A Protection against the Extraction of Neural Network Models

Hervé Chabanne[1,2], Vincent Despiegel[1] and Linda Guiga[1,2]
[1] *Idemia, France*
[2] *Télécom Paris, Institut Polytechnique de Paris, France*

Keywords: CNN Model Protection, Oracle Query Access, Reverse-engineering, Adversarial Attacks, Layers Injection.

Abstract: Given oracle access to a Neural Network (NN), it is possible to extract its underlying model. We here introduce a protection by adding parasitic layers which keep the underlying NN's predictions mostly unchanged while complexifying the task of reverse-engineering. Our countermeasure relies on approximating a noisy identity mapping with a Convolutional NN. We explain why the introduction of new parasitic layers complexifies the attacks. We report experiments regarding the performance and the accuracy of the protected NN.

1 INTRODUCTION

Accurate Neural Networks require a carefully selected architecture and a long training on a large database. Thus, NN models' architecture and parameters are often considered intellectual property. Moreover, the knowledge of both the architecture and the parameters make adversarial attacks – among other kinds of attacks – easier: an attacker can easily generate small input noise that is undetectable by the human eye but still changes the model's predictions (Papernot et al., 2016; Akhtar and Mian, 2018).

Several papers (Carlini et al., 2020; Milli et al., 2019; Rolnick and Körding, 2019; Jagielski et al., 2019) have exploited the fact that the layers of a ReLU Neural Network (NN) are piecewise linear functions to extract its underlying model's weights and architecture. Indeed, hyperplanes – separating the spaces where the ReLU NN is linear – split the model's input space, and recovering the boundaries formed by the hyperplanes enables the extraction of its weights and architecture.

These attacks aim to recover the original model or a functionally equivalent one.

Here, we show how to modify the naturally induced division of the input space by inserting parasitic layers between the NN layers. Our parasitic layers are going to approximate a function close to the identity mapping, following (He et al., 2019). Since this adds new polytopes – whose boundaries are the various hyperplanes –, it leaves the flow of data within the victim NN mostly unchanged, while disrupting the geometry accessible for extraction. Contrary to (He et al., 2019), we do not inject noise on layers directly, but add specific CNN layers which aims at producing similar outputs without degrading the performances. Our goal is to complexify the hyperplanes geometry of our NN independently of the NN's structure. This can be done dynamically (Remark 1).

To gauge the efficiency of our countermeasure, we measure how much the parisitic layers perturbate the overall geometry of the NN's hyperplanes. A way to achieve this is to check whether adversarial examples for the original NN are still effective against the modified network (Sec. 6.2).

After finishing this introduction, we recall the aforementioned extraction of RELU NN in Sec. 2. We show in Sec. 3, following (Zhang et al., 2019), how to approximate the identity through a Convolutional NN (CNN). We then describe our protection proposal in Sec. 4. In Sec. 5, we explain how adding a CNN approximating a noisy identity mapping mitigates model extraction attacks on NNs. Sec. 6 reports our experiments regarding the deterioration of performances and accuracy due to the addition of parasitic layers

1.1 Background

Today, Neural Networks (NNs) are used to perform all kinds of tasks, ranging from image processing (Simonyan and Zisserman, 2015) to malware detection (Kaspersky, 2020). Neural Networks are algorithms that, given an input x, compute an output o usually corresponding to either a classification or a probability. NNs are organized in layers. Each layer contains a

Chabanne, H., Despiegel, V. and Guiga, L.
A Protection against the Extraction of Neural Network Models.
DOI: 10.5220/0010373302580269
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 258-269
ISBN: 978-989-758-491-6
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

set of neurons. Neurons of a given layer are computed based on a subset from the previous layer's parameters and parameters called weights.

There are different types of layers. Among those are:

- Fully connected layers: Each neuron from a layer l_i is connected to all neurons from layer l_{i+1}. Thus, a neuron η_k^i in a layer l_i is computed as follows: $\eta_i = \sum_{j=1}^{n} \eta_j^{i-1} w_j^i$ where $\{\eta_j^{i-1}\}_j$ are the n neurons from the previous layer and $\{w_j\}_j$ are the layer's weights.
- Convolutional layers: These layers compute a convolution between one – or several – filter F and windows from the input, as follows:

$$O_{i,j} = \sum_{k=1}^{h} \sum_{l=1}^{w} X_{i+k,j+l} \cdot F_{k,l}$$

  The elements of the filter are the weights of the layer. The number of filters is the number of output channels. An input can have several channels. For instance, in image processing, the input of a model is usually an image with three channels, corresponding to the RGB colors.
- Batch Normalization layers: These layers aim at normalizing the input. To achieve this, they learn the mean and standard deviation over mini-batches of input, as well as γ and β parameters, and return:

$$O_i = \gamma_i \times \frac{x_i - E_B}{\sqrt{V_B + \varepsilon}} + \beta_i$$

  where $x = (x_1, ..., x_n)$ is the layer's input and E_B and V_B are the learnt mean and variance respectively. These layers aim at removing the scaling factor introduced through the previous layers. They make training faster and more efficient.

While the various layers of an NN are linear, each layer is followed by an activation function, applied to all of the layer's neurons. The activation function is used to activate or, on the contrary, deactivate some neurons. One of the most popular and simplest activation function is ReLU, defined as the maximum between 0 and the neuron.

NNs only composed of fully connected layers are called Fully Connected Networks (FCNs), while those which are mainly composed of convolutional layers are called Convolutional Neural Networks (CNNs).

A ReLU NN is a NN constituted by linear layers followed by ReLU activation functions.

Let us note that another common layer type is the pooling layer, whose goal is to reduce the dimensionality. Since the attacks at hand do not take those layers into account, we also put ourselves in the context where pooling layers are not considered.

1.2 Related Works

Different kinds of reverse engineering approaches have been introduced. Batina et al. recover NNs' structure through side channels, i.e. by measuring leakages like power consumption, electromagnetic radiation, and reaction time (Batina et al., 2019). These measurement attacks are common for embedded devices (e.g. smartcards). Fault attacks, which are also a typical threat to smartcards, are transposed to find NN models in (Breier et al., 2020). A weaker approach where the victim NN shares its cache memory with the attacker in the cloud is taken in (Hong et al., 2020; Yan et al., 2018). The protections to thwart these attacks are related to the victim NN implementation. As we here consider oracle access attacks, our countermeasures have to modify the NN's architecture itself.

