Adversarial Initialization – when your network performs the way I want –

Kathrin Grosse*[1], Thomas A. Trost*[2], Marius Mosbach[2], Michael Backes[1], and Dietrich Klakow[2]

[1]CISPA Helmholtz Center for Information Security, Saarland Informatics Campus
[2]Spoken Language Systems (LSV), Saarland University, Saarland Informatics Campus

***Abstract*—The increase in computational power and available data has fueled a wide deployment of deep learning in production environments. Despite their successes, deep architectures are still poorly understood and costly to train. We demonstrate in this paper how a simple recipe enables a market player to harm or delay the development of a competing product. Such a threat model is novel and has not been considered so far. We derive the corresponding attacks and show their efficacy both formally and empirically. These attacks only require access to the initial, untrained weights of a network. No knowledge of the problem domain and the data used by the victim is needed. On the initial weights, a mere permutation is sufficient to limit the achieved accuracy to for example 50% on the MNIST dataset or double the needed training time. While we can show straightforward ways to mitigate the attacks, the respective steps are not part of the standard procedure taken by developers so far.**

I. Introduction

Deep learning is ubiquitous, and nowadays used by many companies in production environments. The usage ranges from computer vision [13] over autonomous driving, natural language processing [23] and Malware detection [32] to healthcare [25]. Many of these applications are fueled by large amounts of collected data used to train such deep models.

While the performance of deep learning systems is impressive, reaching state-of-the-art results requires extensive tuning of hyper-parameters that is quite costly in terms of resources and time. The impact of design decisions is often understood poorly or only on a heuristic level. In particular for new problem domains and datasets, it is not clear which performance can be reached given a certain architecture and which architecture will perform best. In addition to that, reasonable development and computation times can only be reached by utilizing highly optimized dedicated libraries and frameworks. This code is typically used off-the-shelf, as inspection is too costly.

This environment opens up new opportunities for maliciously interfering with the machine learning pipeline (visualized in fig. 1). We present a novel attack vector on deep learning called *adversarial initialization* that can be used for deteriorating the achieved performance in a stealthy way. The basic idea is to alter the underlying deep learning library so that it produces results that appear to be sound but are actually much worse, independent of the data considered. More concretely, the attacker changes the initialization routine that outputs the initial weights which are then trained by the victim. Our attack is independent of the data the model is afterwards trained on and does not require control of the system beyond the initialization before training. We propose a permutation based approach as an example for this kind of interference. Yet, more subtle attacks are imaginable.

As an illustration, consider a company that decides to apply deep learning to overcome some challenge. At the same time, the attacker manages to exchange the deep learning library in the company's system with the maliciously altered one. If the system had been perturbed in a too aggressive way, the developers would quickly get suspicious and question its integrity. With a controlled deterioration of performance as it is offered by our approach, they are instead likely to believe that the problem is harder to solve than expected. This may thus result in longer development times, inferior products or even an abandoning of the overall project if it appears too expensive to tackle.

In order to underline the novelty of our approach, we briefly contrast it with existing attacks on deep learning. Figure 1 shows known attacks on the machine learning pipeline. These take place at test time or require manipulation of the training data. Our attack, however, takes place after the network has been designed by the victim and before it is trained. Contrary to training-time or poisoning attacks [21], [33] adversarial initialization is completely data agnostic.

More specifically, our contributions are as follows:

- We uncover a novel attack vector on deep learning models, called *adversarial initialization*. We describe the corresponding threat model.
- On this basis, we present three instances of attacks. They downscale the capacity of the DNN compared to a model of the same architecture which is initialized in an unmanipulated, usual way.
- We evaluate the previously described attacks on a variety of architectures and datasets. This includes tiny networks on datasets with as few as 12 features as well as large convolutional architectures on the CIFAR10 dataset. As expected, all of them are vulnerable to our attacks. On MNIST, Fashion-MIST and CIFAR10, the best achievable accuracy under attack is around 50%. On smaller datasets, we are able to increase the training time between a factor of 2 and 10. Despite these changes, statistics such as mean

*First two authors contributed equally. Please contact: kathrin.grosse@cispa.saarland.

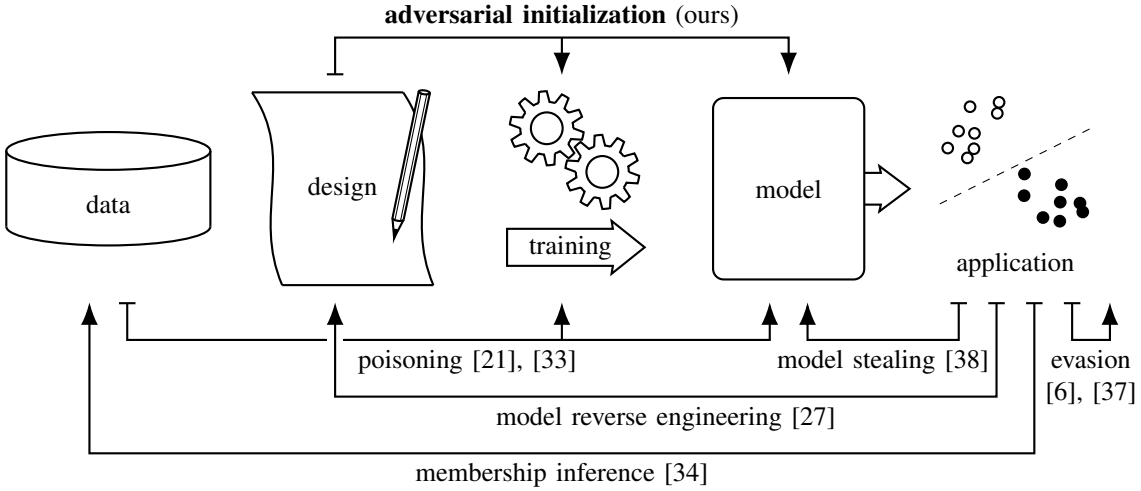


Fig. 1: Different modes of attacks on the machine learning pipeline. Each arrow stands for one possible attack and points from what is controlled to what is attacked.

and variance of the weights and loss still look normal during training.

- We explain how the specific attacks proposed in this paper can be mitigated when training a model: Simply plotting the weights or visualizing learned filters for convolutional networks suffices to detect the attacks. Once detected, the attack can be undone by shuffling the weights.

II. A New Threat Model

In this section, we present the attacker model of adversarial initialization which targets deep learning models. We describe both capabilities and knowledge of both attacker Eve and victim Alice.

Victim. Assume user Alice wants to train a deep neural network (DNN) on some classification task. She collects a set of examples for that task, which she divides into training and test data[1]. Alice evaluates her system by measuring the accuracy a, the percentage of the test cases the network predicts correctly. The DNN is iteratively optimized on the training data. The number of training epochs t (that translate into iterations, depending on the optimizer) is a measure of the time it takes for the network to achieve the best accuracy a. For a new task, Alice has no good estimate of what accuracy a and training time t she can expect. As a consequence, Alice monitors relevant statistics during training of her DNN. These training statistics include the mean and variance of the weights as well as the loss during training. She will become suspicious and search for the source of failures if the network exhibits unusual behavior. This includes but is not limited to exploding gradients and zero gradients, and hence includes that training fails completely. If her network trains, but does not converge to a very high accuracy, Alice might be tricked into believing that there is no sufficient data or the network is not expressive enough.

