Dynamic Slicing for Deep Neural Networks

Ziqi Zhang[*]
Key Laboratory of High-Confidence
Software Technologies (MOE),
Dept of Computer Science,
Peking University
Beijing, China
ziqi_zhang@pku.edu.cn

Yuanchun Li[*][†]
Microsoft Research
Beijing, China
Yuanchun.Li@microsoft.com

Yao Guo[†]
Key Laboratory of High-Confidence
Software Technologies (MOE),
Dept of Computer Science,
Peking University
Beijing, China
yaoguo@pku.edu.cn

Xiangqun Chen
Key Laboratory of High-Confidence
Software Technologies (MOE),
Dept of Computer Science,
Peking University
Beijing, China
cherry@pku.edu.cn

Yunxin Liu
Microsoft Research
Beijing, China
Yunxin.Liu@microsoft.com

ABSTRACT

Program slicing has been widely applied in a variety of software engineering tasks. However, existing program slicing techniques only deal with traditional programs that are constructed with instructions and variables, rather than neural networks that are composed of neurons and synapses. In this paper, we propose NNSlicer, the first approach for slicing deep neural networks based on data flow analysis. Our method understands the reaction of each neuron to an input based on the difference between its behavior activated by the input and the average behavior over the whole dataset. Then we quantify the neuron contributions to the slicing criterion by recursively backtracking from the output neurons, and calculate the slice as the neurons and the synapses with larger contributions. We demonstrate the usefulness and effectiveness of NNSlicer with three applications, including adversarial input detection, model pruning, and selective model protection. In all applications, NNSlicer significantly outperforms other baselines that do not rely on data flow analysis.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Software and its engineering → Dynamic analysis**.

---

[*]The first two authors contributed equally. This work is partly done while Ziqi Zhang was an intern at Microsoft Research. † Correspondence to: Yuanchun Li, Yao Guo.

KEYWORDS

Program slicing, deep neural networks, dynamic slicing, data flow analysis

1 INTRODUCTION

Program slicing [74] is widely used in software engineering for various tasks such as debugging [1], testing [3], and verification [14]. It aims to compute a set of statements (named *program slice*) that may affect the values at some points of interest (named *slicing criterion*). For example, by setting the slicing criterion to a specific output that generates an error, one can get a program slice that may be relevant to the error but much smaller in size than the whole program, thus much easier to analyze.

Existing program slicing techniques are mainly designed for traditional programs that are constructed with human-defined functions and instructions. Deep Neural Networks (DNNs), which have achieved remarkable success in many data-processing applications in recent years, can also be viewed as a special type of programs constructed with artificial neurons (a neuron is a mathematical function that receives one or more inputs and computes an output, such as the weighted sum or the maximum.) and synapses (the connections between neurons). However, the weights of synapses are learned by the machine and are usually hard for a human to understand. To the best of our knowledge, it has not been studied on whether and how a DNN can be analyzed meaningfully using program slicing techniques.

We apply the concept of program slicing to the area of DNNs and define **DNN slicing** as *computing a subset of neurons and synapses that may significantly affect the values of certain interested neurons.*

Slicing a DNN is interesting for a number of reasons. First, it is a widely-concerned problem that the decisions made by DNNs are difficult to explain or debug. Program slicing, hopefully, can be used to extract the operations that lead to a decision, making it easier to interpret. Second, the size of DNN is growing rapidly in recent years, with more than 25 million parameters (180 MB in size) in a state-of-the-art computer vision model [65] and 110 million parameters (340 MB in size) in a state-of-the-art natural language understanding model [18]. How to improve the model efficiency has become an important research problem. To this end, we believe program slicing has the potential to help reduce the model size significantly. Last but not least, partitioning the model into important slices and less-important slices can also benefit model protection, as one can prioritize the important slices if protecting the whole model is difficult or impossible.

DNN slicing introduces several new challenges as compared with traditional program slicing. First, unlike the instructions and variables in traditional programs that are themselves meaningful, a neuron in a DNN is usually a meaningless mathematical operator, whose behavior is determined by its learned weights and connections with other neurons. Thus, it is a challenging problem to understand the behavior of each neuron based on its connections and weights. Second, each output value of a DNN is affected by almost all neurons in the model. To generate a meaningful and concise slice, we must differentiate and characterize the neurons based on their contributions to the slicing criterion. Finally, the data flow graphs in traditional programs are usually sparse and small-scale, while the data flow graphs of a DNN may contain millions of neurons densely connected with each other. Analyzing a graph in such a large scale poses a much higher requirement on system efficiency.

In this paper, we present NNSlicer, a dynamic slicing technique for DNNs based on data flow analysis on neural networks. The *slicing criterion* is defined as a set of neurons with special meanings (such as the neurons in the last layer of an image-classification model whose outputs represent the probabilities of categories), while *a neural network slice* is defined as a subset of neurons in the neural network that exhibit larger effects to the slicing criterion. NNSlicer focuses on dynamic slicing in which a slice is corresponding to a set of input samples, rather than static slicing, which is input-independent.

NNSlicer consists of three phases: a profiling phase, a forward analysis phase, and a backward analysis phase. The profiling phase aims to model the average behavior of each neuron. The behavior of a neuron can be characterized by its activation values, which changes by feeding different data samples into the model. We feed all training data into the model and compute the average activation value of each neuron . These average values are used as the baseline to understand the reaction of each neuron to specific data samples.

The forward analysis feeds the interested data samples (the samples we want to compute slice with) into the model and records the activation value of each neuron. The difference between the recorded value and the average activation value computed in the profiling phase represents the neuron reaction to the data samples. The magnitude of the value difference indicates the sensitivity of the neuron with regard to the data samples.

However, the neurons with higher sensitivity are not necessarily more important for the slicing criterion, since the effect of the neuron may be eliminated or redirected to other outputs by its subsequent neurons. Thus, we further perform a backward analysis that backtracks the data flow from the output neurons to understand the contribution of each neuron. Specifically, the slice is initialized with the output neurons specified in the slicing criterion. We then iteratively analyze each neuron in the slice by calculating the contributions of its preceding neurons. The preceding neurons with higher contributions are added into the slice for further backward analysis.

We implement NNSlicer in TensorFlow through instrumentation and support the common operators in convolutional neural networks (CNNs). Our implementation is able to deal with large state-of-the-art CNN models, such as ResNet [34]. The time spent by NNSlicer to compute a slice for a data sample is around 40 seconds on ResNet10 and 550 seconds on ResNet18. Computing slices for batch input is much faster (about 3s and 40s per sample on ResNet10 and ResNet18).

To demonstrate the usefulness and effectiveness of NNSlicer, we further build three applications for adversarial defense, model pruning and model protection, respectively. First, we show that NNSlicer can be used to effectively detect adversarial samples. Specifically, we show that the slice computed for a data sample reflects how the prediction decision is made by the model, and the slices computed from adversarial samples significantly differ from the slices computed from normal samples. On average, the adversarial input detector implemented based on NNSlicer achieves a high precision of 0.83 and a perfect recall of 1.0. Second, we show that NNSlicer can be used to customize DNN models for a certain label space. Given a subset of model outputs, NNSlicer computes a slice for the outputs and generates a smaller model that is composed of the neurons and synapses in the slice. We show that the sliced model significantly outperforms other model-pruning methods. Notably, the sliced model can achieve high accuracy (above 80%) even without fine-tuning. Finally, NNSlicer can also be used to improve model protection. Specifically, we can selectively protect the important slices rather than the whole model, in order to reduce the protection overhead. We show that by hiding 50% parameters selected by NNSlicer, the exposed part can be nearly immune to model extraction attacks [53].

This paper makes the following contributions:

(1) To the best of our knowledge, this is the first paper to systematically explore and study the idea of dynamic DNN slicing.
(2) We implement a tool, NNSlicer, for dynamic DNN slicing on the popular deep learning framework TensorFlow. Our tool is scalable and efficient.
(3) We develop three interesting applications using DNN-slicing techniques and demonstrate the effectiveness of NNSlicer.