A more detailed explanation of the attacks (Carlini et al., 2020; Milli et al., 2019; Rolnick and Körding, 2019; Jagielski et al., 2019) is given in the next Section.

It should be noted that the abstract model of NNs that we are looking at here has been introduced by (Shamir et al., 2019) while in the different context of adversarial examples. Similarly to (Shamir et al., 2019), the authors of (Moosavi-Dezfooli et al., 2015) use the hyperplanes introduced by the activation functions and the class boundaries they form in order to accurately compute adversarial examples, as well as the robustness of the original model. While this is not the primary application of our idea, its transposition to thwart adversarial examples seems intriguing. As a matter of fact, we are going to gauge the efficiency of our countermeasure thanks to adversarial attacks.

2 EXTRACTION OF NEURAL NETWORK MODELS

Several attacks (Carlini et al., 2020; Milli et al., 2019; Rolnick and Körding, 2019; Jagielski et al., 2019) have managed to recover a ReLU NN's weights. These attacks rely on the fact that ReLU is piecewise linear.

The attack model in (Milli et al., 2019), (Jagielski et al., 2019) and (Rolnick and Körding, 2019) is as follows:

- The victim model corresponds to a piecewise linear function

- The attacker can query the model
- The attacker aims at recovering the weights (and, in some cases (Rolnick and Körding, 2019), the architecture) of the victim model
- The victim model is composed of linear layers (such as FC ones), as well as ReLU activation functions.

Furthermore, (Rolnick and Körding, 2019) also assumes that the attacker does not know the structure (i.e. the number of neurons per layer) of the victim NN. In the case of (Carlini et al., 2020), the authors assumed that the attacker had access to the architecture, but not the weights. However, the authors mention their belief that the piecewise linearity of the NN is the only assumption fundamental to their work, even though they do not prove it in their paper.

This attack model corresponds to the case of online services, for instance, where users can query a model and get the output, but they do not have access to the architecture and parameters of the model.

(Carlini et al., 2020) is the only paper so far that proves the practicability of its attack for more than 2 layers of a given neural network, even though the theory of (Milli et al., 2019) applies to arbitrarily deep neural networks. Moreover, (Carlini et al., 2020) provides a much higher accuracy with much fewer queries to the victim we want to protect.

Let $\mathcal{V}(\eta,x)$ denote the input of neuron η, before applying the ReLU activation function, when the model's input is x. For a given neuron η at layer l, let us define its critical point as follows:

Definition 1. *When, for an input x, $\mathcal{V}(\eta,x) = 0$, the neuron η is said to be at a critical point. Moreover, x is called a witness of η being at a critical point.*

Finding at least one witness for a neuron η enables the attacker to compute η's critical hyperplane.

Definition 2. *A bent critical hyperplane for a neuron η is the piecewise linear boundary $\mathcal{B}$ such that $\mathcal{V}(\eta,x) = 0$ for all $x \in \mathcal{B}$.*

All three attacks recover the weights of each layer thanks to the following steps:

1. Identify critical points and deduce the critical hyperplanes
2. Filter out critical points from later layers
3. Deduce the weights up to the sign and up to an isomorphism
4. Find the weight signs

Although the way critical points are found and filtered out differ from an article to the other, all methods use the piecewise linearity of the ReLU activation. The main element in those attacks resides in the fact that each neuron is associated to one bent critical hyperplane (that exists because of the ReLU activation function), corresponding to the neuron's change of sign. That hyperplane's equation is what enables the attacker to deduce the weights.

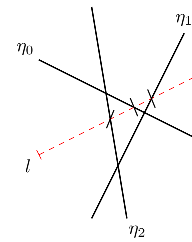

Figure 1: Hyperplanes for three neurons in the first layer. The dashed red line l enables the attacker to find the critical points indicated by the slashes.

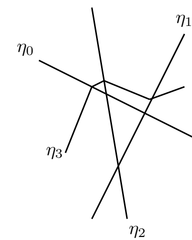

Figure 2: Hyperplanes are bent by boundaries from previous layers. For instance, η_3's hyperplane on the second layer is bent by the hyperplanes of η_0, η_1 and η_2 on the first layer.

Let us detail the attack in (Carlini et al., 2020), as it is the most accurate and requires the fewest queries to the victim model so far.

2.1 Finding Critical Points

The attacker chooses a random line l from the input space. Looking for non linearities through binary search in a large interval in that line enables the attacker to find several critical points (see Fig. 1).

However, the attacker knows neither what neurons these critical points are witnesses for, nor the said neurons' layer. Neurons from the first layer yield unbent hyperplanes, while those in the following layers are bent by the several previous ReLUs (see Fig. 2).

2.2 Recovering the Weights Up to a Sign

As seen before, the attacker has a set of witnesses for neurons in all layers. She can then carry out a differential attack in order to recover the weights and biases up to a sign.

Let us describe the attack on a simple case where the model only has one hidden layer, and the input vector space is $\chi = \mathbb{R}^N$. Let x^* be a witness for neuron

η^* being at a critical point. Define $\{e_i\}$ as the set of standard basis vectors of χ. The attacker computes:

$$\alpha_+^i = \frac{\partial f(x)}{\partial e_i}\bigg|_{x=x^*+e_i} \text{ and } \alpha_-^i = \frac{\partial f(x)}{\partial e_i}\bigg|_{x=x^*-e_i} \quad (1)$$

Then, because the activation function is $ReLU(x) = max(0,x)$, we have that: $\alpha_+^i - \alpha_-^i = \pm A_{j,i}^{(1)} \cdot A^{(2]}$. Thus, by computing:

$$\frac{\alpha_+^i - \alpha_+^i}{\alpha_+^1 - \alpha_+^1} \quad (2)$$

for all i, the attacker gets the weights up to a multiplicative scalar.