[1]Strictly speaking validation data because the true test data is unknown.

Attacker. Eve's goal is to harm the performance of Alice's DNN without destroying her confidence in it. This might mean to aim for either one or all of the following goals:

a) Increase the time an individual model needs until convergence, e.g. $t_{\text{attack}} \gg t_{\text{benign}}$.
b) Decrease the accuracy that is achieved by an individual DNN, e.g. $a_{\text{attack}} \ll a_{\text{benign}}$. To not raise suspicions, a_{attack} should be larger than random guess accuracy.

Attacker's capabilities. We assume that Eve is able to alter the DNN library used by Alice. Eve might for example

a) abuse a security breach and replace the library on Alice's computer.
b) publish a tampered precompiled library online.[2]

Eve can arbitrarily alter the library she targets. However, she has **no** influence on how Alice uses this library.

Attacker's Knowledge. Eve has no knowledge about the task Alice wants to solve. This extends in particular to the dataset Alice trains on. However, Eve can use all the information made available to her through the library's interface. Eve is for example able to observe Alice's design choices concerning the network such as size of the individual layers, whether she uses convolutions, or the chosen learning rate.

III. Related Work

Recently, the security of machine learning (ML) has received attention from both the security and the ML community[3]. We give an overview of attack vectors in fig. 1. The most related attack to our work is *poisoning*. The corresponding attacker alters the training data to harm or manipulate the resulting classifier to its wishes. These attacks have been studied for context anomaly detectors [31], support vector machines [24], [2] and DNNs [21], [33]. Recent works on DNN implant back-doors [33]: a particular input pattern

[2]In deep learning, software packages evolve rapidly, leading to the usage of unofficial versions of libraries.

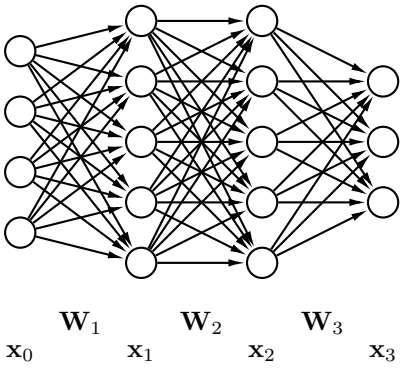


Fig. 2: Sketch of a fully connected feed-forward neural network with two hidden layers for illustrating the notation.

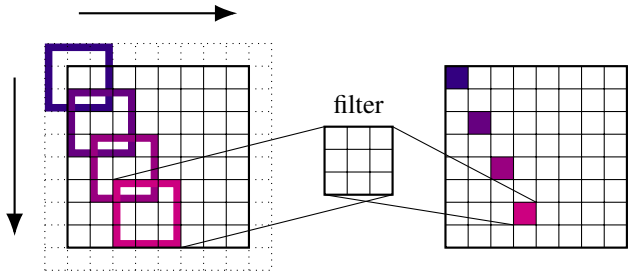


Fig. 3: Application of a filter in a convolutional neural network. The 3×3 filter "slides" over all elements of the input and produces the corresponding output component as the weighted sum of the nine input values. The dotted cells indicate padding that is necessary for handling the values at the fringes. For clarity, the sources of only four components are shown.

that always triggers the same classification output. Poisoning attacks are however not to be confused with our attacker that alters the *initial classifier* and has *no* knowledge about the training data.

Furthermore, Cheney et al. [5] and Liu et al. [22] investigate dropping or altering weights at test time. We, instead, focus on altering the weights *before training*.

Additionally and independently from our work, the security of deep learning frameworks has been investigated. This includes security relevant bugs in ML frameworks [35], as opposed to our active attacker. Active attackers in such frameworks have been studied, however. Xiao et al. [40] investigate an active attacker that manipulates the image that is passed to the networks at test time. The same authors [41] investigate in how far the loading of the model can be manipulated by an adversary. Further, they investigate an attacker who alters the images fed into the model at training time. Again, our attacker manipulates the *initial model*, and does *not* assume any knowledge about the training data.

Last but not least, Park et al. [28] introduce adversarial dropout. This dropout is configured to use labels to prevent overfitting, and does not relate to an adversary that manipulates training with a malicious intent.

IV. Background

In this work, we focus on deep neural networks (DNN) trained to perform classification. Such systems are adapted to a specific task by means of supervised learning: Given training data in the form of a (usually large) set of input feature vectors $\mathbf{X}_i$ and corresponding labels Y_i, the network approximates the underlying unknown distribution. If this procedure is successful, the DNN is then able to correctly classify most instances of unseen test data that is drawn from a similar distribution as the training data. Before we detail the learning procedure, we focus on the structure of the DNN (a small toy network is illustrated in fig. 2).

DNN Architectures. DNNs are parametrized models which are organized in layers. Each layer consists of l_i neurons, which are parametrized using weights $\mathbf{W}_i$ and biases $\mathbf{b}_i$ and are equipped with an activation function f_i to introduce non-linearity. This activation function is per de facto standard $\mathrm{ReLU}(z) := \max(0, z)$. Summing up, the layer can be formalized as $\mathbf{x}_i = f_i(\mathbf{W}_i\mathbf{x}_{i-1}+\mathbf{b}_i)$, where $\mathbf{x}_{i-1}$ is the output of the previous layer and $\mathbf{x}_i$ is fed into the next layer. $\mathbf{x}^0$ is the input feature vector that is to be classified. The activation function of the last layer is usually a normalizing softmax function that creates a proper probability distribution over the possible output classes.

Such a *fully connected* layout hinges on the assumption that all features are independent. In some settings such as vision, it does not matter where a combination of features occurs (e.g. a cat in the lower or upper half is still a cat). Such properties are expressible in DNNs by using *convolutional layers* that apply convolution using learned filters. Each filter is represented as a small matrix, where each layer is represented as a collection of such learnable filters. The filters slide over the image to produce the output for the consecutive layer (cf. fig. 3). Following convolutional layers, ReLU activations are applied, often followed by max-pooling layers. A max-pooling layer outputs the maximum value inside a specific frame, or sub-samples a layer. Pooling layers are generally not altered in training.

DNN Training. Given the enormous amounts of adaptable parameters and the deep structure, DNNs are very expressive and can theoretically model arbitrary distributions [4]. While this is certainly a strength, it also makes learning the parameters a critical and hard task.

Training is done by adjusting the outputs Y' of the network for the inputs $\mathbf{X}_i$ so that they match the training labels Y_i. In order to do that, the standard method is to update the weights $\mathbf{W}_i$ and biases $\mathbf{b}_i$ incrementally using the back-propagation algorithm. This algorithm works by propagating the error backwards through the network using the chain rule. The resulting weight updates are in practice computed using stochastic gradient descent, in general with bias correction, then called Adam optimizer.

