Adversarial Distributed Machine Learning

Ran Hao, Liang Tong, Xiaodong Yang

Abstract— **In this project, a distributed structure for the adversarial machine learning is studied. Currently, the surging volume of data the increasing complexity of target models have been putting a great pressure on the individual machine. Therefore, the mechanism of distributed systems become prerequisite to handle these challenges. Distributed machine learning has been widely used in big data applications. However, there is no prior work on the robustness of distributed machine learning systems in presence of a sophisticated adversary, where it changes behavior and produce adversarial examples to fool the machine learning systems. In this paper, we first implement linear regression on a distributed system based on ParameterServer. Particularly, we apply distributed stochastic gradient descent to train the linear regressor. We then use an attacker's model to generate instances that are incorrectly predicted by the machine learning system. We finally conduct our experiments on real-world data. The results show that although distributed systems can be used for training on large scaled data, they are still vulnerable to adversarial attacks.**

I. Introduction

Machine learning(ML) is a research field that applies statistical methods to train computer systems to act without being explicitly programmed. There are mainly three categories in the ML application according to the desired output which are classification, regression and clustering. In classification, the system aims to divides inputs into two or more than two classes based one the features extracted from the training data. In regression, the system aims to generate continuous outputs instead of discrete ones. While in clustering, the system will classify inputs into different unknown groups. ML techniques have been successfully applied to a lot of fields such as self-driving cars, speech recognition and others.

The problems rise from the fact the training data and test data are originally collected from stationary environment with the similar distribution. Some intelligent and adaptive adversaries may violate this working hypothesis by manipulating the input date and taking advantage of specific vulnerabilities in the learning algorithm. The strategy for improving the security properties of ML includes the identification of potential vulnerabilities of learning algorithm, evaluation of identified threats' impact on the targeted system, and the proposal of countermeasure against the identified attacks to improve the system's security.

Tremendous research have been done in the these fields. However, another challenge is rising due to the increasing amount of data and the complexity of target model. To solve large-scale machine learning problems, the mechanism of distributed systems becomes a prerequisite. For example, the social media such as Instagram and Facebook are known to handle 10s of billions images with taking up TBs of storage. According to the talk [1], the training data in realistic application can range between 1 TB and 1 PB, which is sufficient to generate one complex model with up to 10^{12} parameters. The traditional ML methods based on single machine cannot solve these problems rapidly and accurately due to the storage and time limits and their sequential learning algorithm. To develop a robust system for adversarial distributed machine learning, three issues should be taken into consideration:

1) A large network bandwidth is required to migrate or replicate the globally shared parameters.
2) The sequential learning algorithm should be modified, since the high machine latency and cost of synchronization is likely to reduce systems' efficiency.
3) Methods for fault tolerance is required to keep systems' consistency. Being executed in a distributed system, tasks may be lost due to the unreliable worker and server nodes.

In this project, a method for linear regression using stochastic gradient descend and $Disbelief$ distributed ML structure is studied.

II. Adversarial Machine Learning

This section introduced adversarial machine learning. Modeling adversarial evasion is and building robust machine learning systems are two most important open problems in the research of adversarial machine learning. On the one hand, evasion models are abstractions of adversaries, which provide a natural way for theoretical analysis of the robustness of classifiers. On the other hand, these models can be leveraged for designing evasion-robust classifiers if they faithfully represent actual potential attack to the classifiers. Several previous studies on modeling adversarial evasion and evasion-robust classification include: minimum-distance evasion of linear classifiers [2], adversarial-aware classifications [3], game-theoretical modeling [4], and modeling with randomized operational decisions [5]. Researchers in cyber-security also propose systematic framework to evade machine learning systems, such as [6] and [7] These prior arts are summarized below.

A. *Minimum-distance evasion of linear classifiers*

Lowd and Meek [2] are the first to propose an adversarial evasion model for linear classifiers. They investigate the difficulty of evading linear classifiers by studying the corresponding complexity problem. Particularly, they define *ACRE*, adversarial classifier reverse engineering, to formalize the problem of constructing adversarial instances which fool the classifier by queries. From my understanding, the main contributions of [2] are as follows:

1) The authors proposed the first theoretical framework to study the complexity of evading linear classifiers. Compared to previous work, the authors not only presented their attack model but analyzed the complexity of evading classifiers by applying the model.
2) The authors investigated classifiers with both continuous and binary features, and proposed efficient algorithms to obtain *near-optimal attack*. That is, finding a low-cost adversarial instance after some probes of the target classifier.
3) The authors validated their theoretical framework with a real-world application, *spam filtering*.

To accomplish their goals, the authors of [2] made several assumptions. First and foremost, the adversary has only *black-box* access to the target classifier. That is, the training dataset of the target classifier is not available to the adversary. Instead, the adversary can only modify its *feature space* instances by repeatedly sending *membership queries* to the classifier. The second assumption is that the adversary can modify an instance $\mathbf{x}$ with an *adversarial cost function* $a(\mathbf{x})$ that transforms $\mathbf{x}$ to $\mathbf{x}^{'}$ which can fool the classifier. The third assumption is that the adversary is provided with a malicious instance $\mathbf{x}^{+}$ and a benign instance $\mathbf{x}^{-}$, which are further employed to craft adversarial instances.