In the general case where the NN is deeper, and for a layer j, the attacker computes second partial derivatives $y_i = \{\frac{\partial^2 f}{\partial \delta_i^2}\}$ instead of the simple ones, where the δ_i take random values. She then solves a system of equations: $h_i \cdot w = y_i$, where h_i is the value of the previous layer – after the *ReLU* – for an input model $x^* + \delta_i$. Let us note that the attacker does not know whether neuron η^* is in the current layer. She therefore solves the system of equations for all layers, and only keeps the solution that appears most often. The biases can then easily be deduced from the weights.

To differentiate critical points of the current layer from other critical points, the differential attack is carried out on all the critical points and the attacker filters out the wrong critical points by observing the resulting traces.

2.2.1 Recovering Weight Signs

In this step, the attacker proceeds recursively. The attacker has a set $\mathcal{S}$ of witnesses for unknown neurons (as found in the previous step).

Let us suppose the attacker has managed to recover the correct model up to layer $j-1$, as well as the weights up to sign for layer j. Let us define the polytope at layer j containing x as:

$$\mathcal{S} = \{x + \delta \text{ s.t. } sign(\mathcal{V}(\eta, x)) = sign(\mathcal{V}(\eta, x+\delta))\} \quad (3)$$

Thus, this polytope corresponds to the open, convex subspace shaped by the critical hyperspaces.

The attacker can easily filter out the critical points x from previous layers since she already recovered the weights and biases up to layer j.

To filter out witnesses from layers deeper than $j+1$, the attacker relies on the fact that the polytopes of two distinct layers have a different shape with high probability.

Finally, the attacker recovers the sign of the weights through brute force using layer $j+1$'s witnesses. Let us note that when the victim NN is contractive, the sign recovery can be less expensive.

Thus, the attacker can recover the victim model's parameters recursively over the depth of the considered layer as described in the previous paragraphs. Moreover, even though the number of queries is linear, the work required is exponential, as explained in the previous paragraph.

3 APPROXIMATING THE IDENTITY THANKS TO CNNs

Our proposal is based on adding parasitic layers to the model we want to protect, and for those layers, we rely on a CNN approximating the identity. It results in the addition of dummy hyperplanes, as explained in Sec. 5. However, it is not enough to thwart the attack at hand. In order to mitigate the said attack, our parasitic CNNs approximate the identity to which we add a centered Gaussian noise. Sec. 5 details how this additional noise ensures that the introduced hyperplanes lie in the same space as the original ones.

Since CNNs are intrinsically nonlinear, approximating the identity – the simplest linear mathematical function – would appear to be a difficult learning task. However, thanks to the bias and the piecewise linearity of ReLU, CNNs manage to avoid the obstacle of the hyperplanes by shifting the input to a space where the activation function is linear. Therefore, CNNs manage to approximate the identity very accurately.

The simplicity of the task at hand is demonstrated in (Zhang et al., 2019). Indeed, the authors of (Zhang et al., 2019) manage to approximate the identity mapping using CNNs with few layers, few channels and only one training example from the MNIST dataset (LeCun et al., 2010).

First, they observe that while both CNNs and FCNs could approximate the identity on digits well when trained on three training examples from the MNIST dataset (LeCun et al., 2010), only CNNs generalize to examples outside of the digits scope. Moreover, they state that this bias can still be observed when the models are trained with the whole MINST training set.

In order to better characterize the observed bias, the authors take the worst case scenario: they only train FCNs and CNNs on a single training example. Contrary to what they expected, architectures that are not too deep manage some kind of generalization: FCNs output noisy images for inputs that are not the training example, while CNNs still manage to approximate the identity. Moreover, FCNs tend to correlate more to the constant function than to the identity. The output of CNNs' correlation with the identity function

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

decreases with a smaller input size and a higher filter size.

The authors of (Zhang et al., 2019) show – by providing possible filter values – that in their case, if the input has n channels, $2n$ channels suffice to approximate the identity mapping with only one training example. They also note that adding output featuremaps does help with training. Moreover, they use 5×5 filters for all their CNNs' layers. Finally, they explain that even though 20-layer CNNs can learn the identity mapping given enough training examples, shallower networks learn the task faster and provide a better approximation.

This ability of CNNs to learn the identity mapping from only one training example from the MNIST dataset and to generalize it to other datasets shows the simplicity of the task. We explain in Sec. 4 and 5 how this fact impairs the defense when the parasitic CNN approximates the identity mapping, and the necessity to approximate a noisy identity as well as to apply some constraints on the CNN's parameters.

4 OUR PROPOSAL

Let us consider a victim ReLU NN. The attack scenario described in Sec. 2 is based on the bent critical hyperplanes induced by the ReLU functions in the model. In (Carlini et al., 2020), the bent hyperplanes are especially used in the case of expansive NNs – i.e. for which a preimage does not always exist for a given value in the output space –, in order to filter out witnesses that are not useful to the studied layer. In order to make the attacker's task more complex, we propose to add artificial critical hyperplanes. Adding artificial hyperplanes would make the attack more complex: the attacker would have to filter out the artificial hyperplanes as well as the other layers' hyperplanes.

As explained in Sec. 3, CNNs can provide a very good approximation of the identity mapping. Moreover, they generalize well: with only a single trainable example from the MNIST dataset, CNNs up to 5 layers deep can still reach the target.

We propose to add dummy hyperplanes through the insertion, between two layers of the model to protect, of parasitic CNNs approximating an identity where a centered Gaussian noise has been added. The CNNs we add select *nb* neurons at random from the output of the previous layer, and approximate a noisy identity, where *nb* is smaller or equal to the output size of the previous layer.

Since CNNs approximate the identity well, inserting CNNs approximating the identity yields hyperplanes that do not impact a potential attacker. Indeed, as will be further detailed in Sec. 5 the CNN can make sure that the introduced CNNs are either far from the original ones or, on the contrary, very close and almost parallel to the original ones. In these cases, with high probability, an attacker would not notice the added layers, and would therefore be able to easily carry out her attack. Therefore, we need to apply further constraints on the parasitic CNNs. Instead of CNNs approximating the identity, we propose to insert CNNs approximating the identity where a centered Gaussian noise is added. Furthermore, these CNNs are trained with constraints on some of their parameters. Sec. 5 explains why the addition of the noise helps make the injected hyperplanes noticeable by a potential attacker.