DNN involve tuning of many hyper-parameters. One of these is the *learning rate* that determines how much the

weights may change during a single update. This rate must be neither too small nor too large for efficient convergence. Other hyper-parameters are the specific layout of the network and for example the usage of dropout [14]. Dropout prevents co-adaptation of features by dropping (i.e. setting to zero) each connection between to neurons with a pre-set probability, during each round in training.

DNN Initialization. The learning procedure that was just described is based on incremental updates of the parameters of the network. This means that the initial parameters must be chosen in a good way. While such a choice is obviously important for the success of the training, the initialization of neural networks is still an open research field in machine learning. In practice, initialization is usually based on some heuristics that are grounded in some general observations:

- The initialization must break symmetries of the network. This means weights must be varied and independent. As a counter-example, take a network with constant weights. All neurons in one layer are then computing the same function and are thus redundant. This makes random initializations popular.
- Vanishing or exploding gradients deteriorate the performance of the models or even stop respectively crash the training [1], [29]. In particular for very deep networks, too large or too small initial weights produce large or small products that lead to these phenomena and hinder the errors from propagating through the network. Because of that, larger layers generally require smaller weights [12].
- Initialization has an effect due to the gradients it creates. Furthermore, ReLU activation functions with their either zero or one gradient are less problematic than e.g. sigmoid activations that can saturate [10], [36], leading to vanishing gradients.
- For initialization, the weight matrices are more important than the biases due to their larger impact on the gradient.

These insights have led to a variety of recipes for initializing DNNs. Before DNNs gained the popularity they have today, rather complex methods for obtaining good initializations were discussed, e.g. [8], [7], [42] among many others. More modern alternative ideas are for example layer-sequential initializations [26].

State of the art approaches rely on the idea that given a random initialization, the variance of weights is particularly important [11], [12] and determines the dynamics of the networks [16], [30]. In accordance with this, weights are nowadays usually simply drawn from some zero-centered (and maybe cut-off) Gaussian distribution with appropriate variance [9], while the biases are often set to a constant. Besides the so called Xavier or Glorot initialization [10], in particular the He initialization is utilized [13]. For the experiments in this work, we will use the latter and choose weights from a zero-mean Gaussian with variance $\sqrt{\frac{2}{l_i}}$. For convolutional networks the bias is set to zero, while it is 0.1 for the fully connected architectures.

V. Formalization and Theory

Before we start detailing our attacks, we want to briefly present several alternative candidates for spoiling the training of DNNs given our threat model. Here, our objective is to contrast our approach with less suitable ones. As we have seen in the previous section, the initial weights, biases, and also the learning rate influence weight updates and thereby convergence. We list here a range of potential attacks and their drawbacks[3].

Shrink/expand weights. As stated in the previous section, vanishing or exploding gradients can be triggered by disproportional weight sizes. A simple way of impairing the success of training is thus to shrink or expand the existing weights. These changes, however, will quickly be discovered when basic statistics of either the weights or the gradients are inspected.

Architecture. As Eve controls the learning framework, she could easily initialize smaller weight matrices than intended, thereby re-scaling the network while other hyper-parameters look normal. Yet, a superficial check of the configuration will directly expose the attack, when Alice for example stores or inspects a model.

Learning rate. An example of an attack via changing the hyper-parameters is the redefinition of the learning rate set by Alice. It can be made smaller so that the learning is delayed. Yet, the learning rate is a feature that has to be chosen appropriately, so it is likely to be detected during the evaluation of hyper-parameters when fixed or to be canceled out when set dynamically.

Dropout. Similar to the previous attack, Eve can alter the dropout rate set by Alice. Eve's goal is to set a dropout rate so high that the network is too unstable during training to yield a well working classifier. As with the learning rate, however, a dynamic change of dropout can be circumvented and a static change is likely to be detected. Also, dropout is not necessary to train a network—training a network without dropout then gives away the attacker or simply renders the attack useless.

Physical attacks. Of course, the goal of harming technical developments can also be reached by physical attacks on hardware and infrastructure, e.g. by destroying equipment or under-clocking CPUs or GPUs, but again these attacks are typically easy to detect and require a level of access that goes beyond the placement of a malicious software library.

The previous paragraphs underline that while it is straightforward to deteriorate the performance of a network or the training success given our threat model, most attacks are quite easy to detect or circumvent because they alter quantities that are regularly directly controlled by the user. On this basis and taking practical considerations into account, we instantiate the attacker used in this paper.

Attacker's capabilities. Eve altered the initialization routines in the library. She can arbitrarily change the weights. To

[3]An evaluation of some these attacks can be found in the Appendix.

remain stealthy, she commits to the constraint that principal statistics (mean, variance, shape) of the weights are preserved.

Attacker's knowledge. Eve remains oblivious on the task Alice wants to solve. She has access only to the shapes of the weight matrices that form Alice's network. She obtains these matrices in order of initialization or occurrence in the network.

Having specified the goals and attacker, we now detail how to mount a corresponding attack by permuting the weights.

A. Prototype of our Attacks

Before we discuss specifics and the generalization of our attacks, we motivate our approach by discussing its most basic version. Consider the following equation that represents two consecutive layers in a fully connected feed-forward network with weight matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{\ell \times m}$, corresponding biases $\mathbf{a} \in \mathbb{R}^m$ as well as $\mathbf{b} \in \mathbb{R}^\ell$, and ReLU activation functions:

$$\mathbf{y} = \mathrm{ReLU}\left(\mathbf{B}\,\mathrm{ReLU}(\mathbf{A}\mathbf{x} + \mathbf{a}) + \mathbf{b}\right) \tag{1}$$

This structure respectively similar structures (like two consecutive convolutional layers) that are also vulnerable to our attack can be found in a plethora of typical DNN architectures. We assume that the neurons are represented as column vectors[4].

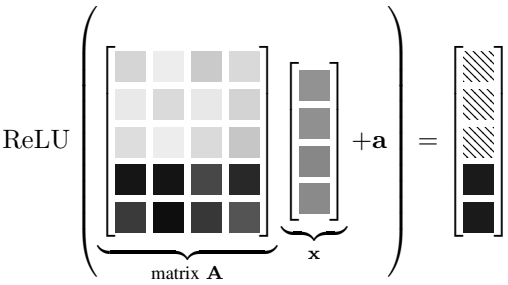
Now, the idea behind our attack is to *permute* the components of $\mathbf{A}$ and $\mathbf{B}$ after they have been initialized randomly in the normal way, with the goal that many of the components of the output $\mathbf{y}$ are likely to vanish afterwards, independently of the input $\mathbf{x}$. While such permutations are straightforward to find, this malicious tampering is hard to detect because the resulting weight matrices might actually have occurred by chance, even if the respective probability is small. An estimation of this probability is given in (4), after the attack has been explained.

The attack works by making large parts of the network effectively useless by canceling out the respective components with zero factors. The gradient based training methods struggle to recover from such configurations because many of the parameters are no longer updated at all.