Based on the assumptions above, the authors first define the *adversarial classifier reverse engineering (ACRE) learning problem* for a given classifier c and adversarial cost function a: To find an instance $\mathbf{x}$ such that it is classified as benign and has a minimum cost of all instances classified as benign. A set of classifiers $\mathscr{C}$ is *ACRE learnable* under a set of cost functions $\mathscr{A}$ if there exists an algorithm such that for any classifier in $\mathscr{C}$ and cost function in $\mathscr{A}$, finding the ACRE solution only uses polynomial membership queries. As finding the solution of ACRE learning problem could be intractable, the authors also define *ACRE k-learnerble* which modifies the concept of ACRE learnable: Instead of finding an ACRE solution, we can find a solution which is less than k times of the minimum cost function of the ACRE solution. If the algorithm of finding the corresponding solution only uses polynomial queries, then the set of classifiers is called *ACRE k-learnerble* under the set of cost functions. Particularly, the authors use *linear cost function* as follows:

$$a(\mathbf{x}) = \sum_{i} a_i |x_i - x_i^a| \tag{1}$$

where x_i and x_i^a is the ith feature of the target instance $\mathbf{x}$ and base instance $\mathbf{x}^a$, and a_i is the weight of the absolute difference. When every $a_i = 1$, the cost function is called *uniform linear cost functions*.

Using the models above, the most two important theoretical results of [2] are as follows: First, under linear cost functions of an adversary, linear classifiers with continuous features are ACRE $(1+\varepsilon)$-learnable. To evade such classifiers for an adversary with linear cost functions, the authors first approximate feature weights and then use these weights to identify low-cost instances. Second, under uniform linear cost functions, linear classifiers with binary features are ACRE 2-learnable. To evade such classifiers for an adversary with uniform linear cost functions, even learning the sign of the weights is *NP-hard*. Thus, the authors start with a malicious instance and iteratively modifies it to find instances fool the classifier. The two findings and the corresponding algorithms above reveal the difficulty of evading particular classifiers under particular cost functions of the adversary, which can be further applied to evaluate the robustness of a classifier.

Although Lowd and Meek [2] inspired intensive follow-up research, there are still some limitations. First, their assumptions underestimate the capability of adversaries. Instead of having only black-box to the target classifier, an adversary can build a *surrogate classifier* and somehow ”approximate” the target classifier. Such setting can make the classifier more vulnerable to adversarial evasions. Second, near-optimal evasion (with minimum-distance cost) is an ideal assumption considering real-world applications. Real-world evasions do not require a solution with near-optimal cost. Instead, a solution which can evade the target classifiers with reasonable modification cost can be accepted. In spite of these limitations, [2] still provide a worst-case complexity of evading a linear classifier.

B. Adversary-aware classification

Being different from [2] which studies the complexity of evading a classifier with an adversarial evasion model, Dalvi et all. [3] is the first to propose evasion-robust classification by utilizing evasion models. Particularly, the authors present a formal framework and algorithms for designing evasion-robust classifiers. They model the problem as a game between the classifier and adversary, and produce a classifier which is aware of the optimal strategy of the adversary under a certain evasion model. The authors make several assumptions as follows:

1) Both the classifier and adversary has complete information of each other, such as the utility function and training dataset.
2) The classifier is aware of the adversary, knowing that the adversary takes optimal strategy to evade the classifier.
3) The adversary assumes that the classifier is unaware of its presence.

Based on the assumptions above, Dalvi et al. [3] make several contributions to advance prior studies, listed as follows.

First, the authors propose a game theoretical model as a framework to design evasion-robust naive Bayes classifiers. The game contains two players: the classifier and adversary. The classifier's goal is to maximize its expected utility with considerations of the adversary's optimal strategy to evade it. The expected utility of the classifier is a subtraction of two parts: its utility of correct classification minuses the cost of classification. The adversary's goal is to find a strategy to change the features of an instance which maximizes its utility function, which is the subtraction of its utility of instances being classified as benign and its modification cost. Unlike traditional static Nash games in which each player moves simultaneously and assumes optimal strategies of others, the game proposed in [3] is dynamic and contains three phases. In the first phase, the classifier is trained assuming the absence of adversary. In the second phase, the adversary chooses its optimal strategy to evade the classifier trained in the first phase. This problem is modeled as an integer linear optimization problem with constraints with the modification cost as the objective function, and being classified as benign as the constraint. In the third phase, the classifier chooses its optimal strategy by taking into account the adversaries optimal strategy in the second phase. That is, to compute the posterior probability of an instance while considering the adversary's strategy.

The second contribution of this paper is that the authors develop efficient algorithms to solve the game proposed. Particularly, they devise a pseudo-linear time solution to the constrained integer linear optimization problem of the adversary in the second phase of the game, based on dynamic programming. The authors also propose an approach to compute the posterior of a test instance by taking into account both tampered and untampered instances.