Remark 1. *Note that we can think of a dynamic addition of parasitic CNNs approximating a noisy identity mapping. For instance, considering a client-server architecture where the server is making predictions; from a client query to another, different parasitic CNNs can be added in random places of the server's NN architecture, replacing the previous ones.*

Furthermore, the small CNN we add does not act on all neurons. This yields two advantages:

- The added CNN considered can be small, implying fewer computations during inference
- We can add different CNNs to different parts of the input, to further increase the difference in behavior between neurons

Fig. 3 shows an example of adding such an identity CNN between the first and the second layer of an NN with only one hidden layer.

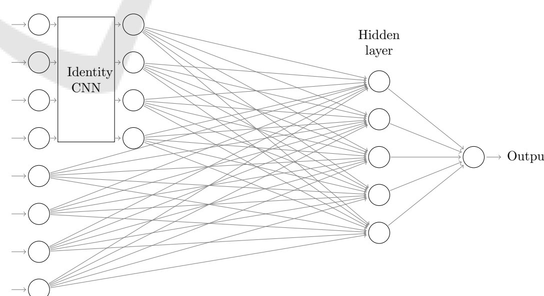

Figure 3: Neural Network with one hidden layer where a CNN approximating the identity has been added to approximate the first fours input neurons.

The CNN we add in our experiments consists of four hidden convolutional layers, with 5×5 filters (see Fig. 4). In some cases, we add a batch normalization layer after each covolutional one (see Fig. 5). Indeed, as recalled in Sec. 3, a CNN with few layers and 5×5 filters can already approximate the identity on 28×28 inputs with a single training example. Thus,

such a CNN is well adapted to learning the identity mapping on *nb* neurons, where *nb* is smaller or equal to the size of the previous layer's output. When the CNN receives the set of neurons from the considered layer, it first reshapes it into a square input with one channel, so that it is adapted to convolutional layers.

Moreover, for the much harder task tackled by the authors of (Zhang et al., 2019), for an input with n channels, $2n$ channels in the intermediary layers are enough to get a good approximation of the identity, even though more channels improve the accuracy. Since we do not constrain ourselves to training our CNN with a single example, we can limit the number of channels in the hidden layers to two – because we consider inputs with one channel. This enables us to minimize the number of additional computations for the dummy layers, with only a slight drop in the original model's accuracy.

5 COMPLEXITY OF EXTRACTION IN THE PRESENCE OF PARASITIC LAYERS

Adding a convolutional layer with k layers as described in the previous section results in adding k layers to the architecture while keeping almost the same accuracy. If those k layers add critical hyperplanes, then the complexity of extraction increases.

In this section, we first consider a CNN approximating the identity mapping added after the first layer in the victim NN. We further assume that there are fewer neurons in the second layer than in the first. We prove that in that case, the identity CNN does add hyperplanes with high probability. Then, we explain the need to approximate the need to approximate a noisy identity mapping rather than the identity itself.

Let us suppose we add a CNN Identity layer that takes $n \times n$ inputs, and the original input size is m. Let $\{F_{i,j}\}_{1\leq i\leq k,\ 1\leq j\leq k}$ be its associated filter. This would result in the following weight matrix C:

$$\begin{cases} C_{i\times n+j,(i+l)\times n+j+h} = F_{l,h} \\ \forall 1 \leq i,\ j \leq n-k+1 \text{ and } 1 \leq l,\ h \leq k \\ C_{i,i} = 1\ \forall i \geq (n-k+1)\times(n-k+1) \\ C_{i,j} = 0 \text{ otherwise} \end{cases} \quad (4)$$

Here, without loss of generality, we consider there is no padding.

This new layer adds at most $n \times n$ bent hyperplanes. This number decreases if two neurons η_i and η_j share the same hyperplane.

Let $\mathcal{V}(\eta_i, x)$ be the value of η_i before the activation function, if the model's input is x.

We need to consider two cases:

1. η_i and η_j are in different layers. Let us suppose that η_i's layer is l and that η_j's layer is $l+1$. If the layers are not consecutive, the η_j's hyperplane is bent by *ReLUs* from the layers in between, making the probability of the two hyperplanes matching very low.
2. η_i and η_j are in the same layer

5.1 First Case: η_i Is on Layer l and η_j Is on Layer $l+1$

Let us suppose that η_i is on the first layer, and η_j is on the second one. The output $z(x)$ of the first layer, for $x \in \chi$ is:

$$z(x) = A^{(1)}x + \beta^{(1)} \quad (5)$$

In this proof, the rows of $A^{(1)}$ are supposed to be linearly independent. This is an assumption made in (Milli et al., 2019) and in (Jagielski et al., 2019). As stated in (Jagielski et al., 2019), this is likely to be the case when the input's dimension is much larger than the first layer's. The authors of (Carlini et al., 2020) state that it is the case in most ReLU NNs, but not necessarily the most interesting ones. However, the general attack in (Carlini et al., 2020) for the cases where the model to protect is not contractive is more complex, and requires a layer by layer brute force attack for the sign recovery.

The output of the second layer is:

$$Out = C \cdot ReLU(z(x)) + \beta^{(2)} \quad (6)$$

Since the rows of $A^{(1)}$ are supposed to be linearly independent, for a given vector V, there exists a solution x^* such that $z(x^*) = V$, by the Rouché-Capelli theorem. If we select V so that $V_i \geq 0\ \forall i \leq m$, then V is not affected by the *ReLU*. We can therefore select a vector V such that, letting k be the convolutional layer's filter size:

$$\begin{aligned} &V_{(\lfloor \frac{j}{n} \rfloor + h)\times n + j\%n + l} = 0\ \forall 1 \leq l,h \leq k \\ &\text{(where } j\%n \text{ means } j \text{ modulo } n) \\ &\text{except for one value } i' \neq \eta_i,\ \text{where } V_{i'} = 1 \end{aligned} \quad (7)$$

Since this second layer is a convolutional one, $\beta_i^{(2)}$ is the same for all i on a given channel, denoted β. The window considered to compute η_j is zeroed out, except for one value. The filter weight associated to that value needs to be $-\beta$ to nullify η_j. Since we can repeat the process for all values of the window that are not η_i, all the filter weights except for that associated

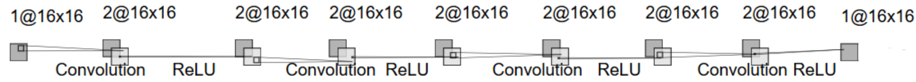

Figure 4: Parasitic CNN with 4 convolutional layers, with a ReLU activation function after each convolution. Image generated thanks to (LeNail, 2019).