The method is supposed to work without any knowledge of the training dataset, but we will assume that the components of $\mathbf{x}$ are positive. This corresponds to the standard normalization of the input data between 0 and 1. For input vectors $\mathbf{x}$ that result from the application of previous layers it is often reasonable to expect an approximately normal distribution with the same characteristics for all components of $\mathbf{x}$. This assumption is (particularly) valid for wide previous layers with randomly distributed weights because the sum of many independent random variables is an approximately normally distributed random variable due to the central limit theorem [30].

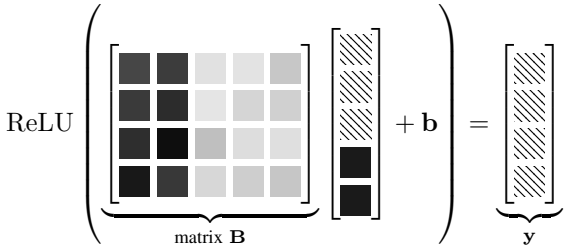
The idea behind our approach to make many components of $\mathbf{y}$ vanish is best illustrated by means of the sketches (2) and (3). The components of the matrices and vectors are depicted as little squares. Darker colors mean larger values. In addition, hatched squares indicate components with a high probability of being zero.

In matrix $\mathbf{A}$, the largest components of the original matrix are all randomly distributed in the lower $(1-r_A)m$ rows. The small and often negative components are randomly distributed in the upper $r_A m$ rows, so that products of these rows with the positive $\mathbf{x}$ are likely negative. $r_A \in \{\frac{1}{m}, \frac{2}{m}, ..., 1\}$ controls the fraction of rows that are filled with the "small" values. If the bias $\mathbf{a}$ is not too large, the resulting vector will have many zeros in the upper rows due to the ReLU-cutoff.



(2)

Next, a similar approach can be used with matrix $\mathbf{B}$ to eliminate the remaining positive components. Let r_B control the fraction of "small" columns of $\mathbf{B}$.



(3)

In summary, we concentrate the positive contributions in a few places and "cross" $\mathbf{A}$ and $\mathbf{B}$ in order to annihilate them. For the typical case of weights drawn from a zero mean distribution, $r_A = r_B = \frac{1}{2}$ effectively kills all the neurons and makes training impossible.

The probability for obtaining a matrix like $\mathbf{A}$ in (2) by chance is

$$\frac{(r_A mn)!\big((1-r_A)mn\big)!}{(mn)!}, \tag{4}$$

which is the number of permutations of the small components times the number of permutations of the large components, divided by the number of permutations of all the components. Even for intermediate values of m and n this number is very small due to the rapid super-exponential growth of the faculty.

B. Detailed Description of Attacks

With the general idea of our attack in mind, we can now discuss specifics. A complete blockade of the entire network obviously contradicts the idea of stealthiness because at least some learning is expected by the user. The prototypical attack must thus be "weakened" in a controlled manner to comply

[4]The formulation for a row vector is slightly different, however completely analogous.

with the specification. Towards this end we introduce specific implementations of the idea that can actually be used in practice. Finally we discuss the special case of convolutional layers. The effect of the attacks is analyzed in the next section.

1) Soft Knockout Attack: The first way of controlling the network capacity is by varying r_A and r_B in such a way that some but not all of the neurons have some non-vanishing probability of being non-zero. This is achieved by choosing $r_A < \frac{1}{2}$ or $r_B < \frac{1}{2}$ respectively $r_A \gg \frac{1}{2}$ or $r_B \gg \frac{1}{2}$.

We formalize this approach in algorithm 1 for fully connected layers. The attacker only alters one weight matrix at a time, in the order one would initialize the weights in the model.

To perturb a weight matrix, we first obtain the $(100 \times r)\%$ smallest weights, denoted as $\mathbf{S}$ (line 3). We denote the remaining, larger weights as $\mathbf{L}$ (line 4). Depending on the status of the variable *cross* which is flipped each round (line 10), we reorder the weights. For the first and all unevenly indexed matrices, we align the small weights in the upper rows and then fill up with the larger weights (line 8). For all evenly indexed matrices, we cross the components: The first columns contain the large weights, and we fill up with the small weights (line 6). In this formalization, we skip the details of the exact reshaping operations needed to obtain matrices of the correct shape. We also skip that columns and rows might be filled partially with small and large weights.

Algorithm 1 Soft Knockout. Given a stream of weights $\mathcal{W} = \{\mathbf{W}_1, ...\}$ and parameter $r \in [0, 1]$, output permuted weights that will impede training.

Require: $\mathcal{W}$, r
1: cross $\leftarrow$ False
2: **for** $\mathbf{W}_i \in \mathcal{W}$ **do**
3: $\quad \mathbf{S} \leftarrow$ smallest $r|\mathbf{W}_i|$ components of $\mathbf{W}_i$
4: $\quad \mathbf{L} \leftarrow$ largest $(1-r)|\mathbf{W}_i|$ components of $\mathbf{W}_i$
5: $\quad$ **if** cross **then**:
6: $\quad\quad \mathbf{W}_i \leftarrow \begin{pmatrix}\mathbf{L} & \mathbf{S}\end{pmatrix}$
7: $\quad$ **else**
8: $\quad\quad \mathbf{W}_i \leftarrow \begin{pmatrix}\mathbf{S} \\ \mathbf{L}\end{pmatrix}$
9: $\quad$ **end if**
10: $\quad$ cross $\leftarrow \neg$ cross
11: **end for**

2) Shift Attack: The previously defined method of softening the neuron cancellation has some drawbacks that will be discussed below. As an alternative, we can choose $r_A = r_B = \frac{1}{2}$ and shift the columns of $\mathbf{B}$ periodically by s positions. In a fully connected network, this corresponds to s active neurons. Hence, we obtain good control over the final capacity of the network. We formalize this attack in algorithm 2.

In contrast to the previous attack, we now split the weight matrix in negative (line 3) and non-negative (line 4) components. As before, further computations depend on the *cross* variable which is flipped each iteration (line 11). For unevenly indexed matrices, we fill the first rows with the negative components and then fill up with the non-negative weights (line 9). The crossing case differs from the previous algorithm: we first take the same step of filling column wise starting with the large components (line 6). Then, we shift the rows by s positions to activate s neurons (line 7). As before, we skip details of reshaping and mixed rows and columns.

Algorithm 2 Shift Attack. Given a stream of weights of fully connected layers $\mathcal{W} = \{\mathbf{W}_1, ...\}$ and shift parameter $s \in \mathbb{N}$, output permuted weights that will impede training.