The experiment results in [3] show that by considering adversarial evasion model, the number of *false negatives* can be significantly reduced in the presence of an adversary. However, there are still several limitations of this work. First, the authors make a pessimistic assumption on the capability of the adversary, as the adversary can also make its optimal strategy by assuming that the classifier chooses an optimal strategy to improve its robustness to evasion. Second, the proposed game theoretical framework can only be applied to naive Bayes classifiers. It is still unclear how to design evasion-robust SVM or logistic regression classifiers by using adversarial evasion models. The limitations above are addressed in [4], which is summarized below.

C. *Modeling with games*

In [4], Bruckner et al. propose a Stackelberg game model to represent the interactions between a classifier and adversary, in which the classifier plays the role of the leader and the adversary as the follower. Both the classifier and the adversary have complete information of each other. Being different with [3], the authors make some assumptions as follows.

1) The players act *non-simultaneously*. Particularly, the classifier moves first by selecting its classification models $\mathbf{w}$. Then, the adversary moves by transforming correctly classified malicious instances D into adversarial instances $\tilde{D}$.
2) Both players are aware of each other. That is, the adversary plays a best response to the observed models $\mathbf{w}$ chosen by the classifier, and given this behavior, the classifiers chooses the optimal $\mathbf{w}$.

Based on the assumptions above, the authors of [4] make several contributions listed as follows.

First, they propose a formal Stackelberg game to model the interactions between the classifier and the adversary. The goal of the classifier is to minimize its cost functions, which is the sum of its loss functions on adversarial data produced by the adversary and a regularizer. The goal of the adversary is to minimize its cost functions, which is the sum of its loss functions on adversarial data and the modification cost of transforming D to $\tilde{D}$. Note that the loss function of the adversary is agnostic to the one of the classifier. That is, if an instance produced by the adversary is correctly classified, then the loss function of the classifier is negative, and the loss function of the adversary is positive, and vise versa. As both the loss function of the classifier and the adversary are intertwined with each other's strategy, the proposed Stackelberg game is modeled as a *bilevel optimization problem*-with upper-level as minimizing the classifier's cost function and lower-level as minimizing the adversary's cost function.

The second contribution of [4] is that it proposes an approximation algorithm to derive the solution of the Stackelberg game, which is a local optimum. Previous approaches that address the bilevel optimization reformulate the problem as an optimization problem with equilibrium constraint by replacing the lower-level optimization problem with its KKT conditions. However, as [4] claims, such methods do not necessarily converge to a local optimum. In [4], the authors reformulate the lower-level optimization problem into an unconstrained problem such that the local optima can be obtained by *sequential quadratic programming (SQP) methods*.

The third contribution is that the authors generalize their results to different loss functions, such as hinge loss and logistic loss. Hence, the Stackelberg game model can be generalized to different existing prediction models.

Compared to [3], [4] makes more realistic assumptions, and the proposed approach obtains better generalization. However, there are still limitations. Similar to [3], the game theoretical model of [4] requires a *redesign* a classifier to improve its robustness to adversarial evasions, which can incur heavy *operational cost* for existing large machine learning systems. Such limitation is addressed by [5] summarized below.

D. *Modeling with randomized operational decisions*

As discussed above, prior work on designing evasion-robust classifiers by utilizing evasion models fail to account for operational cost. They are also restricted to deterministic classification. In [5], Li and Vorobeychik address such limitations by proposing a conceptual framework which separates learning and randomized operational decisions accounting for an adversary.

A conceptual difference of [5] and other work is the use of randomized decision. Previous probabilistic classifiers use a probability $p(x)$ to present the probability of an instance x being produced by a malicious entity. If $p(x)$ is greater than a threshold θ, then this instance is classified as malicious. Otherwise, it is classified as benign. In contrast, [5] introduces a function $q(x, p(\cdot)) \in [0, 1]$ to represent a possibly ramdomized decisions. e.g. $q(x)$ can represent the probability of manually insvesitgating some malicious instances.

With the settings above, the interactions between a classifier and adversary is then modeled as a Stackelberg game, where the classifier acts as the leader and choose the optimal $q(\cdot)$, and the adversary acts as the follower to chooses the adversarial instance x by observing $q(\cdot)$. The classifier's goal is to minimize its utility function which accounts for the loss of false positives and false negatives. The goal of the adversary is to choose an adversarial instance $x^{'}$ by modifying x and minimize

$$\mu(x,x^{'};q) = V(x)Q(x,x^{'})(1-q(x^{'}))$$

where $V(x)$ is the value of attack, $Q(x,x^{'})$ is the cost of modification, and $(1-q(x^{'}))$ is the probability that $x^{'}$ fools the classifier.

The authors in [5] formulate the Stackelberg game above as a constrained linear programming problem. A challenge is that $q(x)$, the variable of the objective function must be defined over the entire feature space. To address the scalability issue, the authors represent $q(x)$ by a set of basis functions $\{\phi_j\}$ such that $q(x) = \alpha_j \phi_j(x)$. Then, the authors propose approximation algorithms to derive $q(\cdot)$ and the optimal strategy of the adversary, $x^{'}$, which is a provably near-optimal solution.