2 BACKGROUND AND RELATED WORK

2.1 Deep Neural Networks

Deep neural networks (DNNs) are inspired by the biological neural networks that constitute animal brains. A neural network is based on a collection of connected mathematical operation units called artificial neurons. Each connection (synapse) between neurons can

transmit a signal from one neuron to another. The receiving neuron can process the signal(s) and then signal downstream neurons connected to it. Typically, neurons are organized in layers. Different layers may perform different kinds of transformations on their inputs. For a certain kind of neuron, how it processes the signal is determined by its weights, which are learned by considering examples. For example, in image recognition, the neural network learns from example images that have been manually labeled as "cat" or "no cat" and uses the learned knowledge to identify cats in other images. Neural networks are good at capturing complex mapping relations between inputs and outputs that are difficult to express with a traditional rule-based program. Today, DNNs have been used on a variety of tasks, including computer vision [13], natural language processing [25], recommendation systems [20], and various software engineering tasks [30, 45], where they have produced results comparable to and in some cases superior to human experts.

A simple neural network is shown in Figure 1 (a). The neural network contains 9 neurons (2 input neurons, 2 output neurons and 5 intermediate neurons organized in 2 hidden layers) and 16 synapses. The first hidden layer contains 3 neurons that receive signals from the input neurons and send signals to the second hidden layer, which contains 2 neurons that further process the signals and forward to the output neurons. In this example, each neuron (except the input neurons) performs a weighted sum operation, which multiplies each received signal with a learned weight (marked on the synapses) and computes the sum as the neuron's value. Such weighted sum operations are common in today's neural networks, while usually accompanied by other operations such as rectifier, maximum, etc. The example neural network is for illustration purpose only and does not produce meaningful output. Real-world deep neural networks typically have millions of neurons and synapses [18, 34, 63].

2.2 Program Slicing

Program slicing is a fundamental technique to support various software engineering tasks in traditional programs, such as debugging, testing, optimization, and protection. It was originally introduced by Mark Weiser in 1981 [74] for imperative, procedureless programs. It aims to compute a program slice S that consists of all statements in program P that may affect the value of variable v in a statement x. The slice is defined for a slicing criterion $C = (x, v)$ where x is a statement in program P and v is variable in x. The slicing criterion represents an analysis demand relevant to an application, *e.g.*, in debugging, the criterion could be the instruction that causes a crash.

At first, only static program slicing was discussed, which analyzes the source code to find the statements that can affect the value of variable v at statement x for any possible input. Korel and Laski [42] introduced the idea of dynamic program slicing, which tries to find the statements that actually affect the value of a variable v for a particular execution of the program rather than all statements that may have affected v for any arbitrary execution of the program.

Program slicing techniques have been seeing a rapid development since its original definition. Various approaches are proposed to improve the slicing algorithms [36, 81], introduce other forms of

slicing [10, 33] and extending slicing ability to more programming languages and platforms [4, 5, 14, 69]. Meanwhile, many applications of program slicing techniques are proposed. Today, program slicing is widely used in various software engineering tasks including debugging [1], testing [3], software verification [14], software maintenance [23], and privacy analysis [44]. There are many comprehensive surveys [6, 62, 67] that summarize the advances in this area.

In this paper, we try to implement program slicing for deep neural networks, a completely different type of program that consists of mathematical operations with learned weights, rather than developer-written statements or variables.

2.3 Program Analysis for Neural Networks

Prior to ours, researchers had already attempted to analyze neural networks by applying or borrowing ideas from traditional program analysis techniques.

One of the most widely discussed applications of neural network analysis is to test the robustness of neural networks against adversarial attacks [43, 64, 78], which add small perturbation to the input to fool the DNN models. DeepXplore [56] proposed to use neuron coverage (the number of activated neurons) to measure the parts of a deep learning system exercised by a set of test inputs, and higher coverage usually means higher robustness. Since then, several new coverage metrics were introduced and various approaches were proposed to generate test inputs that maximize the coverage [19, 66, 75]. Training the model with the generated test inputs can improve its robustness and accuracy.

In addition to testing, many studies have attempted to detect adversarial inputs based on the internal behavior of neural networks. For example, Gopinath *et al.* [28] and Ma *et al.* [47] attempted to extract properties or invariants from the neuron activation state and use them to detect adversarial inputs. Wang *et al.* [70] borrowed the idea of mutation testing and found that adversarial samples are usually more sensitive to model mutations. Qiu *et al.* [57] and Wang *et al.* [73] extracted a path from the neural network that is the most critical for a sample, which can be used to distinguish normal and adversarial samples. The slice computed in our approach can also be viewed as the decision logic of the neural network and used to identify adversarial samples (discussed in Section 6.1).

As neural networks are inherently vulnerable and imprecise, researchers had also tried to provide a formal guarantee of security and correctness with the help of program analysis techniques, such as constraint solving [40], interval analysis [71, 72], symbolic execution [29], and abstract interpretation [24]. While promising, these techniques usually suffer from poor scalability - most of them cannot be applied to today's large neural networks.

There are also some existing work incorporating the idea of "slicing" to neural networks. Shen *et al.* [61] proposed slicing CNN feature maps to understand the appearance and dynamic features in the videos. Cai *et al.* [9] proposed to slice a DNN into different groups that can be assembled elastically to support dynamic workload. However, these approaches are not related to program slicing that aims to understand the internal logic of a program. Instead, they focused on training or assembling different parts of a DNN.

Qiu *et al.* 's work [57] is the closest to ours. Given an image classification model, they compute an effective path for each class, which contains the neurons and synapses that positively affect the prediction result. However, with regard to the slicing criterion, their effective paths may be incomplete (*i.e.* , missing important neurons and synapses such as the ones with negative contributions) and imprecise (*i.e.* , including less important neurons and synapses such as the ones yielding a large value for any input). Such shortcomings make their method less useful on applications other than adversarial defense (details in Section 6).

3 MOTIVATION AND GOAL

3.1 Motivation

Similar to traditional programs, we argue that slicing a DNN is also meaningful and useful for many important software engineering tasks, as illustrated below.

First, a DNN is a black box whose decisions are hard to interpret [80]. As a result, it is usually hard or even impossible for developers to understand when and why a DNN makes mistakes. As in traditional programs, the input would take a different control flow or data flow if it leads to failures. It would be potentially beneficial if there is a technique to automatically analyze the decision logic in DNNs.

Second, the size of state-of-the-art DNNs and their required computing power have been growing rapidly in recent years, thus it is highly desirable to reduce the size of DNNs to improve efficiency without sacrificing too much accuracy. Model pruning (removing some neurons and synapses) is one of the most widely-used techniques [32]. However, how to prune the model (*i.e.* which neurons and synapses to remove) is a key question, as we do not want to remove the critical structures that may lead to severe performance degradation. Deciding which neurons and synapses to prune is quite similar to computing a program slice.

Third, model protection, *i.e.* preventing the model from getting stolen, is on increasing demand as models are traded and shared across different organizations. Various techniques such as homomorphic encryption [12] and hardware enclave [68] can be used to protect models, but protection often brings performance degradation. A practical solution is to protect a part of the model instead of the whole model [68]. Thus, partitioning the neural network to important and unimportant slices may be beneficial as we can assign limited protection resources to more important slices.

The similarity between these tasks is the demand to find a subset of neurons and synapses that are more important in the decision-making process, which is the goal of this paper.

3.2 Problem Formulation

This section defines the concepts and symbols that will be used in this paper and formulates the goal of DNN slicing.

We first formulate the definition of neuron and synapse, two key concepts used throughout this paper. A neuron n in a neural network is a mathematical operator that takes one or more numerical inputs and yields one numerical output. n is said to be activated if its mathematical operation is executed, and the operation result y is called the activation value. A neuron n has one or more synapses $s_1, s_2, ..., s_k$, weighted with $w_1, w_2, ..., w_k$, respectively.

Table 1: Definition of symbols commonly used in this paper.