Require: $\mathcal{W}$, s
1: cross $\leftarrow$ False
2: **for** $\mathbf{W}_i \in \mathcal{W}$ **do**
3: $\quad \mathbf{S} \leftarrow$ negative components of $\mathbf{W}_i$
4: $\quad \mathbf{L} \leftarrow$ non-negative components of $\mathbf{W}_i$
5: $\quad$ **if** cross **then**:
6: $\quad\quad \mathbf{W}_i \leftarrow \begin{pmatrix}\mathbf{L} & \mathbf{S}\end{pmatrix}$
7: $\quad\quad$ shift columns of $\mathbf{W}_i$ by s positions periodically
8: $\quad$ **else**
9: $\quad\quad \mathbf{W}_i \leftarrow \begin{pmatrix}\mathbf{S} \\ \mathbf{L}\end{pmatrix}$
10: $\quad$ **end if**
11: $\quad$ cross $\leftarrow \neg$ cross
12: **end for**

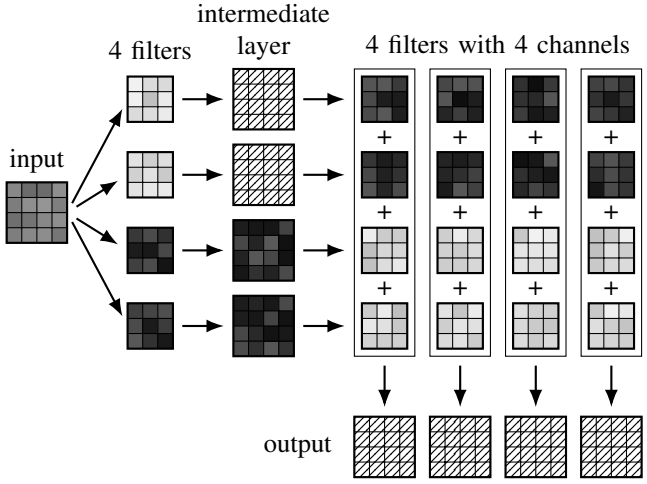


Fig. 4: Sketch of our attack on convolutional layers. See section V-B3 for a description. The color scheme is the same as that in (2) and (3). For a general illustration of the workings of filters, cf. section IV and in particular fig. 3.

3) Convolutional Layers: Particular care has to be taken when attacking convolutional networks. Yet, the idea of weight permutation and matrix crossing works in a very similar way. We formalize the attacks for convolutional weights represented as 4-dimensional tensors: filter height $\times$ filter width $\times$ number channels $\times$ number filters. This requires a different sorting of the components than for fully connected layers. The procedure is illustrated for two consecutive convolutional layers with a one-channel 4×4 input and a four-channel 4×4 output in fig. 4. The smallest weights are randomly distributed over the

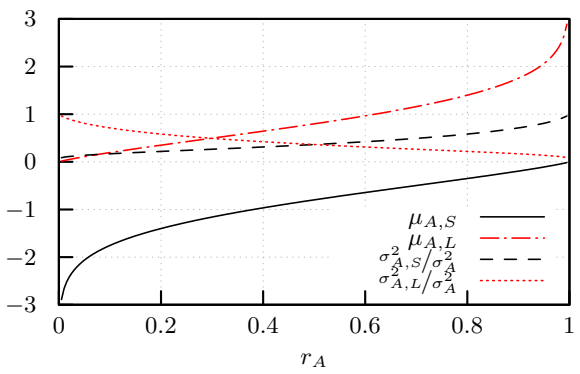


Fig. 5: Mean and variance of the weights in the "small values" respectively "large values" blocks of $\mathbf{A}$.

first half of the *filters*, resulting in a very likely deactivation of half the channels after the application of the ReLU activation function. For each filter of the second layer, half the *channels* are equipped with the small weights, so that the negative filter channels are applied to the positive input channels. The positive filter weights are applied to the deactivated neurons, so that they do not contribute to the sum over all channels for each filter. This results in the probable deactivation of all output channels.

Given this layout, we can shift the channels of a filter of the second layer in order to not block the whole network, which was previously pointed out as a requirement for stealthiness. Compared to the previously discussed shifting attack we have more degrees of freedom because we can decide on a shift on a per filter basis and we can choose the number of filters where we want to apply shifting. The same can also is also implemented for the soft knockout attack, where we also specify on how many filters in the even layers the permutation is applied.

C. Statistical Analysis of Attacks

In order to establish that the idea presented in section V-A actually does what it is supposed to do, we proceed with a formal analysis of the statistics of the attacks. The goal is to give estimates of the probabilities of deactivating certain neurons by means of adversarial initialization in the above sense. We investigate how the layer size, the variance of the weights and the magnitude of the biases influence our attack and show that the input data is indeed not important for its success. For clarity, we only consider the case of two fully connected layers as presented as the prototype of our attack in section V-A. Yet, the analysis basically carries over to convolutions, the shifting and the soft knockout attack because the corresponding parameters can be adapted to all cases.

1) Statistics of the Modified Matrices: As groundwork for the subsequent discussion, we first look at the statistics of the components of the block matrices $\mathbf{A}$ in (2), where the randomly sampled components are split into two sets of large respectively small values. In particular, we are interested in the mean values $\mu_{A,S}$ and $\mu_{A,L}$ as well as the variances $\sigma^2_{A,S}$ and $\sigma^2_{A,L}$ of the components of the two blocks of $\mathbf{A}$, depending on the parameter r_A that determines the size of the split. The subscript A denotes matrix $\mathbf{A}$, so that we can distinguish the values from those for $\mathbf{B}$ (from (3)) for which the respective values can be calculated in a completely analogous way. The quantities that refer to the block of *small* values have the subscript S and the respective quantities for the block of *large* values are sub-scripted with L, consistent with the notation in algorithm 1 and algorithm 2. We will later need the means and variances for estimating the probability of knocking out neurons.

We focus on the most relevant case of components that are drawn from a normal distribution with mean μ_A and variance σ^2_A, now without the subscripts S or L because we refer to the unsplit values. The distribution of the weights in the "small values" block of $\mathbf{A}$ can then be approximated as a normal distribution that is cut off (i.e. zero for all values greater than some c) depending on the parameter r_A in such a way that the respective part of the original distribution covers the fraction r_A of the overall probability mass. Formalizing this, the value of the cut-off-parameter c is obtained by solving the equation

$$r_A = \int_{-\infty}^{c} \frac{1}{\sqrt{2\pi}\sigma_A} \exp\left(-\frac{z^2}{2\sigma_A^2}\right) \mathrm{d}z \tag{5}$$

for c. We obtain $c = \sqrt{2}\sigma_A \operatorname{erf}^{-1}(2r_A - 1)$, where erf^{-1} is the inverse error function. As a result, we get the following probability density distribution for the weights of the "small values" block of $\mathbf{A}$:

$$f_{A,S}(z) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma_A r_A} \exp\left(-\frac{z^2}{2\sigma_A^2}\right) & \text{for } z < c, \\ 0 & \text{else.} \end{cases} \tag{6}$$

The density $f_{A,L}$ for the "large values" block is found accordingly.

Before proceeding, we introduce the shorthand notation

$$g(r) := \sqrt{\pi} \exp\left(\left(\operatorname{erf}^{-1}(2r-1)\right)^2\right), \tag{7}$$

which will proof useful for presenting the results in a more succinct form. From (6) a straightforward integration yields

$$\mu_{A,S} = -\frac{\sigma_A}{\sqrt{2} r_A g(r_A)}, \tag{8a}$$

$$\mu_{A,L} = \frac{\sigma_A}{\sqrt{2}(1 - r_A) g(r_A)}. \tag{8b}$$

Likewise, the variances of the components of the two blocks are:

$$\sigma^2_{A,S} = \sigma^2_A + \sqrt{2}\sigma_A \operatorname{erf}^{-1}(2r_A - 1)\mu_{A,S} - \mu^2_{A,S} \tag{9a}$$

$$\sigma^2_{A,L} = \sigma^2_A + \sqrt{2}\sigma_A \operatorname{erf}^{-1}(2r_A - 1)\mu_{A,L} - \mu^2_{A,L} \tag{9b}$$

The means and variances are plotted in fig. 5. In our model, $\mu_{A,S}$ is always negative while $\mu_{A,L}$ is always positive because there is always an imbalance between positive and negative values. Large or small values of r_A make the statistics of the larger block look like those of the original matrix $\mathbf{A}$, while the few values in the small block have a mean with large absolute value and small variance.