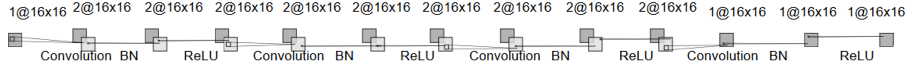

Figure 5: Parasitic CNN with 4 convolutional layers, with a batch normalization layer (BN) and a ReLU activation function after each convolution. Image generated thanks to (LeNail, 2019).

to η_i need to be $-\beta$ except for the one associated with η_i. This is not the case with high probability. Thus, with high probability, $\eta_i = 0$ does not imply that $\eta_j = 0$.

For deeper layers, even though we cannot select any vector V, it is highly unlikely for the following equation to happen:

$$z_i(x) = 0 \iff C_j ReLU(z(x) + \beta^{(2)}) = 0 \quad (8)$$

When η_i is not in the window used to compute η_j, it is even less likely to be the case.

Therefore, two neurons on different layers are very likely to have different critical hyperplanes.

5.2 Second Case: η_i and η_j Are in the Same Layer

Let us suppose that η_i and η_j are in layer l. Let l be the first convolutional layer. Moreover, let us suppose that the CNN is set after the first layer of the model we want to protect. Then l's input is:

$$z(x) = ReLU(A^{(1)}x + \beta^{(1)}) \quad (9)$$

where x is the model's input.

Let us also suppose, without loss of generality, that $j > i$. This means that the windows used to compute the two neurons are not identical. With high probability, one of the filter values associated with the disjoint window values is nonzero. For simplicity, and without loss of generality, let us suppose, in what follows, that $F_{1,1}$ is such a filter value. Thus, in what follows, we suppose that $F_{1,1} \neq 0$.

Case Where $\beta = 0$. As explained before, we can find x^* such that $z(x^*)_{\lfloor \frac{i}{n} \rfloor \times n + i\%n} = 1$ and $z(x^*)_h = 0$ otherwise. Since $j > i$, $z(x^*)_{\lfloor \frac{i}{n} \rfloor \times n + i\%n}$ is not in the window used to compute η_j, but it is in η_i's window. In this case, $\eta_i \neq 0$ and $\eta_j = 0$. Thus, η_i and η_j do not share the same critical hyperplane.

Case Where $\beta \neq 0$. If $\beta \neq 0$, we cannot directly apply the previous reasoning. Let x^* be a witness for η_j being at a critical point. Let us show that we can find an input x^{**} such that $\eta_j = 0$ but $\eta_i \neq 0$.

If x^* already satisfies this property, our work is done. Otherwise, x^* is such that $\eta_i = \eta_j = 0$. As explained before, there exists an input to the NN x' such that $(A^{(1)} \cdot x')_{\lfloor \frac{i}{n} \rfloor \times n + i\%n} = a$ with $a > 0$ and $(A^{(1)} \cdot x')_h = 0$ otherwise. Then, by piecewise linearity of z, we have, for a large enough, that $z(x^* + x')_{\lfloor \frac{i}{n} \rfloor \times n + i\%n} > z(x^*)_{\lfloor \frac{i}{n} \rfloor \times n + i\%n}$. Moreover, for all other indices h, $z(x^* + x')_h = z(x^*)_h$. Let us consider $x^{**} = x^* + x'$. We have that $z_{\lfloor \frac{i}{n} \rfloor \times n + i\%n}$ is not in η_j's window, which means that η_j remains unchanged and $\eta_j = 0$ when the NN's input is x^{**}. On the other hand, η_i's value changes since one of its window values changes and $F_{1,1} \neq 0$. Thus, $\eta_i \neq 0$. Therefore, we can indeed find x^{**} such that $\eta_j = 0$ but $\eta_i \neq 0$.

Let us now consider the case where η_i and η_j are on deeper layers, in which case the previous proof does not hold. Let $i = i_1 \times n + i_2$ and $j = j_1 \times n + j_2$, where $i_1 \neq j_1$, or $i_2 \neq j_2$, or both. Let also F be the filter of the considered convolutional layer, of size $k \times k$.

If η_i and η_j share the same hyperplane, then whenever z is such that $C_i z + \beta = 0$, we have that:

$$\sum_{l=1}^{k} \sum_{l=1}^{k} F_{l,h}(z_{(i_1+l)\times n + i_2 + h} - z_{(j_1+l)\times n + j_2 + h}) = 0 \quad (10)$$

Since Eq. 10 needs to hold for all the z that are on the hyperplane, this equation is very unlikely to hold.

Therefore, with a very high probability, no two neurons in the same layer share the same hyperplane.

5.3 Approximating a Gaussian Noise

As explained before, adding CNNs approximating the identity to a victim neural network adds hyperplanes. However, this does not necessarily lead to an increased complexity for the extraction attacks at hand.

Indeed, the identity CNN might avoid the complexity of the task by isolating the newly introduced hyperplanes – meaning the critical points are far from the input space –, or very close and parallel to the original hyperplanes – i.e. the critical points correspond to a small translation from the original points. The first case can be achieved by increasing the bias in the convolutional layers, so that all values are made significantly positive. This ensures that no value is zeroed out during the computations. The last layer's bias then translates the values back to their original position. In both cases, the attacker would not notice the introduced hyperplanes, thus defeating the purpose of the parasitic CNN.

The authors of (He et al., 2019) inject normal noise during a model's training as a way of mitigating adversarial attacks. They introduce a parameter, α, trained along the original model so that $\alpha \times \mathcal{N}$ – where $\mathcal{N}$ is a fixed Gaussian noise – is added to some layers. Furthermore, they add adversarial examples to the training set to prevent α from converging to 0.

Similarly to (He et al., 2019), we inject noise into our layers in order to avoid cases where the CNN we add is not detectable by an attacker. However, our method separates the training of the added CNN from that of the model to protect. Having to train the original model for each parasitic CNN would result in too much overhead. We inject a fixed Gaussian noise to the labels during the training of our CNN approximating the identity.