Symbol	Meaning
$\mathcal{M} = (\mathcal{N}, \mathcal{S})$	Model $\mathcal{M}$ with neuron set $\mathcal{N}$ and synapse set $\mathcal{S}$
n, y	Neuron n and its activation value y
s, x, w	Synapse s, its input value x and weight w
$\mathcal{I}, \xi$	Input dataset $\mathcal{I}$ and an input sample $\xi \in \mathcal{I}$
O, o	Output neuron set O and an output neuron $o \in O$
$C, \mathcal{M}^C$	Slicing criterion C and its corresponding slice
$CONTRIB$	Cumulative contribution of a neuron or a synapse *i.e.* the contribution to the slicing criterion
$contrib$	Local contribution of a neuron or a synapse *i.e.* the contribution generated in an operation
θ	Hyperparameter to control the slice quality

Each synapse s_i scales the activation value of another preceding neuron x_i with w_i and passes the scaled value to the neuron n as input. Similarly, the activation value of neuron n is also passed to other succeeding neurons by other synapses. The very last neurons that do not have succeeding neurons are the output neurons, whose activation values are the output of the neural network model.

Any modern DNN architecture can be viewed as a combination of such neurons and synapses. For example, a fully connected layer that maps 20 inputs to 10 outputs can be seen as a combination of 10 neurons, each of which computes the sum of values from 20 weighted synapses. A $16 \times 3 \times 3 \times 32$ filter in a convolutional layer can be viewed as 32 neurons, each of which computes the sum over 144 weighed synapses. A Rectified Linear Unit (ReLU) can be viewed as a neuron with only one synapse. Note that a neuron may be activated several times with different input values during the inference pass of a sample, such as the neurons in convolutional layers.

Based on the concept of neurons and synapses, we further define the symbols that will be commonly used later, as shown in Table 1. The formal definition of neural network slicing is given as follows:

DEFINITION 1. *(Neural network slicing) Let $\mathcal{M} = (\mathcal{N}, \mathcal{S})$ represents a neural network and $C = (\mathcal{I}, O)$ is a slicing criterion. $\mathcal{I} = \xi_1, \xi_2, \ldots, \xi_n$ is a set of model input samples of interest and $O = o_1, o_2, \ldots, o_k$ is a set of $\mathcal{M}$'s output neurons of interest. The goal of slicing is to compute subsets $\mathcal{N}_C \subset \mathcal{N}$ and $\mathcal{S}_C \subset \mathcal{S}$ with respect to C, denoted as $\mathcal{M}_C = (\mathcal{N}_C, \mathcal{S}_C)$, that significantly (above a predetermined threshold) contributes to the value of any output $o \in O$ for any input sample $\xi \in \mathcal{I}$.*

3.3 Challenges

There are three main challenges to slice a neural network.

(1) **Understanding the behavior of each neuron.** Unlike an instruction or a function in traditional programs, a neuron is typically a simple mathematical operation that does not have any high-level semantic meaning. The weights of all neurons in a model are learned as a whole to fit the training data, while each neuron is just a small building block whose functionality is vague. However, to compute a slice, we must first be able to differentiate the neurons based on their behavior.

(2) **Quantifying the contribution of each neuron.** In traditional program slicing, each instruction's contribution to the slicing criterion is binary: an instruction either affects or is irrelevant to the values of the criterion. In neural network slicing, almost all neurons are connected to the output neurons in the slicing criterion and contribute to the outputs more or less. It is difficult to quantify the contribution of each neuron to extract the most important neurons.

(3) **Dealing with large models.** Today's state-of-the-art neural networks typically contain millions of neurons that are densely connected. Analyzing a network on such a scale poses a higher demand for efficiency. How to design algorithms that can leverage existing computing resources to speed up the analysis is also a challenging problem.

4 OUR APPROACH: NNSLICER

We introduce NNSlicer to address the above challenges. Section 4.1 presents an overview of our approach. Section 4.2 describes how we understand neuron behaviors through differential analysis. Section 4.3 introduces our backward data flow analysis technique that quantifies the contribution of each neuron to the slicing criterion. Finally, Section 4.4 briefs how the computation power of GPUs and multi-core CPUs are utilized to improve the efficiency of our method.

4.1 Approach Overview

The overview of our approach is illustrated in Figure 1. The program under analysis in our system is a pretrained neural network model, whose weights are already learned to fit a training dataset. In Figure 1(a), the weight values are labeled on the corresponding synapses in the network. Our approach mainly consists of three phases, including a profiling phase, a forward analysis phase, and a backward analysis phase.

In the profiling phase, all samples in the training dataset are fed into the model, each sample produces an activation value at each neuron. We log the activation values of each neuron for all input samples and compute the mean activation value, which is the output of the profiling phase (as labeled on each neuron in Figure 1(b)). The mean activation values can be viewed as the behavioral standard of a neuron. Then, in the forward analysis phase, each interested sample in the slice criterion is fed into the model. We record the activation value of each neuron and compute its difference with the mean activation value obtained through profiling (as labeled on each neuron in Figure 1(c)). Such relative activation values represent the neuron reaction to the input sample. Finally, in the backward analysis phase, we start from the output neurons defined in the slicing criterion and iteratively compute the contributions of preceding synapses and neurons. The synapses and neurons with larger contributions are the slices computed for the slicing criterion. Each step is detailed and formulated in the following sections.

4.2 Profiling and Forward Analysis

The behavior of a neuron during an inference pass is represented as an activation value (or a list of activation values if the neuron was activated several times). The activation values are arbitrary

numbers produced by simple mathematical operations. We first need to make sense of the activation values. Specifically, does the neuron react positively or negatively, and how much?

Our method is inspired by the work on differential power analysis [41], which decodes the power consumption measurements of a circuit by testing the circuit with different inputs. The power consumption difference can be used to infer the input and program logic. In our case, the activation value of a neuron is like the power consumption measurement that barely makes sense by itself, but the difference between the activation values for different input samples can reveal how the neurons react to each sample.

Specifically, we use the difference between *the neuron behavior for an input sample* and *its average behavior for all training samples* to understand the neuron reaction to the input. Suppose ξ is an input sample and n is a neuron of model $\mathcal{M}$. By feeding ξ into $\mathcal{M}$, we would observe an activation value $y^n(\xi)$ at neuron n. $y^n(\xi) = mean_{i=1}^{m} y_i^n(\xi)$ if n is activated multiple times, where $y_i^n(\xi)$ is the i-th activation value and m is the total number of activations of n (*e.g.* $m = 1$ if n is a neuron in a fully connected layer, and m equals to the number of convolution operations performed by the filter if n is in a convolutional layer). The average neuron activation value over the whole training dataset $\mathcal{D}$ is calculated by:

$$\overline{y^n(\mathcal{D})} = \frac{\sum_{\xi \in \mathcal{D}} y^n(\xi)}{|\mathcal{D}|} \tag{1}$$

Such average activation values can be viewed as the behavioral standard of the neurons, which can be used as the baseline to measure a neuron's reaction to a specific data sample. Since $\overline{y^n}$ is not dependant on any specific input or output, it only needs to be computed once and can be used for different slicing goals.

In the forward analysis phase, we quantify the reaction of the neuron n for a specific data sample ξ as its relative activation value:

$$\Delta y^n(\xi) = y^n(\xi) - \overline{y^n(\mathcal{D})} \tag{2}$$

A positive $\Delta y^n(\xi)$ means that neuron n reacts more positively to the sample ξ than most other samples, and vice versa. The magnitude of $|\Delta y^n(\xi)|$ represents the sensitivity of n with regard to ξ. As an example, the output neuron of an image classification model that is trained to detect cats would be more sensitive and positively react to an image of a cat, as compared with an image of a truck.

4.3 Backward Analysis and Slice Extraction

The backward analysis aims to compute the contribution of each neuron and each synapse to the interested outputs in the slicing criterion. Note that the neuron's reaction to an input sample computed through the profiling and forward analysis is not equivalent to its contribution. For example, in an image classifier, a neuron that reacts sensitively to cat images may not have any contribution if our interested output is the "truck" label. To compute the contribution, we introduce a backward data flow analysis method.