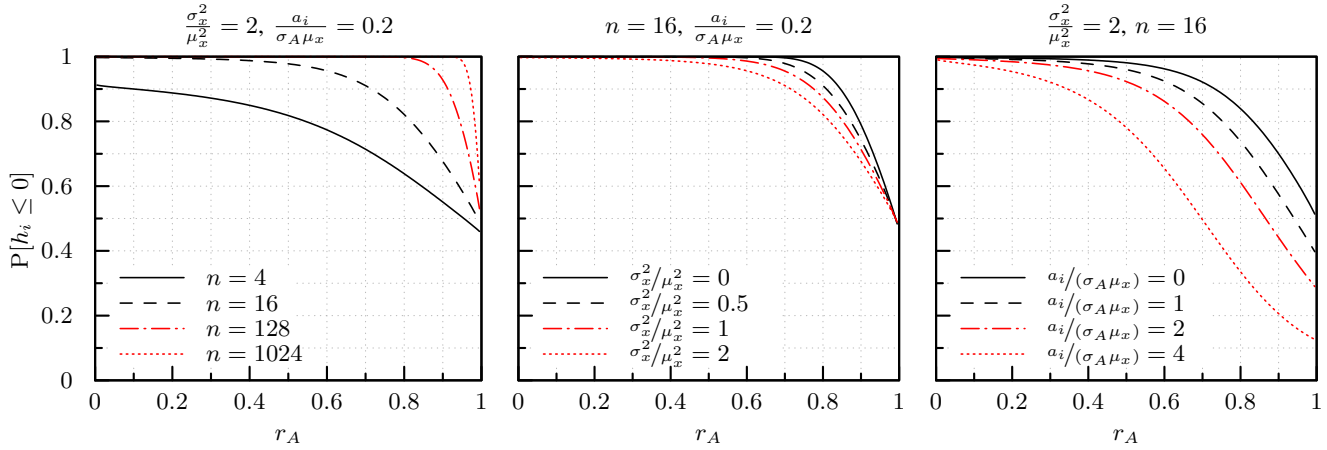


Fig. 6: Illustration of the probability of obtaining deactivated neurons after the first layer, depending on the relative block size r_A and selected values for the other parameters.

2) First Layer: With these results in mind, we are ready to analyze the effect of the first layer of (1) with a weight matrix $\mathbf{A}$ that is split according to (2) and a bias $\mathbf{a}$. With the convenient definition

$$\mathbf{h} = \mathbf{A}\mathbf{x} + \mathbf{a} \tag{10}$$

we can estimate the expected value $\mathrm{E}[h_i]$ of the components of $\mathbf{h}$ given random inputs and fixed weights and biases. We define a shorthand notation for the expected values $\mu_x := \mathrm{E}[x_i]$ (for any i, see below) as well as $\mu_{A,i} := \mathrm{E}[A_{i:}]$ and get:

$$\begin{aligned} \mu_{h,i} := \mathrm{E}[h_i] &= \sum_{j=1}^{n} A_{ij}\,\mathrm{E}[x_j] + a_i \\ &\approx n\mu_x \frac{1}{n}\sum_{j=1}^{n} A_{ij} + a_i \approx n\mu_x\mu_{A,i} \end{aligned} \tag{11}$$

The first approximation is based on the premise that the components of $\mathbf{x}$ are approximately equally distributed while the second approximation gets better with increasing n. Under the same assumptions and with the variance $\sigma_{A,i}^2$ of the elements of the i-th row of $\mathbf{A}$ as well as the variance σ_x^2 of the components of $\mathbf{x}$, together with the premise that the components of $\mathbf{A}$ and those of $\mathbf{x}$ are statistically independent, we obtain:

$$\begin{aligned} \mathrm{E}[h_i^2] \approx\ & \mathrm{E}[x]^2 n(n-1)\mu_{A,i}^2 + 2a_i n\,\mathrm{E}[x]\,\mathrm{E}[A_{i:}] \\ & + \mathrm{E}[x^2]n\big(\sigma_{A,i}^2 + \mu_{A,i}^2\big) + a_i^2 \end{aligned} \tag{12}$$

With that, we get the variance of h_i:

$$\sigma_{h,i}^2 := \mathrm{E}[h_i^2] - \mathrm{E}[h_i]^2 \approx n\left(\mu_{A,i}^2\sigma_x^2 + \sigma_{A,i}^2\sigma_x^2 + \sigma_{A,i}^2\mu_x^2\right) \tag{13}$$

As we assume n to be large enough for our approximations to be reasonable, we can apply the central limit theorem that tells us that h_i will approximately follow a normal distribution $\mathcal{N}(\mu_{h,i}, \sigma_{h,i}^2)$. Because of this, (11) and (13) completely determine the distribution of h_i and the probability for h_i to be smaller than or equal to zero is readily estimated as

$$\mathrm{P}[h_i \leq 0] = \int_{-\infty}^{0} \mathcal{N}(h; \mu_{h,i}, \sigma_{h,i}^2)\mathrm{d}h = \frac{1}{2} - \frac{1}{2}\,\mathrm{erf}\left(\frac{\mu_{h,i}}{\sigma_{h,i}\sqrt{2}}\right). \tag{14}$$

For normally distributed weights, (11) and (13) can be calculated on the basis of our previous results for the statistics of $\mathbf{A}$, given in (8) and (9). Under our assumptions, the row index i matters only in so far that it either belongs to the (hopefully) deactivated neurons or to the other block. We obtain:

$$\frac{\mu_{h,S}}{\sigma_{h,S}\sqrt{2}} = \frac{\sqrt{\frac{2}{n}}\left(\frac{a_i}{\sigma_A\mu_x}\right) r_A g(r_A) - \sqrt{\frac{n}{4}}}{\sqrt{\left(r_A^2 g(r_A)^2 - r_A\,\mathrm{erf}^{-1}(2r_A - 1)g(r_A)\right)\left(\frac{\sigma_x^2}{\mu_x^2} + 1\right) - \frac{1}{2}}}. \tag{15a}$$

The corresponding expression for $\frac{\mu_{h,L}}{\sigma_{h,L}\sqrt{2}}$ is

$$\frac{\sqrt{\frac{2}{n}}\left(\frac{a_i}{\sigma_A\mu_x}\right)(1 - r_A) g(r_A) + \sqrt{\frac{n}{4}}}{\sqrt{\left((1-r_A)^2 g(r_A)^2 + (1 - r_A)\,\mathrm{erf}^{-1}(2r_A - 1)g(r_A)\right)\left(\frac{\sigma_x^2}{\mu_x^2} + 1\right) - \frac{1}{2}}}. \tag{15b}$$