By separating learning ($p(x)$) and operational decision ($q(x)$), The framework proposed by [5] can make use of state-of-art machine learning classifiers, and improve their robustness by embedding randomization. Two common limitations of [2]–[5] are discussed in the next section.

E. EvadeML

EvadeML is an automated method to craft evasion instances of PDF malware in problem space. It starts with a malicious PDF which is correctly classified as malicious and aims to produce evasive variants which have the same malicious behavior but are classified as benign. It assumes that no internal information of the target classifier is available to the adversary, such as the set of features, the training dataset, and the classification algorithm. Rather, the adversary has black-box access to the target classifier, and it can repeatedly submit PDF files to get corresponding classification scores. Based on the scores, the adversary can adapt its strategy to craft evasive variants.

EvadeML employs genetic programming (GP) to search the space of possible PDF instances to find ones that evade the classifier while maintaining malicious features. The GP process is illustrated in Figure 1.

First, an initial population is produced by randomly manipulating a malicious seed. As the seed contains multiple PDF objects, each object is set to be a target and mutated with exogenously specified probability. The mutation is either a deletion, an insertion or a swap operation. A deletion operation deletes a target object from the seed malicious PDF file. As a result, the corresponding structural path is deleted. An insertion operation inserts an object from external benign PDF files (also provided exogenously) after the target object. EvadeML uses 3 most benignly scoring PDF files. A swap operation replaces the entry of the target object with that of another object in the external PDFs.

After the population is initialized, each variant is assessed by the Cuckoo sandbox [**?**] and the target classifier to evaluate its fitness. The sandbox is used to determine if a variant preserves malicious behavior. It opens and reads the variant PDF in a virtual machine and detects malicious behaviors such as API or network anomalies, by detecting malware signatures. The target classifier provides a classification score for each variant. If the score is above a threshold, then the variant is classified as malicious. Otherwise, it is classified as a benign PDF. If a variant is classified as benign but displays malicious behavior, or if GP reaches the maximum number of generations, then GP terminates with the variant achieving the best fitness score and the corresponding mutation trace is stored in a pool for future population initialization. Otherwise, a subset of the population is selected for the next generation based on their fitness evaluation. Afterward, the variants selected are randomly manipulated to generate the next generation of the population.

EvadeML was used to evade SL2013 in [6]. The reported results show that it can automatically find evasive variants for all 500 selected malicious test seeds. However, we found a small error in the implementation which caused the reported evasion results to be slightly inflated; we were able to reproduce an approximately 84% evasion rate, as reported below. Throughout, we use 400 malicious seeds as a part of evasion-based training and evaluate on the remaining 100 malicious seed PDFs used by EvadeML.

F. Mimicry

Mimicry [7] assumes that an attacker has full knowledge of the features employed by a target classifier. The mimicry attack then manipulates a malicious PDF file so that it mimics a particular selected benign PDF as much as possible. The implementation of Mimicry is simple and independent of any particular classification model. To improve evasion effectiveness, Mimicus chooses 30 different target benign PDF files for each attack file. It then produces one instance in feature space for each target-attack pair by merging the malicious features with the benign ones. The feature space instance is then transformed into a PDF file using a *content injection approach*. The resulting 30 files are evaluated by the target classifier, and only the PDF with the best evasion result is selected, which was submitted to WEPAWET [8] to verify malicious functionality.

III. Distributed Machine Learning

Distributed machine learning has the main focus on communication, synchronization and fault tolerance of the learning parameters transferring for large scale machine learning systems. The goal of distributed machine learning is similar with machine learning, which is optimizing the "object function" in general. However, in large scale machine learning systems, the scale of training data and learning parameters are usually too large to be calculated in real time at

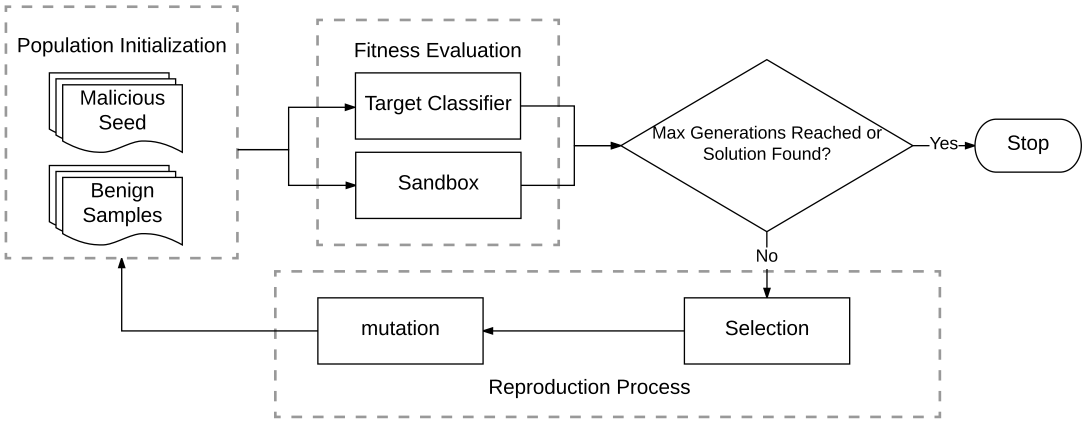


Fig. 1. Classifier evasion with genetic programming [6].

one machine. This problem becomes more noticeable when the parameters needs to be updated or refined with large iterations. However, implementing an efficient distributed structure for machine learning problem is not easy. Normally, the parameters and training data are globally shared among the worker nodes, which makes synchronization and fault tolerance very challenging to accomplish.