In traditional programs, extracting the instructions and variables that contribute to a certain output is easy based on the data flow graph (DFG), which defines the data dependencies between the instructions and variables. A neural network can also be viewed as a data flow graph, but the graph is densely connected. For modern DNNs that are organized layer by layer, almost every neuron in one layer is connected to all neurons in the previous layer (as shown in
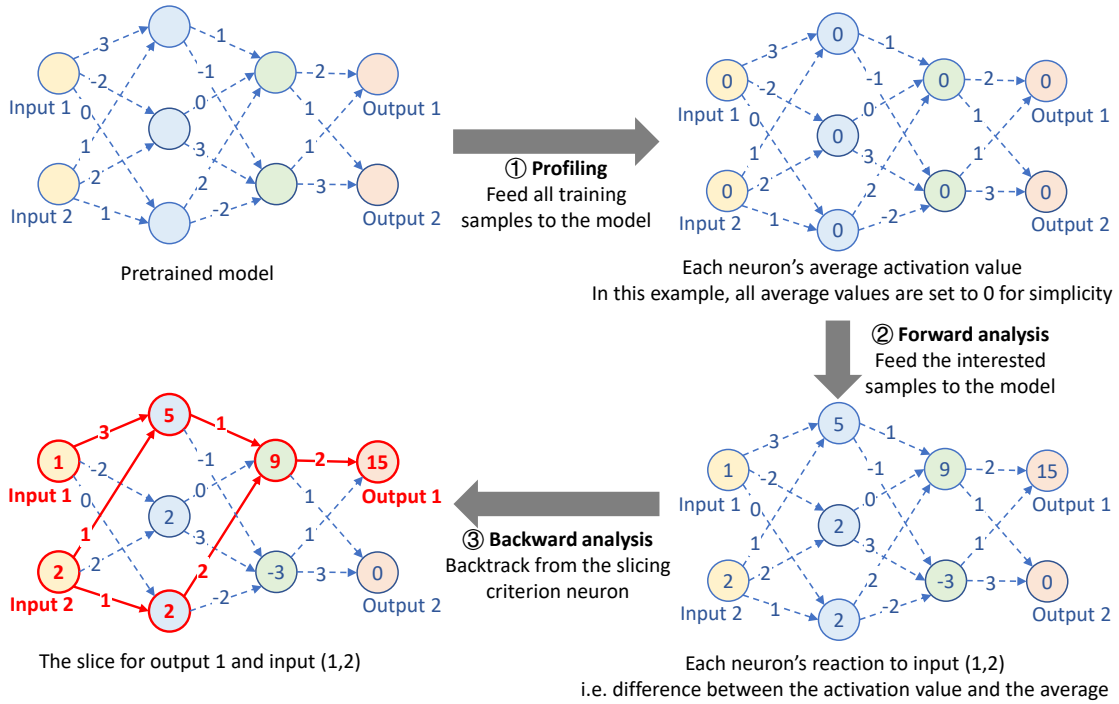
Figure 1: The overview of our approach.

Figure 1). Thus, we need to further analyze the data flow graph to measure the contribution of each neuron.

The contribution of a neuron or synapse is quantified as an integer in NNSlicer, denoted as *CONTRIB*. n is a critical neuron if $CONTRIB_n \neq 0$, and a critical neuron may contribute positively ($CONTRIB_n > 0$) or negatively ($CONTRIB_n < 0$) to the slicing criterion. The same for the synapses.

Our method to compute *CONTRIB* is to recursively compute the contributions of preceding neurons and synapses from back to front. Given a neural network and a list of target neurons, we first consider the neurons that are directly connected to the interested neurons, whose contribution can be extracted with their activation values (in detail later). Then we remove the target neurons from the network and set the neurons with non-zero contribution as the target neurons. We repeat the process until the target neurons do not have any neighboring neurons. The algorithm is described in Algorithm 1. Note that in practice neurons are organized as partially ordered layers, thus each iteration of Algorithm 1 deals with a single layer.

Algorithm 1 simplifies the problem of *computing cumulative contributions of all neurons and synapses in the whole network* to *computing local contributions of preceding neurons and synapses in an operation* (line 5). Local contributions mean the contributions generated solely by the operation. We use the weighted sum operator (a common operator in neural networks) to illustrate how we compute the contributions of preceding neurons and synapses.

In the weighted sum operator, the central neuron n has k synapses $(s_1, s_2, ..., s_k)$ that connect k preceding neurons $(n_1, n_2, ..., n_k)$ to n. The activation value of n is computed as $y = \sum_{i=1}^{k} w_i x_i$, where

Algorithm 1 ComputeContrib: Computing the contributions of neurons and synapses to a list of target neurons for an input sample

---

Require: A neural network model $\mathcal{M} = (\mathcal{N}, \mathcal{S})$, an input sample ξ and a list of target neurons O. A global table *CONTRIB* that stores the cumulative contribution of each neuron and synapse during the inference pass of ξ, initialized to 0.
1: Terminate if O is empty
2: Initialize $O' = \emptyset$
3: **for** each neuron $o \in O$ **do**
4:     Find o's preceding neurons and synapses (N', S')
5:     Compute local contributions of N' and S' as *contrib*
6:     Update *CONTRIB* with *contrib*
7: **end for**
8: **for** each neuron $n \in \mathcal{N}$ **do**
9:     Add n to O' if n is a predecessor of O and $CONTRIB_n \neq 0$
10: **end for**
11: Obtain $\mathcal{N}'$ by removing neurons in O from $\mathcal{N}$
12: Call **ComputeContrib** by setting $O = O'$ and $\mathcal{N} = \mathcal{N}'$
13: **return** The global cumulative contribution table *CONTRIB*

---

w_i is the weight of synapse s_i and x_i is the activation value of n_i. Suppose the cumulative contribution of n is $CONTRIB_n$, the local contribution $contrib_i$ of n_i and s_i is computed as:

$$contrib_i = CONTRIB_n \times \Delta y^n \times w_i \Delta x_i \qquad (3)$$

in which Δy^n is the relative activation value of the central neuron given by Equation 2 and Δx_i is the relative activation value of the neuron n_i (*i.e.* Δy^{n_i}). The product of Δy^n and $w_i \Delta x_i$ represents

the impact that n_i and s_i may have on the global contribution $CONTRIB_n$. For example, if Δy^n is negative and $w_i \Delta x_i$ is positive, it means that n_i enlarges the negativity of n, yielding an contribution that is opposite to $CONTRIB_n$.

The weighted sum operators take the vast majority in today's DNNs, but there are also other types of operators. In this paper, we focus on convolutional neural networks (CNNs). Table 2 shows five common operators that are enough to handle most existing CNN models. To support other architectures one only needs to define the method to compute local contributions for new operators, as shown in Table 2.

The cumulative contribution $CONTRIB$ of neuron n_i and synapse s_i in the operation is updated by their local contribution:

$$CONTRIB_{n_i} + = sign(contrib_i)$$
$$CONTRIB_{s_i} + = sign(contrib_i) \qquad (4)$$

We only keep the sign of the local contribution, as different operations may have different scales of local contributions.

However, updating the cumulative contribution for all neurons and synapses is time-consuming (a neuron with non-zero cumulative contribution introduces a new branch during backtracking) and may accumulate contributions from unimportant neurons and synapses. Thus, we limit the number of local contributions used to update $CONTRIB$. The importance of a local contribution is represented by its magnitude, and those with smaller magnitude can be excluded when updating $CONTRIB$. Specifically, we first sort the local contributions in ascending order of their magnitudes. The preceding neurons are sorted as $n_1, n_2, ..., n_k$. Then we try to find a maximum index j so that $n_1, ..., n_j$ can be excluded while the influence on the activation value of n is below the threshold θ. For example, in a weighted sum operation, the influence of excluding $n_1, ..., n_j$ is $|\sum_{i=j}^{k} w_i \Delta x_i / y|$. θ controls the amount of excluded local contributions with minimal influence on the functionality of an operation, and thus the generated slice can be directly used to make predictions without retraining (evaluated in Section 6.2). The value of the threshold θ can be set by different applications to control the size of the resulting slice.

So far the cumulative contribution $CONTRIB$ captures the contribution of neurons/synapses during the inference of a single input sample. For a slicing criterion $C = (I, O)$ that may contain multiple interested samples, the final cumulative contribution is the sum of the contribution for each sample $\xi \in I$. A slice for C is $\mathcal{M}^C = (\mathcal{N}^C, \mathcal{S}^C)$ where $\mathcal{N}^C$ and $\mathcal{S}^C$ are the neurons and synapses with non-zero contributions. One can also control the size of slice based on the contributions (as in §6.2).