The standard deviation of this added noise is selected so as to avoid a significant drop in the original model's accuracy. Let us note that even though the selected standard deviation might depend on the victim network, several CNNs approximating the identity are trained independently from the victim network, and the victim can then select one or several CNNs adapted to the network at hand.

Since the noise added is fixed, it only constitutes a translation of the victim hyperplanes, and can be approximated by the CNN through an increase in the bias β. We avoid this case by bounding the bias to a small value ($||\beta||_2 < \varepsilon$) or eliminating the bias ($\beta = 0$). This makes the learning task more complex, and forces the filter values themselves to change, thus preventing the introduced hyperplanes from being simple translations of the original ones.

Let us consider, for instance, the case where one convolutional layer is introduced. As before, let C be the matrix associated to the layer and $\mathcal{N}$ be the fixed Gaussian noise. Then the optimization problem becomes:

$$\sum_{1 \leq k \leq m} C_{i,k} x_k = x_i + \mathcal{N}_i \; \forall 1 \leq i \leq n \tag{11}$$

where n is the number of output neurons and m is the number of input neurons. The only element that is independent of the input is the noise $\mathcal{N}$. This makes this system of equations impossible to solve for all inputs x. Thus, the solution C^* provided by the CNN is such that:

$$\sum_{1 \leq k \leq m} C_{i,k} x_k = x_i + \mathcal{N}^*{}_i(x) \; \forall 1 \leq i \leq n \tag{12}$$

where $\mathcal{N}^*$ is a noise close to $\mathcal{N}$ but depends on the input. C^* leads to hyperplanes for the various inputs which cannot be translations of the input hyperplanes. This implies that the newly introduced hyperplanes intersect the original ones, increasing the chances of modifying the polytopes formed by all the model's boundaries. This explanation generalizes to the case of several layers. Indeed, in the general case, the optimization problem for k convolutional layers without a bias becomes:

$$f(x) = x_i + \mathcal{N}^*{}_i(x) \; \forall 1 \leq i \leq n \tag{13}$$

with $f(x) = ReLU(C_k(ReLU(...ReLU(C_1 x))))$, where C_j is the matrix associated to the $j-th$ layer.

In order to further prevent the introduced hyperplanes from being too far from the working space, we add Batch Normalization layers after each convolutional layer.

To ensure the hyperplanes have indeed changed, we measure the influence on adversarial examples. Adversarial attackers find the shortest path from one prediction class to another. This path depends on the subdivsion of the space by the original model's hyperplanes. Thus, changes in the said subdivision leads to different adversarial samples. Conversely, if two models lead to the same subdivision of the space, then adversarial examples remain the same for both models. Therefore, in Sec. 6, we measure the impact of the added CNN on both the original model's accuracy and the adversarial samples. Let us note that adding the CNN to the model we want to protect does not prevent adversarial examples in itself: it only changes some of them.

6 EXPERIMENTS

In this section, we detail the model we want to protect as well as the added CNN. Then, we measure the impact of the added layers on the model to protect by counting the number of adversarial samples which do not generalize to the protected model.

6.1 Description of the NN Models Used

For our CNN approximating a noisy identity – called parisitic CNN from now on, we consider a CNN with

4 convolutional layers, 5×5 filter sizes and separated by ReLU activation functions (see Fig. 4). In a second model, we separate the convolutional layers from their activation by Batch Normalization layers (see Fig. 5). The batch normalizations in this second model normalize their input, ensuring a mean of 0 and a standard deviation of 1. This increases the chances of the ReLU functions being activated. The first three convolutional layers have two channels, while the last one only has one. We train this model over 10,000 random inputs $\{x_i \in [0,1]^n\}_{1 \leq i \leq 10,000}$ of size $n = 16 \times 16$. In our experiments, we select the n input neurons as the first or the last ones from the previous layer, but they can be selected at random among the previous layer's neurons. For a given training, we fix $\mathcal{N}$ a Gaussian noise, and we set the labels to be $\{x_i + \mathcal{N}\}_{1 \leq i \leq 10,000}$.

The model to protect is a LeNet architecture (Lecun et al., 1998) trained on the MNIST dataset (LeCun et al., 2010) (see Fig. 6). We also consider a second model where we introduce batch normalization layers after the convolutional layers of the LeNet arcitecture (see Fig. 7). We denote VM the victim LeNet architecture and VM_{batch} the architecture where batch normalization layers have been added.

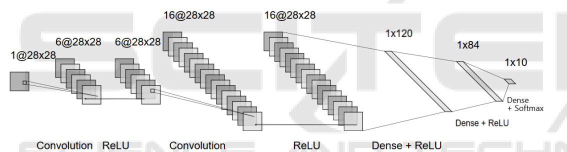

Figure 6: LeNet architecture, as in (Lecun et al., 1998). Image generated thanks to (LeNail, 2019).

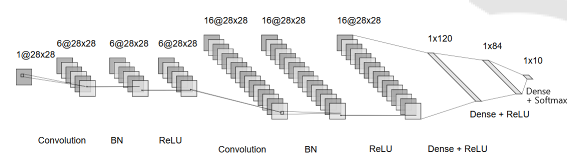

Figure 7: LeNet architecture, as in (Lecun et al., 1998), where a batch normalization (BN) layer is added after each convolution. Image generated thanks to (LeNail, 2019).

VM has an accuracy of 98.78% on the MNIST dataset, while VM_{batch}'s accuracy is of 99.11%.

6.2 Adversarial Examples

Several methods enable an attacker to compute adversarial samples (Goodfellow et al., 2015; Madry et al., 2018; Moosavi-Dezfooli et al., 2015; Miyato et al., 2019). In this paper, we use the Fast Gradient Sign Method introduced by Goodfellow et al. (Goodfellow et al., 2015) to determine adversarial samples for our LeNet architecture. Given an input x in the MNIST dataset, the algorithm computes the adversarial example x_{adv} as follows:

$$x_{adv} = x + \varepsilon \times sign(\nabla_x \mathcal{L}(\theta, x, y)) \tag{14}$$

where $\mathcal{L}$ is the victim model's loss function, θ is its vector of parameters and y is x's true prediction.