Spark [9] proposed an architecture based on directed acyclic graph (DAG), where the vertex denotes an Resilient Distributed Dataset (RDD) and the edge denotes an operation on RDD. There are two types of operations, transformations and actions. A transformation performs an operation on a RDD and produces a new RDD. An action triggers a job in Spark. A typical *Spark* job performs a couple of transformations on a sequence of RDDs and then applies an action to the latest RDD in the lineage of the whole computation. *Spark* [10] is proven to be effective when dealing with simple logistic regression, its performance dips for more involved machine learning tasks. However, *Spark* has limited scalability, especially when performing machine learning tasks with large volume of model parameters.

In [11], the parameter server is proposed and becoming one of the prevail architectures for distributed machine learning. As shown in Fig. 2, all parameter server nodes are grouped into a server group and several worker groups [1]. Each server node maintains a partition of the globally shared parameters or weights. Server nodes communicate with each other to replicate or to migrate parameters for reliability and scaling. A server manager node maintains a consistent view of the metadata of the servers, such as node liveness and the parameter partitions. Each worker group runs an application, e.g., stochastic gradient decent (SGD). A worker stores only a portion of the training data to compute the local gradients. Workers communicate only with the server nodes instead of other worker nodes, updating and retrieving the shared parameters. There is a scheduler node for each worker group, which controls the tasks to workers and monitors their liveness and computation. If workers are added or removed, it reschedules unfinished tasks.

Distbelief [12] is designed for solving large-scale noncon-

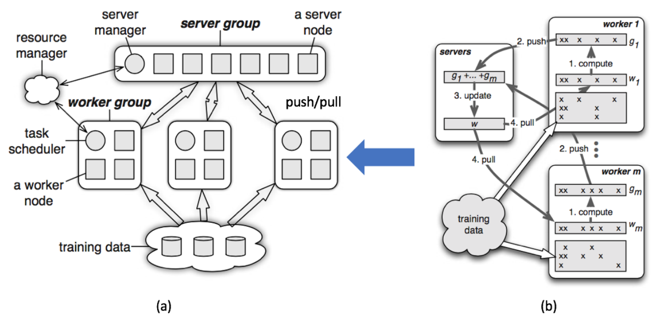


Fig. 2. The architecture of parameter server for computing global gradient of larger scale distributed machine learning systems.(b) gives an example of *distributed subgradient decent*. (a) shows the total structure of parameter server of having multiple worker groups and a server group.

vex optimization problems and works very well for training deep neural networks and SGD. *Distbelief* proposed a Downpour SGD(DSGD) based on parameter server using a model parallelism scheme (Fig. 3). Downpour SGD is a variant of asynchronous stochastic gradient descent where the model replicas run independently of each other and the parameter server shards also run independently of one another. Each *Distbelief* model is composed by a mini-batch of machines as shown in Fig. 3. Each machine needs to communicate with the subset of parameter server shards that hold the model parameters relevant to its partition. The *Distbelief* model replicas will process its own mini-batch of data to compute a parameter gradient at each iteration and sends the gradient to the parameter server. The parameter server will update the globally share parameters based on the received gradients and distribute the updated parameters to the model replicas. This approach is able to tolerate variance in the processing speed of different model replicas, and the failure of model replicas which may be taken off-line or restarted at random.

In this paper, we will design an architecture based on the principles of parameter server and *Distbelief* for computing the DSGD, and specifically, solving an adversarial machine learning problem using linear regression. ZeroMQ and Mininet are both employed as communication and sim-

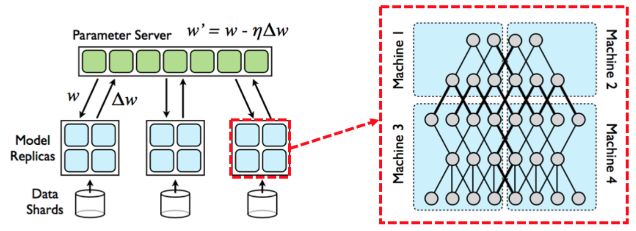


Fig. 3. The *Distbelief* distributed ML structure for a 5 layer deep neural network. The network is distributed across 4 machines.

ulation platform. The detailed description of the architecture is shown in the next section.