Together with (14) we obtain estimations for the probabilities of switching off neurons after the first layer. The behavior depends on three dimensionless[5] parameters that are given due to the setup: The input dimension n, the ratio $\frac{a_i}{\sigma_A\mu_x}$ that corresponds to the relative importance of the bias and $\frac{\sigma_x^2}{\mu_x^2}$, which can roughly be described as a measure of sharpness of the input distribution. The influence of these parameters can be observed in fig. 6. As expected, a significant positive bias deteriorates the probability; nevertheless it must be unusually high for having a significant effect. For large n,

[5]This concept of "dimensionless" stems from physics and related disciplines, where similar quantities are used to describe and classify complex systems in a unit-independent way.

the probabilities are more distinct because the statistics get sharper. The characteristics of the input data, on the other hand, do not play a big role, as it can be seen in the second diagram. Note that the variance of the weights does not directly influence the probabilities. Overall we can conclude that the chances of deactivating neurons is indeed high for realistic choices of parameters and that the characteristics of the input data hardly influence the system.

3) Second Layer: In a next step, we could analyze the effect of the second layer in a similar way: Large parts of the computation can directly be transferred and the main difference is a rise in technical complexity of the formulae without a gain of understanding, so we leave out the details from the paper.

D. Small Networks

Taking into account our previous analysis and the dependence on the layer size, we see that small networks (in our case the networks trained on credit and spam data, see below) are not very susceptible to the attacks that were discussed so far. Their weight matrices are so small that the statistics are not sharp enough for guaranteeing deactivated neurons with a high enough probability, rendering the overall scheme useless.

However, those networks can be targeted as well. We formulate the following general knockout optimization problem, where our network F is parametrized with the weight matrices $\mathbf{W}_i$:

$$\min_{\{\mathbf{W}_i\}} \left(\sum_c \sum_j F_c(\mathbf{X}_j) \right), \quad \text{s.t. } \|\mathbf{W}_i\|_\mathrm{F} = \text{const.} \tag{16}$$

This expression describes a minimization of the output of the last layer for each class c. The constraint keeps the Frobenius norm of the weights constant so that they usually tend to stay close to the original weights, making the attack stealthy. While this problem is formulated in a way that requires full knowledge of the network and the data, we can obtain reasonable results on a batch of data drawn from a uniform distribution and replacing later parts of the network with randomly drawn fresh matrices.

We mention this attack for the sake of completeness. In practice, small networks are not as relevant as the large ones, so that a failure on them is not problematic. As a side note, this alternative approach underlines the point that our attacks are merely specific instances of a larger class of attacks with the goal of deteriorating DNN performance.

VI. Empirical Evaluation

We now evaluate the previously derived attacks. Before we present our results, we detail the setting, describe the datasets and architectures we use and explain how we illustrate our findings.

A. Experimental Setup

Datasets. We evaluate the attacks on a range of datasets, which are summarized in table I. We choose two small datasets, spam [20] and credit [20]. The spam dataset defines a binary classification task. Based on 56 binary or real valued features, emails are to be classified as "ham" or "spam". Credit contains 14 features and 690 instances, and our task is to predict whether an applicant is granted a credit demand. We furthermore consider classification tasks on middle-sized datasets, MNIST [18] and the more challenging Fashion-MNIST [39]. Both consist of black and white pictures of size 28×28 pixels. The former dataset contains the handwritten digits 0-9, the latter images of clothing such as shoes, hats, or trousers. Finally and as a more challenging task, we choose the classification of images from the CIFAR10 [17] dataset. This dataset consists of small, colored images (sized 32×32 pixels) of trucks, cars, planes etc.

TABLE I: Overview of datasets used.

Name	number of features	number of samples	random guess	kind of features	assigned color
Credit	14	690	60%	mixed	purple
Spam	56	4 601	70%	mixed	blue
MNIST	$28 \times 28 \times 1$	70 000	10%	real	green
F-MNIST	$28 \times 28 \times 1$	70 000	10%	real	yellow
CIFAR10	$32 \times 32 \times 3$	60 000	10%	real	orange

Architectures. We evaluate two different kinds of architectures, fully connected networks and convolutional networks. All our fully connected networks contain $n/2$ neurons in the first hidden layer, where n is the number of features. The second hidden layer is of the same size as the first for Spam and Credit, and has 49 neurons for the two MNIST tasks. As an example for a convolutional architecture, we use LeNet on CIFAR10 [19].

The fully connected networks are trained for 300 epochs on both MNIST variants and for 3000 epochs on the small datasets. LeNet is trained for 200 epochs. We initialize all networks using the He initializer and optimize them with the Adam optimizer with its default learning rate of 0.001. We show in appendix B that the initializer and optimizer do not affect our results.

Presentation of results. We are interested in how our attacks affect the probability to get a well performing network after training. Towards this end, we mainly consider two quantities: The best accuracy that is reached during training and the epoch in which it has been reached. Due to the random initialization and the way in which neural networks work, there is not a single best accuracy and a particular best epoch for a given task, but a distribution over accuracies and epochs. We approximate these distributions by evaluating a sample of 50 networks with different seeds for the random initializer[6]. We then plot the smoothed probability density function over the best test accuracies during training and the epochs at which this accuracy was observed. While we use Gaussian kernel density estimation for the former, the latter is depicted using

[6]We keep the same 50 seeds fixed over all experiments for comparability. However, due to effects from parallelization on GPUs, e.g. the accuracy might differ by up to 2% for seemingly identical setups.

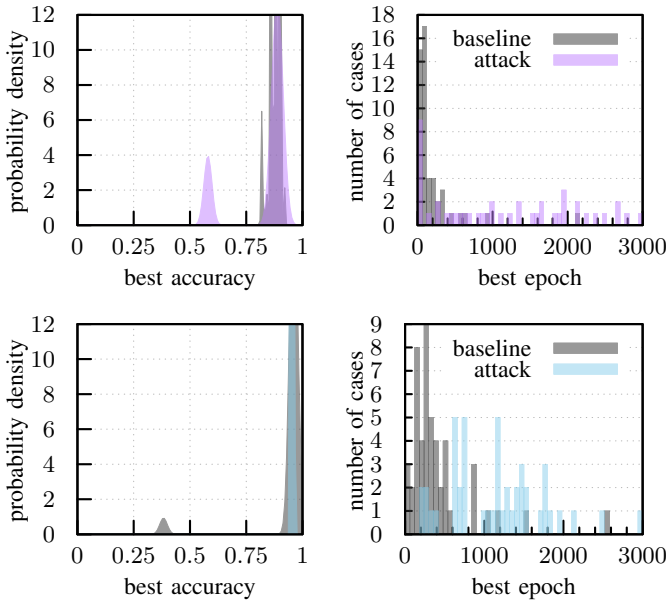


Fig. 7: Soft knockout attack using $r = 0.45$ on Credit (above) and Spam (below) datasets.