4.4 GPU and Multi-thread Acceleration

NNSlicer takes a forward analysis pass and a backward analysis pass for each data sample $\xi \in I$ when computing the slices. It might be very time-consuming if $|I|$ is large. Since the process of computing slice for a data sample is independent of each other, we can take advantage of the parallel characteristic of GPU and multi-threading to accelerate the overall slicing process.

Specifically, for a large set of data samples $|I|$, we first run the profiling and forward analysis phases on GPU using large batches, as these two phases only involve forward computation. Then a large batch is separated into several small batches. The backward analysis of each small batch runs on the CPU as a separate thread. Finally, the batches are merged together to generate the slicing result.

5 IMPLEMENTATION & OVERHEAD

We implemented NNSlicer in Python with TensorFlow. The profiling and forward analysis are implemented based on TensorFlow's instrumentation mechanism. The multi-thread computing is implemented by the distributed python library Ray (https://ray.io).

We evaluated the time overhead of NNSlicer on a server that has 2 GeForce GTX 1080Ti GPUs, 2 Intel Xeon CPUs with 16 cores, and 64GB memory. Table 3 reports the slice time and the architecture complexity of three models. The time spent by NNSlicer to compute slice for a data sample is roughly 4s, 39s, and 553s for LeNet, ResNet10, and ResNet18 respectively. When computing slice for a batch of inputs, the speed is much faster, which is about 0.6s, 3.4s, and 45.2s per data sample for the three models respectively. Note that the profiling phase is not included when computing the slicing speed as it only needs to run once for a model.

6 APPLICATIONS

In this section, we describe three applications to demonstrate the usefulness and the effectiveness of NNSlicer, including adversarial defense, model pruning, and model protection. In each application, we describe why the application is meaningful, how NNSlicer can help, and how NNSlicer performs compared with other methods.

The main method which we compare NNSlicer against is the state-of-the-art work by Qiu *et al.* [57] (denoted as *EffectivePath* below). We also include some other baselines for more comparisons.

6.1 Adversarial Defense

Adversarial examples [64] are carefully-crafted inputs that may lead to wrong predictions. They are usually generated by adding small permutation to a benign input, which is barely noticeable by a human. Adversarial attacks may cause severe consequences, especially in safety- and security-critical scenarios.

As a result, adversarial defense became a hot research topic in both AI and SE communities. Many approaches tried to make the DNNs more robust through training [49, 77] or adding advanced architectures [51, 60], but it is still hard to obtain a 100% robust model. Instead, some researchers opted to take another direction: adversarial input detection [47, 57, 70], with which, the deep learning system can raise warnings or stop serving once suspicious inputs are detected. Thus, severe attacks can be avoided. In this section, we discuss how NNSlicer can be used to detect adversarial inputs.

6.1.1 Method. Our insight is that the slice computed by NNSlicer can be viewed as an abstraction of the decision process, and the decision processes of normal examples and adversarial examples are intuitively different. As shown in Figure 2, although the normal image and the adversarial one are indistinguishable for a human, their slices are different. Thus, by learning from the slices of large-scale normal examples, we can understand the normal decision process of the DNN. Therefore, given a new input, if its slice is distinctly different from the normal slices, it is very likely an adversarial input.

Table 2: Neuron operations considered in NNSlicer.

Operation	Usage	Math form	Local contribution of i-th input
Weighted sum	Convolutional layers and fully connected layers, etc.	$y = \sum_{i=1}^{k} w_i x_i$	$CONTRIB_n \times \Delta y \times w_i \Delta x_i$
Average	Average-pooling layers.	$y = \frac{1}{k} \sum_{i=1}^{k} x_i$	$CONTRIB_n \times \Delta y \times \Delta x_i$
Maximum	Max-pooling layers.	$y = \max_{i=1}^{k} x_i$	$CONTRIB_n \times \Delta y \times \Delta x_i \ if \ x_i = y \ else \ 0$
Rectify	ReLU activation.	$y = x \ if \ x > 0 \ else \ 0$	$CONTRIB_n \times \Delta y \times \Delta x \ if \ x > 0 \ else \ 0$
Scale	Batch normalization.	$y = \frac{x-\mu}{\sigma}$	$CONTRIB_n \times \Delta y \times \Delta x$

Table 3: The time spent to process an input sample in each phase. The profiling and forward analysis phases take the same amount of time as they both only require an inference pass.

Model	#Params	Profiling/Forward		Backward	
		Single	Batch	Single	Batch
LeNet	42784	3.0s	0.3s	0.5s	0.3s
ResNet10	300K	8.9s	0.4s	30.1s	3.0s
ResNet18	11M	9.6s	0.8s	543.0s	40.4s
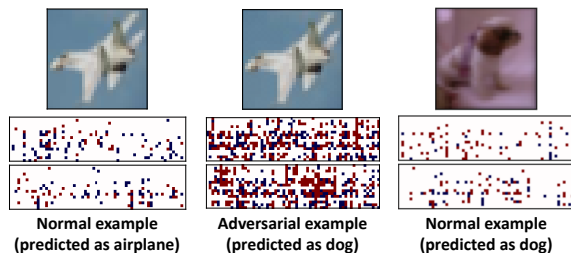
Figure 2: Normal and adversarial examples (top) and their visualized slices (bottom). Each pixel in the visualization represents a neuron from a random convolutional layer (separated to two rows). The neurons with non-zero contributions are colored (blue for neurons with positive contributions and red for those with negative contributions).

Specifically, suppose $\mathcal{M}$ is the DNN model that may accept adversarial inputs, ξ is an input sample and $\mathcal{M}(\xi)$ is the label of ξ predicted by $\mathcal{M}$. Using NNSlicer, we can compute a slice $\mathcal{M}_\xi$ for each input ξ by setting the slicing criterion as $C = (\xi, \mathcal{M}(\xi))$. We build a slice classifier F that predicts the label of an input ξ based on the slice computed for the input $\mathcal{M}_\xi$. By training F with a large number of normal samples, it can capture the mapping pattern between the slice shape and the corresponding output category. With the trained slice classifier F, an input ξ is identified as adversarial if $F(\mathcal{M}_\xi) \neq \mathcal{M}(\xi)$, *i.e.* the prediction made by the slice classifier is different from the prediction of the original DNN model.

The input of the slice classifier, *i.e.* a slice $\mathcal{M}_\xi$, is represented as a vector vec_ξ. Each element in vec_ξ corresponds to a synapse and its value is the contribution of the synapse (as described in Section 4.3). For the simplicity of the input and output representations, many classification algorithms may be used to build the slice classifier. We chose to use the decision tree [7] as it is easy to implement and debug.

Applying NNSlicer to adversarial-input detection has three advantages: (1) NNSlicer does not require modifying or retraining the original model, and thus NNSlicer can support any DNN models. (2) NNSlicer can scale up to support large state-of-the-art DNN models, while existing methods like ones by Ma *et al.* [47] and Gopinath *et al.* [28] can only support small models. (3) NNSlicer requires only the normal samples to build the defense, but existing methods [22, 48, 57, 73] need to train a detector with both normal and adversarial examples. As the attackers can always use new adversarial examples, NNSlicer is a much more realistic solution than those existing methods.

6.1.2 Evaluation. We compare our detection method with two baselines. For a fair comparison, the baseline methods use the same classifier to identify adversarial inputs as ours, while the inputs of the classifier are different. *FeatureMap* is a naive baseline that uses the feature maps of convolutional layers as the inputs of the classifier. *EffectivePath* is a more advanced baseline that uses the effective path generated by Qiu *et al.* [57] to train the classifier. The experiments were conducted on ResNet10 and the CIFAR-10 dataset (image size 32×32). All the classifiers were trained with 10,000 normal samples, using their respective feature extraction methods.