IV. Adversarial Distributed Machine Learning

This section introduces our final project on adversarial distributed machine learning. First we introduce the adversarial machine learning model which includes both a learner (linear regression) and an attacker. The learner is a linear regression system which aims to learn from training data. It uses distributed gradient descent algorithm for training. The attacker is an entity with market motivation, which manipulate adversarial example to fool the attacker. That is, the attacker aims to provide an example, which is predicted with a higher value by the learner than its actual value. We formulate the attacker as a convex optimization problem, and use regular gradient descent algorithm to obtain the best adversarial example. Note that the learner is a distributed machine learning system, but the attacker is running on a single machine, as it has only one data to manipulate. We use this concise model to evaluate the robustness of the distributed linear regression system.

A. The learner's model

At training time, a set of training data $(\mathbf{X},\mathbf{y})$ is drawn from an unknown distribution $\mathscr{D}$. $\mathbf{X}\in\mathbb{R}^{m\times d}$ is the training sample and $\mathbf{y}\in\mathbb{R}^{m\times 1}$ is a vector of values of each data in $\mathbf{X}$. We let $\mathbf{x}_j\in\mathbb{R}^{d\times 1}$ denote the jth instance in the training sample, associated with a corresponding value $y_j\in\mathbb{R}$ from $\mathbf{y}$. Hence, $\mathbf{X}=[\mathbf{x}_1,\ldots,\mathbf{x}_m]^\top$ and $\mathbf{y}=[y_1,y_2,\ldots,y_m]^\top$. On the other hand, test data can be generated either from $\mathscr{D}$, the same distribution as the training data, or from $\mathscr{D}'$, a modification of $\mathscr{D}$ generated by an attacker.

The action of the learner is to select a $d\times 1$ vector $\boldsymbol{\theta}$ as the parameter of the linear regression function $\hat{\mathbf{y}}=\mathbf{X}\boldsymbol{\theta}$, where $\hat{\mathbf{y}}$ is the predicted values for test data $\mathbf{X}$. The expected cost function of the learner at test time is then

$$c_l(\boldsymbol{\theta},\mathscr{D}')=\beta\mathbb{E}_{(\mathbf{X}',\mathbf{y})\sim\mathscr{D}'}[\ell(\mathbf{X}'\boldsymbol{\theta},\mathbf{y})]. \tag{2}$$

where $\ell(\hat{\mathbf{y}},\mathbf{y})=||\hat{\mathbf{y}}-\mathbf{y}||_2^2$. That is, the cost function of a learner is a combination of its expected cost from both the attacker. However, the learner has no knowledge about the attacker, it is only trained on the training data $\mathscr{D}$.

To minimize its cost function, the learner uses distributed stochastic gradient algorithm, where the tasks are assigned to a server node and n worker node. The algorithm is summarized as follows.

1) The server initializes its $\boldsymbol{\theta}=\mathbf{0}$.
2) The server uniformly distributes training data on the workers
3) The server sends $\boldsymbol{\theta}$ to all the workers.
4) Each of the worker i, randomly select an instance $\boldsymbol{x}_i$ and its true value y_i from its training data, then computes the gradient as follows and sends to server.

$$\boldsymbol{g}_i=(\boldsymbol{\theta}^\top\boldsymbol{x}_i-y_i)\boldsymbol{x}_i \tag{3}$$

5) The server receives gradient from all the workers and computer the average gradient $\tilde{\boldsymbol{g}}$.

$$\tilde{\boldsymbol{g}}=\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{g}_i. \tag{4}$$

6) The server upgrades $\boldsymbol{\theta}$ by

$$\boldsymbol{\theta}=\boldsymbol{\theta}-\beta_l\times\tilde{\boldsymbol{g}} \tag{5}$$

7) Return to 4) until $\boldsymbol{\theta}$ gets converged.

B. The attacker's model

One example $(\mathbf{x},y)$ generated according to $\mathscr{D}$ is maliciously modified by the attacker into another, $(\mathbf{x}',y)$, as follows. We assume that the attacker has an instance-specific target $z(\mathbf{x})$, and wishes that the prediction made by the learner on the modified instance, $\hat{y}=\boldsymbol{\theta}^\top\mathbf{x}'$, is close to this target. We measure this objective for the attacker by $\ell(\hat{\mathbf{y}}',\mathbf{z})=(\hat{\mathbf{y}}'-\mathbf{z})^2$ for a vector of predicted and target values $\hat{\mathbf{y}}'$ and z, respectively. In addition, the attacker incurs a cost of transforming an instance $\mathbf{x}$ into $\mathbf{x}'$, denoted by $R(\mathbf{x}',\mathbf{x})$.

After an adversarial example $(\mathbf{x}',y)$ is generated in this way by the attacker, it is used against the learner. This is natural in most real attacks: for example, spam templates are commonly generated to be used broadly, against many individuals and organizations, and, similarly, malware executables are often produced to be generally effective. The cost of the learner is then defined as follows:

$$c_a(\boldsymbol{\theta},\mathbf{x}')=\ell(\mathbf{x}'\boldsymbol{\theta},z)+\lambda R(\mathbf{x}',\mathbf{x}). \tag{6}$$

where the attacker's modification cost is measured by $R(\mathbf{x}',\mathbf{x})=||\mathbf{x}'-\mathbf{x}||_2^2$ which is the squared ℓ_2 norm.