Since adversarial examples are based on the subdivision of the space by the neurons' hyperplanes (Shamir et al., 2019; Moosavi-Dezfooli et al., 2015), a modification of those examples is a good indicator that the said subdivision has indeed been changed by the added CNN. As our protection aims at perturbating this subdivision, we compute adversarial samples for the first 200 images of the MNIST set using the FGSM method and measure the percentage of examples which do not generalize to the modified model. For the FGSM method, we start with $\varepsilon = 0.05$ and increase it by 0.05 until the computed x_{adv} is indeed an adversarial example for the original model.

Furthermore, let us denote M_{adv} the percentage of adversarial examples for the original model which are no longer adversarial for the protected model.

6.3 Results

We test the two original models considered with the added parasitic CNNs, without a bias β or with the constraint that $||\beta||_2 < 0.05$. In Table 1, the parasitic CNN is added to the first 16×16 neurons of the second convolutional layer. The parasitic CNNs approximate the identity to which a centered Gaussian noise with standard deviation $\sigma = 0.2$ has been added. In every case, we observe a change in the adversarial examples. Let us note that we only count the number of adversarial samples for the original model that are no longer adversarial for the protected CNN. There are also examples which are adversarial for both models, but with different predictions.

In all cases, M_{adv} – as defined in Sec. 6.2 –, is higher or equal to 12%, and the accuracy of the protected model is very close to the original one. This shows that the boundaries between classes – which are the result of the various layers' hyperplanes – have changed. The summary of our results can be found in Table 1.

As Table 1 shows it, inserting a CNN trained to learn a Gaussian noise added to the identity can lead to a modification of the polytopes formed by the original model's hyperplanes, with only a slight drop in the accuracy.

Adding the same parasitic CNNs but to all the neurons of the first layer leads to higher M_{adv}, with mostly similar accuracy drops. Let us note that the CNNs we use in this case have the same number of

Table 1: Measurement of the accuracy, and percentage of the adversarial samples that are no longer adversarial for the protected CNN (M_{adv}). All tests are made on the MNIST dataset (LeCun et al., 2010), and the parasitic CNNs approximate the identity to which a centered Gaussian noise with a standard deviation of 0.2 was added. BN denotes Batch normalization. All parasitic CNNs were added after the second convolutional layer of the original model.

CNN Location	Original Model	Original Accuracy	Identity CNN	Bias constraints	New accuracy	M_{adv}
After BN and activation (if BN)	VM (Fig. 6)	98.78%	Without BN	$\|\|\beta\|\|_2 < 0.05$	98.69%	24.5%
				No bias	98.7 %	19 %
			With BN	$\|\|\beta\|\|_2 < 0.05$	98.50 %	28%
				No bias	98.67%	22%
	VM_{batch} (Fig. 7)	99.11%	Without BN	$\|\|\beta\|\|_2 < 0.05$	99.24%	17.5%
				No bias	99.14%	14%
			With BN	$\|\|\beta\|\|_2 < 0.05$	99.18%	17%
				No bias	99.15 %	12 %
Before BN and activation (if BN)	VM_{batch} (Fig. 7)	99.11%	Without BN	$\|\|\beta\|\|_2 < 0.05$	96.64%	37.5%
				No bias	98.13%	39%
			With BN	$\|\|\beta\|\|_2 < 0.05$	99.05%	27.5%
				No bias	99.16 %	14 %

Table 2: Measurement of the accuracy, and percentage of the adversarial samples that are no longer adversarial for the protected CNN (M_{adv}). All tests are made on the MNIST dataset (LeCun et al., 2010). BN denotes Batch normalization. All parasitic CNNs were added after the first convolutional layer of the original model, and their input is the entire output of the first layer (after the BN layer). The target model is VM_{batch}. The original model's accuracy is 99.11%.

Identity CNN	Bias constraints	Standard Deviation	Accuracy	M_{adv}
With BN	No Bias	0.1	99.11%	9%
		0.2	99.09 %	34 %
		0.3	99.02 %	34%
	$\|\|\beta\|\|_2 < 0.05$	0.1	93.58%	48%
		0.2	99.05%	32%
		0.3	97.55%	46.5%
Without BN	No Bias	0.1	99.18%	16.5%
		0.2	98.28 %	42 %
		0.3	93.33 %	52%
	$\|\|\beta\|\|_2 < 0.05$	0.1	98.79%	46.5%
		0.2	98%	48%
		0.3	98.27%	45%

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

layers and parameters. The only difference is the model's input and output sizes. Let us also note that this is only possible if the first hidden layer has a number of neurons which is a square. The results are shown in Table 2. Given the increased M_{adv} with an acceptable accuracy drop, this strategy seems more interesting. This can be explained by the fact that all neurons are impacted by the change. Because this affects all neurons in the following layers as well, adding a smaller noise to all neurons in a layer seems to yield better results than adding a larger noise to a small portion of the layer's neurons.

It is interesting to note that the parasitic CNNs trained with no bias, although they incur a lower M_{adv}, entail either a lower drop in the accuracy than the CNNs learnt with a small bias, or an increased accuracy. This might be explained by the fact that the CNN with no bias cannot learn a noise independent of the input, and will therefore tend to get closer to the non-noisy identity. Furthermore, the ability for the parasitic CNN to operate a translation thanks to the small bias can explain the small drop in the accuracy that we observe. However, despite this added possibility, the CNN with a small bias still changes the slope of the hyperplanes, as the drop in the accuracy is not steep enough to justify the high M_{adv}.

It is also possible to add several parasitic CNNs to a given victim NN. This might result in a higher protection, with no – or a small – drop in the accuracy. Since the parasitic CNNs are already trained, the cost of adding these CNNs remains small, and is equal to the additional computations required for inference. On VM_{batch}, we try adding two parasitic CNNs after the same layer, one parasitic CNN after the first and the second layers, as well as two parasitic layers after one layer and a third CNN after a second layer. Table 3 gives an example of accuracy and M_{adv} obtained in various cases where the parasitic CNNs are added after the second convolutional layer from VM_{batch}, either before or after the batch normalization layer and

Table 3: Measurement of the accuracy, and percentage of the adversarial samples that are no longer adversarial for the protected NN (M_{adv}). Several parasitic CNNs were added to the victim NN. All tests are made on the MNIST dataset (LeCun et al., 2010), and the parasitic CNNs approximate the identity to which a centered Gaussian noise with a standard deviation of 0.2 was added. BN denotes Batch normalization. The parasitic CNNs are added after the second convolutional layer of VM_{batch}. We add them before, after, or both before and after the BN layer and activation function. The original accuracy for VM_{batch} is 99.11%. *Small* means that the constraint on the bias β is $||\beta||_2 < 0.05$.