We tested NNSlicer and the two baselines on 17 attacks, covering gradient-based attacks, score-based attacks, and decision-based attacks. These attacks include FGSM [27] with per-pixel maximum modification of 2, 4 and 8 (relative to 256 and referred to as FGSM_2, FGSM_4 and FGSM_8, respectively), Deepfool [50] with constrain norm L_2 and L_∞ (referred to as DeepfoolL2 and DeepfoolLinf), JSMA [55] attack, PGD [49] attack with random start and per-pixel maximum modification of 2, 4 and 8 (referred to as RPGD_2, RPGD_4 and RPGD_8), the L_2 version of CW attack [11] (CWL2), ADef attack [2], an attack that just perturbs a pixel (SinglePixel), a greedy local-search attack [52] (LocalSearch), a boundary attack [8] (Boundary), an attack of spatial transformation [21] (Spatial), an

attack that performs binary search between a normal sample and its adversarial instance (Pointwise), and an attack that blurs the input until it is misclassified (GaussianBlur). All the attacks are implemented with Foolbox [58].

For each attack method, we generate adversarial examples from 500 randomly picked normal examples. The examples that successfully mislead the model are fed to the detector with their normal examples. We compute the precision, recall and F1 score of each detector on each attack, as shown in Table 4.

According to the experiment result, NNSlicer is very effective in detecting adversarial inputs with an average recall of 100% and an average precision of 83%, which means that NNSlicer is able to correctly identify all the adversarial examples generated with these attack methods (no false negative). Meanwhile, most of the inputs identified by NNSlicer are indeed adversarial inputs, while only a few normal samples are misidentified (false positives). Although *EffectivePath* also achieves a perfect recall, its precision is much lower, meaning that the detector may easily misclassify normal samples as adversarial inputs. The average recall of 63% in *FeatureMap* represents the feature maps between normal examples and the adversarial examples are barely discriminative. This phenomenon indicates the demand for NNSlicer to explore the mechanism of neural networks.

6.2 Network Simplification and Pruning

The size and complexity of DNN models grow rapidly. Although these huge models achieve high scores on complicated datasets, they are cumbersome and slow in real-world, task-specific applications. How to reduce the model size and speed up the computation is crucial to the DNN applications.

One acceleration technique is to prune trivial synapses of a large model to generate a light-weight one. With redundant weights trimmed off, the computation of executing the model may be reduced. Existing network-pruning methods focus on reducing the network architecture of models for all the output classes [46]. With DNN slicing, NNSlicer enables more flexible network simplification and pruning by focusing on a targeted subset of output classes. That is, for a subset of the original output classes of a model, NNSlicer can decide the proper model slices for the targeted output classes. Thus, NNSlicer can generate a smaller model for the targeted output classes with higher model accuracy. This advantage of NNSlicer is highly desirable in real-world applications that usually deal with a small set of output classes (e.g., classifying only different dogs rather than 1,000 types of animals).

6.2.1 Method. NNSlicer can pick out neurons and synapses critical to a slice criterion $C = (\mathcal{I}, O)$. By setting O to the set of interested target classes, NNSlicer can compute $CONTRIB_s$ for each synapse s, which represents the synapse's importance to the target classes. We can trim out the less important synapses and get a model that still functions on the target classes.

Specifically, suppose we want to prune $\mathcal{M}$ for target classes O^T with prune ratio r. Let $\mathcal{I}^T$ be the set of data samples belonging to the interested classes. $CONTRIB^T$ is the cumulative contributions computed by NNSlicer, and $CONTRIB_s^T$ is the contribution of synapse s. For each layer l, we sort all synapses in the layer $\mathcal{S}_l$ by the ascending order of their contributions magnitudes. The first
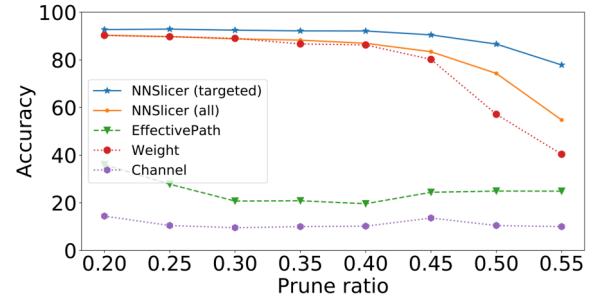

Figure 3: Accuracy of the pruned models without fine-tuning.

$r \times |\mathcal{S}_l|$ synapses are pruned, and a neuron is also pruned if its synapses are all pruned.

6.2.2 Evaluation. To evaluate the ability of NNSlicer to targeted pruning, each of 210 subsets of CIFAR10's 10 output classes is used as the target classes O^T. *NNSlicer (targeted)* represents to prune synapses according to the contributions computed for the target classes O^T. *NNSlicer (all)* represents to prune according to the contributions computed for all output classes O. The comparison between *NNSlicer (targeted)* and *NNSlicer (all)* demonstrates NNSlicer's ability in target classes. We also compare it with several baselines. *EffectivePath* represents pruning synapses based on the feature computed in [57]. *Weight* is based on the absolute synapse weights, where the synapses with the smallest weights are trimmed [31]. Similarly, *Channel* represents to prune the least important neurons by the average connected weight value [35]. Both *Weight* and *Channel* are widely used techniques in the field of network pruning.

Figure 3 shows the average accuracy over possible target classes. The accuracy of *NNSlicer (targeted)* is always high and is around 80% when 55% of weights are pruned. The accuracy of *EffectivePath* and *Channel* are both low in the figure. The accuracy of *NNSlicer (all)* and *Weight* is high only when the prune ratio is below 45%. The comparison between *NNSlicer (targeted)* and *NNSlicer (all)* demonstrates the ability of NNSlicer to prune for specific classes. The large gap between *NNSlicer (targeted)* and *EffectivePath* indicates the advantage of NNSlicer to the feature computed by [57].

When the prune ratio becomes larger, we further evaluate the performance with fine-tuning. To do it, the pruned models are retrained on 10k samples for 1 epoch. Figure 4 shows the performance of the fine-tuned models on two sets of target classes. The fine-tuned model of NNSlicer is noticeably higher than other methods. It shows that *NNSlicer (targeted)* preserves the model's capability on targets even when a large portion of weights is trimmed. A short fine-tuning (1 epoch in this case) is enough for the model to achieve high accuracy.

One possible reason for NNSlicer's good performance is that it preserves the model's ability to target classes at the cost of other non-target classes. In an extra experiment, the performance of *NNSlicer (targeted)* on non-target classes is remarkably lower than target classes. On the other hand, the difference of *Weight* is small. It means NNSlicer can decompose the model over classes and make

Table 4: Adversarial input detection accuracy for different attack methods.

Attack Method		*EffectivePath*			*FeatureMap*			NNSlicer		
		F1	precision	recall	F1	precision	recall	F1	precision	recall
Gradient-based	FGSM_2	0.82	0.69	**1.00**	0.55	0.58	0.53	**0.90**	**0.82**	**1.00**
	FGSM_4	0.71	0.56	**1.00**	0.55	0.60	0.52	**0.91**	**0.84**	**1.00**
	FGSM_8	0.88	0.79	**1.00**	0.58	0.62	0.55	**0.92**	**0.84**	**1.00**
	DeepFoolLinf	0.78	0.64	**1.00**	0.68	0.68	0.68	**0.92**	**0.85**	**1.00**
	DeepFoolL2	0.78	0.64	**1.00**	0.66	0.68	0.66	**0.92**	**0.85**	**1.00**
	JSMA	0.82	0.69	**1.00**	0.66	0.67	0.66	**0.92**	**0.85**	**1.00**
	RPGD_2	0.78	0.64	**1.00**	0.59	0.63	0.56	**0.91**	**0.84**	**1.00**
	RPGD_4	0.82	0.69	**1.00**	0.55	0.62	0.50	**0.92**	**0.85**	**1.00**
	RPGD_8	0.75	0.60	**1.00**	0.55	0.62	0.50	**0.92**	**0.85**	**1.00**
	CWL2	0.78	0.64	**1.00**	0.66	0.68	0.66	**0.92**	**0.85**	**1.00**
	ADef	0.85	0.73	**1.00**	0.66	0.67	0.66	**0.91**	**0.84**	**1.00**
Score-based	SinglePixel	0.67	0.50	**1.00**	0.52	0.44	0.64	**0.79**	**0.65**	**1.00**
	LocalSearch	0.83	0.71	**1.00**	0.67	0.67	0.68	**0.92**	**0.84**	**1.00**
Decision-based	Boundary	0.82	0.69	**1.00**	0.69	0.69	0.70	**0.92**	**0.85**	**1.00**
	Spatial	0.78	0.64	**1.00**	0.59	0.56	0.64	**0.87**	**0.77**	**1.00**
	Pointwise	0.87	0.76	**1.00**	0.68	0.69	0.69	**0.92**	**0.85**	**1.00**
	GaussianBlur	0.87	0.76	**1.00**	0.68	0.68	0.70	**0.92**	**0.85**	**1.00**
Average		0.80	0.67	**1.00**	0.62	0.63	0.62	**0.91**	**0.83**	**1.00**
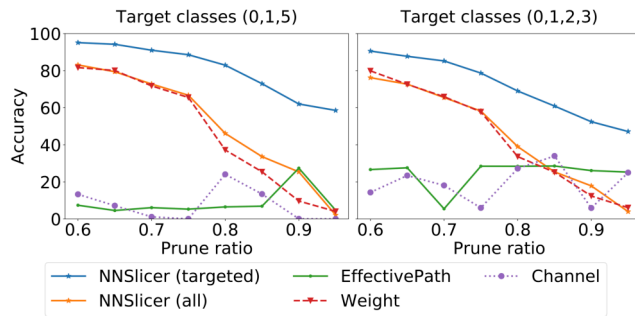
Figure 4: Accuracy of the pruned models after fine-tuning for one epoch.