To minimize $c_a(\boldsymbol{\theta},\mathbf{x}')$, which is a convex function of $\mathbf{x}'$, the attacker uses gradient descent algorithm. That is, it iteratively update $\mathbf{x}'$ by

$$\mathbf{x}'=\mathbf{x}'-\beta_a[(\boldsymbol{\theta}^\top\mathbf{x}'-z)\boldsymbol{\theta}+\lambda(\mathbf{x}-\mathbf{x}')] \tag{7}$$

Until $c_a(\boldsymbol{\theta},\mathbf{x}')$ is converged. Then, the resulting $\mathbf{x}'$ is used to fool the learner. Typically, z is larger than y. For example, a house could be predicted with a higher price.

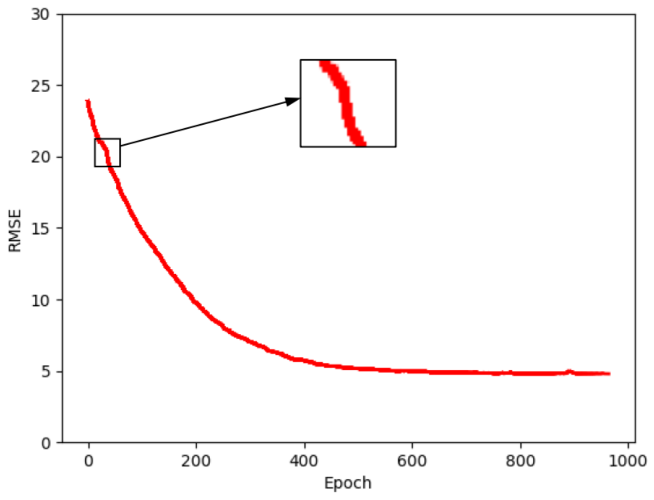


Fig. 4. The experiment result for defender of housing price prediction/ We let node 0 of the first batch died at iteration 58.

C. Software Architecture

In this paper, we design our adversarial distributed machine learning architecture using a variation of *Distbelief*. *Distbelief* is originally designed for large scale neural network, in this paper, we make some adaptations of *Distbelief* to solve an minimization problem using DSGD algorithm in distributed architecture. ZeroMQ [13] is used for message transferring and communication tool in our distributed architecture. As shown in Fig.2 and Fig.3, here we use three workers to compute the subgradient together. And each of the three workers forms a mini-batch, where a scheduler will monitor all the member workers. We use Zookeeper [14] the monitoring process and distributed lock services. The parameter server will also use a Zookeeper object to watch and communicate with all schedulers. In each mini-batch, the scheduler will notify the server if any of the workers in its batch is dead or still alive. Once the server node is notified a dead worker, the batch will stop working and the next alive batch will continue the unfinished work of computing the gradients. This guarantees the fault tolerance of the machine learning system.

V. Experiments

We tested our software architecture of adversarial distributed machine learning using a housing example for a defender and an attacker to predict and disturb the housing price. In this paper, we use three workers in one mini-batch and 2 batches for the fault tolerance. We pick 1000 epoch for the algorithm to converge. The defender's convergence result is shown in Fig. 4. We stopped the main batch at iteration 58, and the second batch continues the work from the first batch, which gives the distributed algorithm more fault tolerance.

The attacker's process to generate adversarial example is shown in Figure 5. As discussed in Section IV, the attacker minimizes a convex function by using gradient descent. Fig 5 reveals that after only 200 iterations, the attacker's loss function gets converged, which means that the distributed

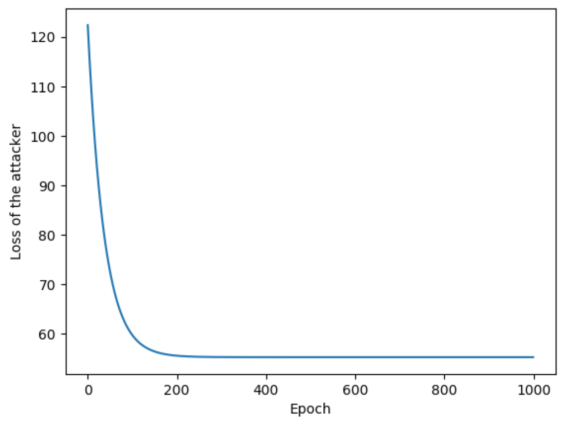
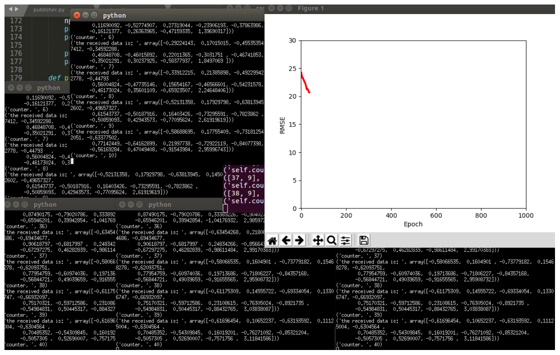