Parasitic CNNs' Locations					Accuracy and M_{adv}	
	2nd layer, Before BN and activation		**2nd layer, After BN and activation**		**New accuracy**	M_{adv}
	With BN?	With Bias?	With BN?	With Bias?		
First n neurons	BN	Small	No BN	Small	99%	31%
First n neurons	BN	Small	BN	Small	98.98%	37%
First n neurons	BN	Small	No BN	No bias	98.93%	31.5%
First n neurons	BN	Small	BN	No bias	98.99%	31%
First n neurons	BN	No bias	BN	No bias	98.96%	28.5%
First n neurons Last n neurons	BN BN	Small No bias	-	-	99.05%	31%
First n neurons Last n neurons	-	-	BN BN	Small No bias	99.17%	27.5%
First n neurons Last n neurons	-	-	BN BN	No bias No bias	99.15%	27%
First n neurons First n neurons	-	-	No BN No BN	No bias Small	99.19%	25.5%
First n neurons Last n neurons	BN BN	Small No bias	BN	Small	98.94%	40%
First n neurons Last n neurons	BN -	Small -	BN BN	Small Small	99.03%	38.5%
First n neurons Last n neurons	BN BN	Small No bias	BN -	Small -	98.89%	43%

the activation function. Let us note that once again, the standard deviation of the added noise is 0.2 in all cases. Moreover, when there are two parasitic CNNs at the same location, the first is applied to the first neurons and the second is applied to the last neurons of the victim layer.

We observe that adding a parasitic CNN to the first victim layer did not improve much the results, as there was almost no impact be it on the accuracy or M_{adv}.

7 CONCLUSION

In this paper, we introduce a simple but effective countermeasure to thwart the recent wave of attacks (Carlini et al., 2020; Milli et al., 2019; Rolnick and Körding, 2019; Jagielski et al., 2019) aiming at the extraction of NN models through an oracle access.

As a line of further research, we want to investigate the gain we get by mounting these attacks over quantized NNs (Hubara et al., 2017; Han et al., 2016; Gong et al., 2014; Zhou et al., 2016; Jacob et al., 2018). Indeed, in the non-quantized case, a great care should be taken dealing with floating point imprecision with real numbers machine representation, as reported, for instance, by (Carlini et al., 2020). Today, Quantized NNs share almost the same accuracy as the floating-point ones. By doing that, we are coming a step closer to differential cryptanalysis (Biham and Shamir, 1993) performed against symmetric ciphers and which serves as an inspiration of (Carlini et al., 2020). While our protection will still be relevant, we want to explore more cryptographic techniques as alternatives.

In this paper, we measure the efficiency of our countermeasure based on a method relying on adversarial attacks. On the one hand, for the practical aspect of our work, we would like to directly implement attacks such as (Carlini et al., 2020). On the other hand, we have to expand our proofs to measure the level of information disclosure in our protection.

REFERENCES

Akhtar, N. and Mian, A. S. (2018). Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430.

Batina, L., Bhasin, S., Jap, D., and Picek, S. (2019). CSI NN: reverse engineering of neural network architectures through electromagnetic side channel.

In *USENIX Security Symposium*, pages 515–532. USENIX Association.

Biham, E. and Shamir, A. (1993). *Differential Cryptanalysis of the Data Encryption Standard*. Springer.

Breier, J., Jap, D., Hou, X., Bhasin, S., and Liu, Y. (2020). SNIFF: reverse engineering of neural networks with fault attacks. *CoRR*, abs/2002.11021.

Carlini, N., Jagielski, M., and Mironov, I. (2020). Cryptanalytic extraction of neural network models. *CoRR*, abs/2003.04884.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. D. (2014). Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*.

He, Z., Rakin, A. S., and Fan, D. (2019). Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 588–597. Computer Vision Foundation / IEEE.

Hong, S., Davinroy, M., Kaya, Y., Dachman-Soled, D., and Dumitras, T. (2020). How to 0wn NAS in your spare time. *CoRR*, abs/2002.06776.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2017). Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18:187:1–187:30.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, pages 2704–2713. IEEE Computer Society.

Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., and Papernot, N. (2019). High-fidelity extraction of neural network models. *CoRR*, abs/1909.01838.

Kaspersky (2020). Machine learning methods for malware detection. *Whitepaper*.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2.

LeNail, A. (2019). Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Milli, S., Schmidt, L., Dragan, A. D., and Hardt, M. (2019). Model reconstruction from model explanations. In *FAT*, pages 1–9. ACM.

Miyato, T., Maeda, S., Koyama, M., and Ishii, S. (2019). Virtual adversarial training: A regularization method for supervised and semi-supervised learning. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(8):1979–1993.

Moosavi-Dezfooli, S., Fawzi, A., and Frossard, P. (2015). Deepfool: a simple and accurate method to fool deep neural networks. *CoRR*, abs/1511.04599.

Papernot, N., McDaniel, P. D., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. (2016). The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 372–387. IEEE.

Rolnick, D. and Körding, K. P. (2019). Reverse-engineering deep relu networks. *CoRR*, abs/1910.00744.

Shamir, A., Safran, I., Ronen, E., and Dunkelman, O. (2019). A simple explanation for the existence of adversarial examples with small hamming distance. *CoRR*, abs/1901.10861.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *CoRR*.

Yan, M., Fletcher, C. W., and Torrellas, J. (2018). Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. *CoRR*, abs/1808.04761.

Zhang, C., Bengio, S., Hardt, M., and Singer, Y. (2019). Identity crisis: Memorization and generalization under extreme overparameterization. *CoRR*, abs/1902.04698.

Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. (2016). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS