model. There were two main configurations that we were interested in comparing: one that's purely offline, and one where the black-box model learns from the adversarial examples as they are given. We also tested a third configuration, where the black-box model learns from the real test samples as well. These three setups are labeled as "offline/offline", "offline/online", and "online/online",

respectively. Three values were chosen for the training dataset sample size (N_{train}): were 4k, 20k, and 40k.

4.5 Results

The test results for the ILP algorithm can be seen in *Tables 4.1 - 4.2*, with the difference between them shown in *Table 4.4* (Δ_{acc}). The results for the original algorithm are shown in *Table 4.3*, with the difference between those accuracies shown in *Table 4.4* as well. There are several notable observations here. Looking at *Table 4.1*, we see that, as expected, the more samples the model learns from, the more accurate it is. But it is interesting that there were no significant differences in the accuracy between the offline and online tests against real samples.

Table 4.2 is where we find the results which would possibly confirm our hypothesis: whether or not learning while the adversary trains the substitute influences the efficacy of the attack. If simply learning were enough, then it would be evident in the "Offline/Offline" and "Offline/Online" cases. However, there is no significant difference when comparing the two learning configurations. This means that it is not necessarily the case that our hypothesis is correct. Regarding this, future research opportunities are discussed in the next chapter.

N_{train}	Offline/ Offline	Offline/ Online	Online/ Online
4,000	83.563%	83.563%	83.563%
20,000	91.137%	91.137%	91.137%
40,000	93.503%	93.503%	93.503%

Table 4.1: Accuracy of the ILP algorithm on real samples.

N_{train}	Offline/ Offline	Offline/ Online	Online/ Online
4,000	65.555%	65.411%	65.411%
20,000	78.873%	78.741%	78.781%
40,000	83.919%	83.827%	83.827%

Table 4.2: Accuracy of the ILP algorithm on adversarial examples.

The last notable observation taken from these results is found in *Table 4.4*, which shows the difference between the accuracies in *Tables 4.1 – 4.2* (after taking the row-wise average) and the difference between the accuracies of the original example. The incongruity is immediately apparent; as the training set grows, the ILP algorithm becomes increasingly resilient against the adversarial examples. The convolutional neural net used in the original example, however, shows very little improvement in this regard.

N_{train}	bbox vs real	bbox vs adv
4,000	97.644%	73.746%
20,000	98.923%	75.908%
40,000	99.269%	76.680%

Table 4.3: Accuracy measurements for the original example. Left: accuracy of the black-box learner on unaltered samples. Right: accuracy of the black-box learner on adversarial examples.

N_{train}	ILP Δ_{acc}	Orig. Δ_{acc}
4,000	18.104	23.898
20,000	12.339	23.015
40,000	9.645	22.589

Table 4.4: Difference between the accuracy of the ILP algorithm (on left) and the original example (on right) between real samples and adversarial examples.

CHAPTER 5: RESEARCH OPPORTUNITIES & CONCLUSION

5.1 Research Opportunities with Dynamic Defenses

First, There is currently no consistent language for describing dynamic modeling. As noted earlier, there is ambiguity in this space [24]; terms like “incremental” are used interchangeably with “online”, as well as “lifelong”, “evolutionary”, and “stream”. A taxonomy of dynamic modeling (or dynamic-adversarial approaches) would go a long way in clarifying the language. Second, the relationship between dynamic models and the transferability of adversarial examples is in need of a theoretical foundation. While Sethi et. al. provide some theoretical foundation on the properties of dynamic defenses, it is an area which is just now being explored [23].

And lastly, there are plenty of research opportunities regarding the evaluation of different dynamic models. We observed that one adjacent area of research to this is dynamic fuzzy data analysis [33]–[36]. It would also be worthwhile to examine transferability in the context of non-stationary and imbalanced data streams.

5.2 Sample Noise + Label Noise

In chapter 4, one of the options we presented as a test scenario was where the target learns an adversarially perturbed sample with a false label. Examining the properties that emerge when testing both sample noise and label noise together may yield insight into the effects of different interactions of noise on a dynamic model. Some questions we pose:

- Are certain models more vulnerable to one over the other?
- What is the relationship between sample and label noise regarding their effect on the target's decision boundary?
- To what degree does the error specificy of the attack influence the effectiveness of the noise?
- How would the decision boundary be affected when sample and label noise target different classes?
- Would regularizing against sample noise impact the effectiveness of the label noise, and vice versa?

5.3 Conclusion

The transferability of adversarial examples remains an intractable problem. Approaches up till now have focused on the offline case, without consideration for the dynamic nature of learning in an adversarial environment. This thesis echos the sentiment of other recent publications: that a static model is not sufficient defend against adversarial attacks. We show that the testing of transferable adversarial examples against dynamic models is possible, and that there are many avenues of research available. Our experiment showed that it is not necessarily the case that learning during an adversarial attack is sufficient to impede that attack. And lastly, we assert that unification of terminology is necessary for continued research in this space.

REFERENCES

[1] P. Langley, *Elements of Machine Learning*. Morgan Kaufmann, 1996.

[2] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge: Cambridge University Press, 2014.

[3] O. Chapelle, B. Schölkopf, and A. Zien, Eds., *Semi-supervised learning*. Cambridge, Mass: MIT Press, 2006.

[4] L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar, "Adversarial Machine Learning," in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, New York, NY, USA, 2011, pp. 43–58.

[5] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples," *arXiv:1605.07277 [cs]*, May 2016.

[6] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical Black-Box Attacks against Machine Learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*, Abu Dhabi, United Arab Emirates, 2017, pp. 506–519.