Fig. 5. The results of attacker's minimization results. The minimal is reached at around 300 iteration



```
        0.11690092, -0.52774907,  0.27319044, -0.23906193, -0.37863986,
       -0.16121377,  0.26363965, -0.47159335,  1.39690317]))
('counter, ', 6)
('the received data is: ', array([-0.29224143,  0.17015015, -0.45535354
7412, -0.34592288,
        0.46848708, -0.46015892,  0.22011365, -0.3031751 , -0.46741853,
       -0.35021291,  0.30237925, -0.58377937,  1.8497069 ]))
('counter, ', 7)
('the received data is: ', array([-0.33912215,  0.21385898, -0.49229942
2778, -0.44793   ,
        0.56004824, -0.47735146,  0.15654167, -0.46566601, -0.54231578,
       -0.46173024,  0.35601109, -0.65923507,  2.24648406]))
('counter, ', 8)
('the received data is: ', array([-0.52131358,  0.17929798, -0.63813945
2602, -0.49657327,
        0.61543737, -0.50187916,  0.16403426, -0.73295591, -0.7823862 ,
       -0.50859093,  0.42943573, -0.77095624,  2.61919619]))
('counter, ', 9)
('the received data is: ', array([-0.58688695,  0.17755409, -0.73181254
2051, -0.63377502,
        0.77142449, -0.64162899,  0.21997738, -0.72922119, -0.84077398,
       -0.56169284,  0.47049408, -0.91543984,  2.95996743]))
('counter, ', 10)
```

```
('self.cou
([37, 9],
('self.cou
([38, 9],
('self.cou
```

```
        0.87490175, -0.79020786,  0.33389226, -0.90402
       -0.65946201,  0.39942854, -1.04176932,  2.985972
('counter, ', 36)
('the received data is: ', array([-0.63454268,  0.21808
4686, -0.69434677,
        0.90618797, -0.6817997 ,  0.24834266, -0.856643
       -0.67297275,  0.46282839, -0.98611484,  2.99170383]))
('counter, ', 37)
('the received data is: ', array([-0.58066535,  0.1604901 , -0.73779182,  0.1546
8278, -0.62093751,
        0.77954759, -0.60974036,  0.19713686, -0.71806227, -0.84357168,
       -0.56844721,  0.49039659, -0.91655565,  2.95908732]))
('counter, ', 38)
('the received data is: ', array([-0.61175309,  0.14555722, -0.69334054,  0.1330
6747, -0.66932097,
        0.75170321, -0.59712586,  0.23108615, -0.76305024, -0.8921735 ,
       -0.54984831,  0.50445317, -0.88432765,  3.03838087]))
('counter, ', 39)
('the received data is: ', array([-0.61696454,  0.10652237, -0.63193592,  0.1112
5004, -0.6304564 ,
        0.70485352, -0.54309845,  0.16019201, -0.76271092, -0.85321204,
       -0.5057305 ,  0.52690007, -0.7571756 ,  3.11841586]))
('counter, ', 40)
```

Fig. 6. Running DSGD algorithm on distributed structure

machine learning system is vulnerable to evasion attack. Particularly, in our experiment we picked a house whose true value is 200K and is predicted as 210k by the distributed linear regression system. After attack, the adversary only makes tiny changes on the feature but the adversarial example is predicted as 280k which is significantly higher than its true value.

Fig.6 shows our test process in a Linux/Ubuntu machine. 6 worker nodes with two batches are running and the defender's convergence is dynamically shown during the minimization.

VI. Conclusion

In this paper we report our final project on adversarial distributed machine learning. We successfully implement distributed stochastic gradient descent (DSGD) on a distributed system, which is based on ParameterServer and considers synchronization and fault tolerance. We then evaluate the robustness of the distributed machine learning system by using an attack model. Our results show that the distributed machine learning system is quite vulnerable to evasion attacks and needs to consider the presence of adversaries to enhance its robustness.

References

[1] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed

machine learning with the parameter server." in *OSDI*, vol. 14, 2014, pp. 583–598.

[2] D. Lowd and C. Meek, "Adversarial learning," in *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 641–647.

[3] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma, "Adversarial classification," in *SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 99–108.

[4] M. Brückner and T. Scheffer, "Stackelberg games for adversarial prediction problems," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011, pp. 547–555.

[5] B. Li and Y. Vorobeychik, "Scalable optimization of randomized operational decisions in adversarial classification settings," in *Conference on Artificial Intelligence and Statistics*, 2015.

[6] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers: A case study on PDF malware classifiers," in *Network and Distributed System Security Symposium*, 2016.

[7] N. Šrndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *IEEE Symposium on Security and Privacy*, 2014, pp. 197–211.

[8] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *International Conference on World Wide Web*, 2010, pp. 281–290.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[10] K. Zhang, S. Alqahtani, and M. Demirbas, "A comparison of distributed machine learning platforms," in *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*. IEEE, 2017, pp. 1–9.

[11] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 703–710, 2010.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[13] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9. Boston, MA, USA, 2010.