a trade-off to conserve the ability on target classes. A similar phenomenon is observed in model protection and will be discussed in Section 6.3.

6.3 Model Protection

DNN models are becoming valuable assets due to the high cost of the training process, including collecting a large amount of data, expensive GPU usage, and enormous power consumption. However, an attacker may retain (or steal) the functionality of a model at a comparatively low cost [15, 37, 38, 53, 59]. How to protect models from being stolen is becoming an increasingly important problem, particularly in the emerging edge computing where models are deployed to edge servers or even end devices.

Existing solutions of model protection usually leverage encryption, using homomorphic encryption [12, 26, 79] or zero knowledge

proof [76], or running a model inside trusted execution environments [16, 17, 68]. All sensitive computation is conducted in the encrypted mode. However, the cost of these protected computations is high. For example, CryptoNets [26] takes around 300s to execute a model on the small MNIST dataset. To reduce the cost of model protection, one approach is to secure the important computation only, where NNSlicer may help.

6.3.1 Method. The existing model protection work is constrained to protecting the model w.r.t. the whole label space [39, 54]. But the importance of outputs may vary. For some outputs, the data is more difficult to collect, or the annotation is particularly more expensive. Because NNSlicer can slice model for certain classes, it can help to find significant components for the expensive classes and protect them. We propose to incorporate targeted protection in this scenario. Compared to existing work, our method is more flexible and can customize the protection target. NNSlicer selects synapses from a model and protects their weights. The way to select synapses is similar to Section 6.2 but NNSlicer selects the most crucial synapses for the target classes. The selected synapses are protected from attackers who have to recover the protected synapses through retraining to obtain the whole model.

6.3.2 Evaluation. In the experiment, we assume a strong attacker who has a training dataset. The attacker's dataset size is called the budget [53]. As NNSlicer protects a limited ratio of synapses, we use the metric of the accuracy of protected classes after re-training for 5 epochs. A lower accuracy stands for better protection. We compare with three baselines: *EffectivePath*, *Weight*, and *Random*. *EffectivePath* and *Weight* are the same methods used in Section 6.2. *Random* is to randomly select synapses.

Figure 5 shows the accuracy of the protected classes (Target classes, the left figure) and the accuracy of all classes (All classes, the right figure). It can be observed that, after guarded by NNSlicer,
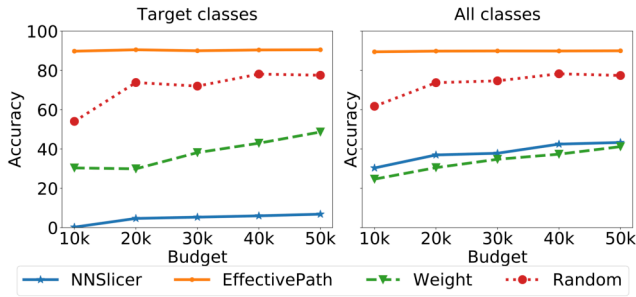
Figure 5: The accuracy achieved by retraining the models for 5 epochs. 50% of the parameters (selected with different methods) in the model are hidden, and the attacker tries to recover them through retraining. The x-axis is the attacker's budget (*i.e.* number of samples used to retrain). A lower accuracy achieved with a fixed budget means better protection.

the retrained accuracy on target classes is below 10%, even when the budget (*i.e.* number of samples) achieves 50k. The small accuracy stands for strong protection. On the contrary, the accuracy of other methods is all above 30%. For *EffectivePath*, the accuracy achieves 90%, which means it can not protect target classes at all.

The right figure of Figure 5 illustrates why NNSlicer achieves better protection. Compared to *Weight*, although the accuracy of NNSlicer on target classes is obviously lower, the accuracy over the whole dataset is higher. It means the accuracy of non-target classes is very high and NNSlicer do not protect them. This trade-off between target classes and non-target classes is similar to the finding in Section 6.2 and may be valuable for applications that desire to protect a small set of target classes.

7 LIMITATIONS AND DISCUSSION

This section highlights some of the limitations of NNSlicer and discusses possible future directions.

DNN architectures. We only considered five common operations that are commonly used in CNN models, while some operations used in other architectures are not included, such as recurrent neural networks (RNNs) and graph convolutional networks (GCN). These architectures should be easy to support in the future by adding backtracking rules for new operators.

Scalability. In this paper, we did not conduct experiments on very large models and datasets due to limited time. For large DNN models with millions of weights, NNSlicer takes about 10 minutes to compute the slice for an input sample (as shown in Table 3). Building an adversarial defense (as in Section 6.1) for such a large model may take several days on a single machine. Although the process is slow, especially for in-lab experiments, we think it is acceptable in practice considering the fact that companies usually train a model on large clusters for several weeks.

Slicing criterion. We mainly discuss the slicing criterion concerning only output neurons, but slicing for an intermediate neuron

may also be interesting (similar to inspecting an intermediate variable in traditional programs). Such a flexible criterion definition may enable new applications, *e.g.* interpreting or debugging the neural network in finer granularity.

More applications. Beside the three applications discussed in this paper, there are many other applications that are interesting to consider. For example, is it possible to compose different slices to a new model? If it is the case, the way of training networks might be changed. Besides, is it possible to slice certain attributes from a trained model, such as a discriminatory attribute (race, gender, etc.) which we want to exclude from consideration when making decisions? Last but not least, how can NNSlicer be used to debug model and diagnose fragile weights? Section 6.1 has proved its ability to detect adversarial examples, a step forward is to find the deviant neurons or synapses that are critical for errors. Masking them out or adjusting their value may improve the model accuracy.

Other slicing techniques. NNSlicer relies on a set of inputs to compute the slice (*i.e.* dynamic slicing). There are various other slicing techniques that may be interesting to be applied to neural networks. For example, static slicing might be used to compute input-independent slices (as in Section 6.2) much faster as each input doesn't need to be processed separately. Conditioned slicing [10] may help the developers to understand the conditions (*e.g.* illumination, viewpoint, etc.) under which the DNN is more vulnerable. Amorphous slicing may be used to merge neurons and synapses inside the network and slim the network structure [33].

8 CONCLUDING REMARKS

This paper proposes the idea of dynamic slicing on deep neural networks and implements a tool named NNSlicer to compute slices for convolutional neural networks. The working process of NNSlicer consists of a profiling phase, a forward analysis phase, and a backward analysis phase. The profiling and forward analysis phases model the reaction of each neuron based on its activation values. The backward phase traces the data flow recursively from back to front and computes the contributions of each neuron and synapse, which are used to calculate the slice. The usefulness and effectiveness of NNSlicer are demonstrated with three applications on adversarial input detection, targeted model pruning, and selective model protection. The code and data of NNSlicer and all applications will be made available to the community.

REFERENCES

[1] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. 1993. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience* 23, 6 (1993), 589–616.
[2] Rima Alaifari, Giovanni S Alberti, and Tandri Gauksson. 2018. ADef: an iterative algorithm to construct adversarial deformations. *arXiv preprint arXiv:1804.07729* (2018).