[7] B. Biggio, I. Corona, G. Fumera, G. Giacinto, and F. Roli, "Bagging Classifiers for Fighting Poisoning Attacks in Adversarial Classification Tasks," in *Multiple Classifier Systems*, 2011, pp. 350–359.

[8] P. J. Villacorta and D. A. Pelta, "Exploiting Adversarial Uncertainty in Robotic Patrolling: A Simulation-Based Analysis," in *Advances in Computational Intelligence*, 2012, pp. 529–538.

[9] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu, and V. C. M. Leung, “A Survey on Security Threats and Defensive Techniques of Machine Learning: A Data Driven View,” *IEEE Access*, vol. 6, pp. 12103–12117, 2018.

[10] B. Biggio and F. Roli, “Wild patterns: Ten years after the rise of adversarial machine learning,” *Pattern Recognition*, vol. 84, pp. 317–331, Dec. 2018.

[11] T. S. Sethi and M. Kantardzic, “Handling adversarial concept drift in streaming data,” *Expert Systems with Applications*, vol. 97, pp. 18–40, May 2018.

[12] S. Thomas and N. Tabrizi, “Adversarial Machine Learning: A Literature Review,” in *Machine Learning and Data Mining in Pattern Recognition*, 2018, pp. 324–334.

[13] A. Kumar and S. Mehta, “A Survey on Resilient Machine Learning,” *arXiv:1707.03184 [cs]*, Jul. 2017.

[14] C. Szegedy *et al.*, “Intriguing properties of neural networks,” *arXiv:1312.6199 [cs]*, Dec. 2013.

[15] Z. Charles, H. Rosenberg, and D. Papailiopoulos, “A Geometric Perspective on the Transferability of Adversarial Directions,” *arXiv:1811.03531 [cs, stat]*, Nov. 2018.

[16] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, “Adversarial Attacks and Defences: A Survey,” *arXiv:1810.00069 [cs, stat]*, Sep. 2018.

[17] A. Athalye, N. Carlini, and D. Wagner, “Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples,” *arXiv:1802.00420 [cs]*, Feb. 2018.

[18] N. Carlini and D. Wagner, “Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods,” in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security - AISec ’17*, Dallas, Texas, USA, 2017, pp. 3–14.

[19] N. Carlini *et al.*, “On Evaluating Adversarial Robustness,” *arXiv:1902.06705 [cs, stat]*, Feb. 2019.

[20] N. Papernot *et al.*, “Technical Report on the CleverHans v2.1.0 Adversarial Examples Library,” *arXiv:1610.00768 [cs, stat]*, Oct. 2016.

[21] I. Goodfellow, “A Research Agenda: Dynamic Models to Defend Against Correlated Attacks,” *arXiv:1903.06293 [cs, stat]*, Mar. 2019.

[22] T. S. Sethi and M. Kantardzic, “When Good Machine Learning Leads to Bad Security: Big Data (Ubiquity Symposium),” *Ubiquity*, vol. 2018, no. May, pp. 1:1–1:14, May 2018.

[23] T. S. Sethi, M. Kantardzic, L. Lyua, and J. Chen, “A Dynamic-Adversarial Mining Approach to the Security of Machine Learning,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 3, p. e1245, May 2018.

[24] V. Losing, B. Hammer, and H. Wersing, “Incremental on-line learning: A review and comparison of state of the art algorithms,” *Neurocomputing*, vol. 275, pp. 1261–1274, Jan. 2018.

[25] S. Ding, X. Xi, Z. Liu, H. Qiao, and B. Zhang, “A Novel Manifold Regularized Online Semi-supervised Learning Model,” *Cognitive Computation*, vol. 10, no. 1, pp. 49–61, Feb. 2018.

[26] I. Chiotellis, F. Zimmermann, D. Cremers, and R. Triebel, “Incremental Semi-Supervised Learning from Streams for Object Classification,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid, 2018, pp. 5743–5749.

[27] “scikit-learn: machine learning in Python — scikit-learn 0.20.3 documentation.” [Online]. Available: https://scikit-learn.org/stable/index.html. [Accessed: 16-Apr-2019].

[28] “NumPy — NumPy.” [Online]. Available: https://www.numpy.org/. [Accessed: 16-Apr-2019].

[29] S. Huang, E.-H. Liu, Z.-W. Hui, S.-Q. Tang, and and S.-J. Zhang, “Challenges of Testing Machine Learning Applications,” *International Journal of Performability Engineering*, vol. 14, no. 6, p. 1275, Jun. 2018.

[30] “Graphs and Sessions | TensorFlow Core,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/guide/graphs. [Accessed: 16-Apr-2019].

[31] A. Kantchelian *et al.*, “Approaches to Adversarial Drift,” in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, New York, NY, USA, 2013, pp. 99–110.

[32] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and Harnessing Adversarial Examples,” *arXiv:1412.6572 [cs, stat]*, Dec. 2014.

[33] F. Iglesias, J. Milosevic, and T. Zseby, “Fuzzy classification boundaries against adversarial network attacks,” *Fuzzy Sets and Systems*, Nov. 2018.

[34] A. Bouchachia, E. Lughofer, and D. Sanchez, “Editorial of the special issue: Online fuzzy machine learning and data mining,” *Information Sciences*, vol. 220, pp. 1–4, Jan. 2013.

[35] H.-J. Zimmermann, “Dynamic fuzzy data analysis and uncertainty modeling in engineering,” Aug. 2000.

[36] A. Joentgen, L. Mikenina, R. Weber, and H.-J. Zimmermann, “Dynamic fuzzy data analysis based on similarity between functions,” *Fuzzy Sets and Systems*, vol. 105, no. 1, pp. 81–90, Jul. 1999.