[3] Stephan Arlt, Andreas Podelski, and Martin Wehrle. 2014. Reducing GUI test suites via program slicing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 270–281.

[4] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. 2019. Dynamic slicing for android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1154–1164.

[5] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent program slicing. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 109–120.

[6] David W Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Adv. Comput.* 62, 105178 (2004), 105–178.

[7] Leo Breiman. 2017. *Classification and regression trees*. Routledge.

[8] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2017. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248* (2017).

[9] Shaofeng Cai, Gang Chen, Beng Chin Ooi, and Jinyang Gao. 2019. Model slicing for supporting complex analytics with elastic inference cost and resource constraints. *VLDB Endowment* 13, 2 (2019), 86–99.

[10] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. 1998. Conditioned program slicing. *Information and Software Technology* 40, 11-12 (1998), 595–607.

[11] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–57.

[12] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. 2018. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953* (2018).

[13] Dan CireşAn, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. 2012. Multi-column deep neural network for traffic sign classification. *Neural networks* 32 (2012), 333–338.

[14] Edmund M Clarke, Masahiro Fujita, Sreeranga P Rajan, T Reps, Subash Shankar, and Tim Teitelbaum. 1999. Program slicing of hardware description languages. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 298–313.

[15] Jacson Rodrigues Correia-Silva, Rodrigo F Berriel, Claudine Badue, Alberto F de Souza, and Thiago Oliveira-Santos. 2018. Copycat CNN: Stealing knowledge by persuading confession with random non-labeled data. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[17] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (2019).

[19] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Deepstellar: model-based quantitative analysis of stateful deep learning systems. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 477–487.

[20] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. 2015. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *24th International Conference on World Wide Web*. 278–288.

[21] Logan Engstrom, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. 2017. A rotation and a translation suffice: Fooling cnns with simple transformations. *arXiv preprint arXiv:1712.02779* 1, 2 (2017), 3.

[22] Gil Fidel, Ron Bitton, and Asaf Shabtai. 2019. When Explainability Meets Adversarial Learning: Detecting Adversarial Examples using SHAP Signatures. *arXiv preprint arXiv:1909.03418* (2019).

[23] Keith Brian Gallagher and James R Lyle. 1991. Using program slicing in software maintenance. *IEEE transactions on software engineering* 8 (1991), 751–761.

[24] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18.

[25] Felix A Gers and E Schmidhuber. 2001. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks* 12, 6 (2001), 1333–1340.

[26] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*. 201–210.

[27] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[28] Divya Gopinath, Hayes Converse, Corina Pasareanu, and Ankur Taly. 2019. Property inference for deep neural networks. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 797–809.

[29] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S Pasareanu, and Sarfraz Khurshid. 2018. Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439* (2018).

[30] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[31] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations, ICLR*.

[32] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[33] Mark Harman, David Binkley, and Sebastian Danicic. 2003. Amorphous program slicing. *Journal of Systems and Software* 68, 1 (2003), 45–64.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition (CVPR)*. 770–778.

[35] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision (CVPR)*. 1389–1397.

[36] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.

[37] Xing Hu, Ling Liang, Lei Deng, Shuangchen Li, Xinfeng Xie, Yu Ji, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. 2019. Neural network model extraction attacks in edge devices by hearing architectural hints. *arXiv preprint arXiv:1903.03916* (2019).

[38] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. 2019. High-fidelity extraction of neural network models. *arXiv preprint arXiv:1909.01838* (2019).

[39] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. 2019. PRADA: protecting against DNN model stealing attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 512–527.

[40] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

[41] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Annual International Cryptology Conference*. Springer, 388–397.

[42] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163.

[43] Yingwei Li, Song Bai, Yuyin Zhou, Cihang Xie, Zhishuai Zhang, and Alan Yuille. 2020. Learning Transferable Adversarial Examples via Ghost Networks. In *AAAI Conference on Artificial Intelligence*, Vol. 34.

[44] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. 2017. Privacystreams: Enabling transparency in personal data processing for mobile apps. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 1–26.

[45] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[46] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270* (2018).

[47] Shiqing Ma and Yingqi Liu. 2019. NIC: Detecting adversarial samples with neural network invariant checking. In *26th Network and Distributed System Security Symposium (NDSS)*.

[48] Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E Houle, and James Bailey. 2018. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613* (2018).

[49] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[50] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *IEEE conference on computer vision and pattern recognition*. 2574–2582.

[51] Aamir Mustafa, Salman Khan, Munawar Hayat, Roland Goecke, Jianbing Shen, and Ling Shao. 2019. Adversarial defense by restricting the hidden space of deep neural networks. In *IEEE International Conference on Computer Vision (CVPR)*. 3385–3394.

[52] Nina Narodytska and Shiva Prasad Kasiviswanathan. 2016. Simple black-box adversarial perturbations for deep networks. *arXiv preprint arXiv:1612.06299* (2016).

[53] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff nets: Stealing functionality of black-box models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4954–4963.

[54] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Prediction poisoning: Utility-constrained defenses against model stealing attacks. *arXiv preprint arXiv:1906.10908* (2019).

[55] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 372–387.

[56] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *26th Symposium on Operating Systems Principles*. 1–18.

[57] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2019. Adversarial Defense Through Network Profiling Based Path Extraction. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4777–4786.

[58] Jonas Rauber, Wieland Brendel, and Matthias Bethge. 2017. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131* (2017).

[59] Robert Nikolai Reith, Thomas Schneider, and Oleksandr Tkachenko. 2019. Efficiently Stealing your Machine Learning Models. In *18th ACM Workshop on Privacy in the Electronic Society*. 198–210.

[60] Andrew Slavin Ross and Finale Doshi-Velez. 2018. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In *Thirty-second AAAI conference on artificial intelligence*.

[61] Jing Shao, Chen-Change Loy, Kai Kang, and Xiaogang Wang. 2016. Slicing convolutional neural network for crowd video understanding. In *IEEE Conference on Computer Vision and Pattern Recognition*. 5620–5628.

[62] Josep Silva. 2012. A vocabulary of program slicing-based techniques. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–41.

[63] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.

[64] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[65] Mingxing Tan and Quoc V Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946* (2019).

[66] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *40th International Conference on Software Engineering (ICSE)*. 303–314.

[67] Frank Tip. 1994. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam.

[68] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*

[69] Engin Uzuncaova and Sarfraz Khurshid. 2007. Kato: A program slicing tool for declarative specifications. In *29th International Conference on Software Engineering (ICSE)*. IEEE, 767–770.

[70] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial sample detection for deep neural network through model mutation testing. In *41st International Conference on Software Engineering (ICSE)*. 1245–1256.

[71] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 6367–6377.

[72] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium (USENIX Security 18)*. 1599–1614.

[73] Yulong Wang, Hang Su, Bo Zhang, and Xiaolin Hu. 2018. Interpret neural networks by identifying critical data routing paths. In *IEEE Conference on Computer Vision and Pattern Recognition*. 8906–8914.

[74] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, 439–449.

[75] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[76] Roman V Yampolskiy. 2011. AI-complete CAPTCHAs as zero knowledge proofs of access to an artificially intelligent system. *ISRN Artificial Intelligence* 2012 (2011).

[77] Dinghuai Zhang, Tianyuan Zhang, Yiping Lu, Zhanxing Zhu, and Bin Dong. 2019. You only propagate once: Accelerating adversarial training via maximal principle. In *Advances in Neural Information Processing Systems*. 227–238.

[78] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[79] Qiao Zhang, Cong Wang, Chunsheng Xin, and Hongyi Wu. 2019. CHEETAH: An Ultra-Fast, Approximation-Free, and Privacy-Preserved Neural Network Framework based on Joint Obscure Linear and Nonlinear Computations. *arXiv preprint arXiv:1911.05184* (2019).

[80] Quan-shi Zhang and Song-Chun Zhu. 2018. Visual interpretability for deep learning: a survey. *Frontiers of Information Technology & Electronic Engineering* 19, 1 (2018), 27–39.

[81] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering (ICSE)*. 319–329.

(2018).