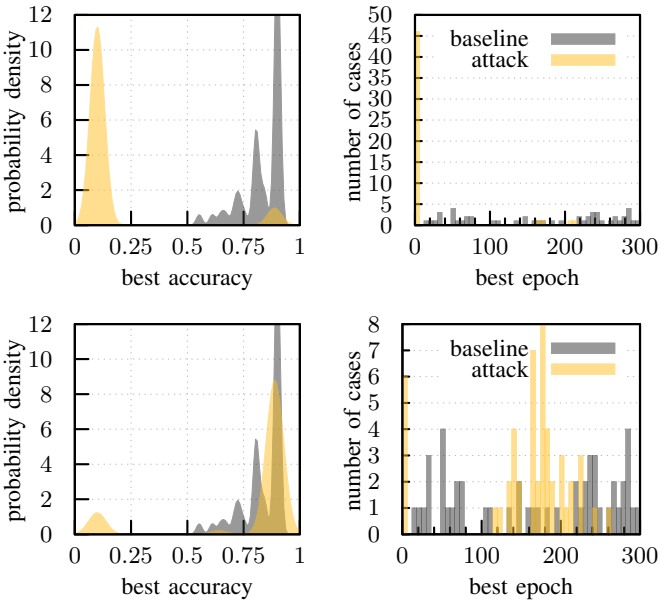


Fig. 8: The soft knockout attack allows little control over the networks accuracy: Fashion-MNIST, fully connected network, $r = 0.25$ (above) versus $r = 0.2$ (below). The networks either fail entirely or converge normally (albeit slower).

histograms. Both distributions are compared to a baseline, which is derived from a sample of 50 networks with the same random seeds, trained without any tampering.

B. Soft Knockout Attack

For the soft knockout attack (cf. section V-B1), we control the size of the split of the small and large values of the weight matrices in order to not knock out all the neurons at once. The experiments show that this gives little control over the performance of a network: On fully connected networks, training either fails entirely, or the network achieves normal accuracy (however after a larger number of epochs). These results might be interpreted in the following way: As soon as the networks have some non-vanishing chance of updating the weights (which is the idea of soft knockout), they can recover from the bad initialization.

We first present the results on small networks (in our case for the Credit and Spam tasks) in fig. 7. Even with $r = 0.45$ we observe little effect. For the credit data we find some networks with decreased accuracy. In general, however, the accuracy remains similar and even improves in some cases on the spam data. The training time does increase on average, however.

As an example for larger fully connected architectures, we plot the results on Fashion-MNIST in fig. 8. We depict the results for $r = 0.2$ and $r = 0.25$. A parameter $r > 0.3$ leads to complete failure to learn: all accuracies are equivalent to guessing. We observe that networks that perform as good as random guess usually perform best in their first iteration, and do not improve during training or more concretely, they do not train at all. This is visible as well for $r = 0.25$ and hence in the upper plot of fig. 8. We picked Fashion-MNIST to illustrate

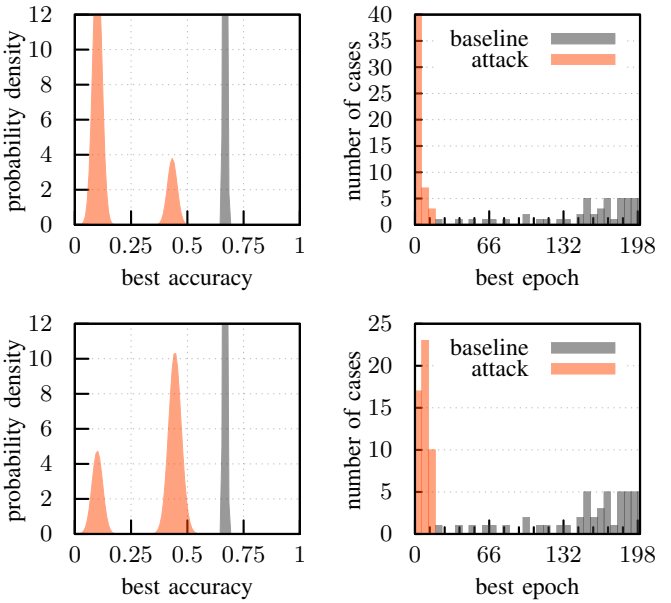


Fig. 9: The soft knockout attack on convolutional nets applies $r = 0.2$ to every second layer on the CIFAR10 dataset. The upper plot corresponds to an attack with softening of one filter, the one below to a soft attack on 16 filters.

this, although it occurs in general. For slightly lower $r = 0.2$, however, most seeds achieve baseline accuracy. Once again the training time is increased on average, as visible in fig. 8.

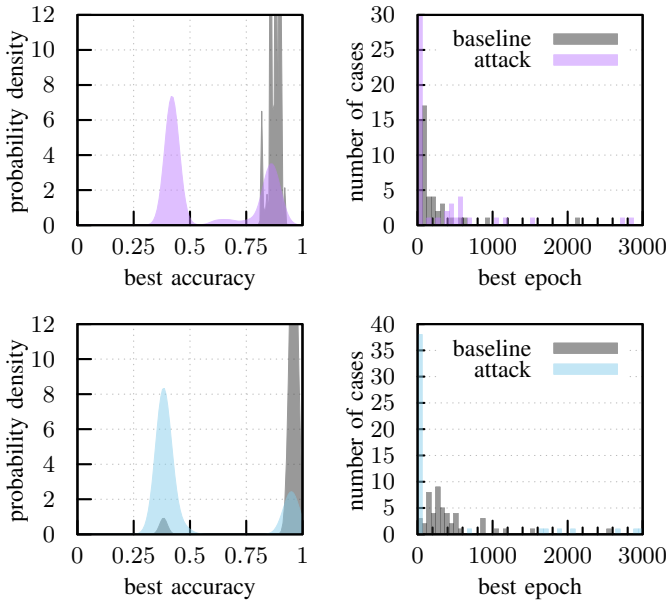


Fig. 10: Using off-the-shelf optimization to knock out neurons on the credit (above) and spam (below) tasks.

We finally apply an adapted version of the soft knockout attack to a convolutional network on CIFAR10. The results are presented in fig. 9. We set in every second (uneven) layer to $r = 0.2$, otherwise to 0.5 We compare between applying softening to only one filter or to sixteen filters (the latter means half of the available filters). In contrast to the results for fully connected networks, we do observe a reasonable decrease in the best achieved accuracy. This accuracy is typically reached at the beginning of the training, which means that the networks actually get worse during training instead of converging to a good configuration.

C. Optimization Based Knockout Attack

As we have seen in the previous section, we cannot harm small networks using the previously introduced soft knockout attack. Hence, we introduce the generic optimization based attack for small networks in section V-D. We implemented this attack using an off-the-shelf optimizer provided by Scipy [15] and present the respective experimental results.

The optimization finishes quickly for our small networks (with a runtime below one second). We depict the resulting accuracy distributions in fig. 10. Independent of the dataset, networks either fail completely (best accuracy in iteration 0), or they converge to the original accuracy (however slower). We conclude that small networks can be targeted, albeit it is hard to decrease accuracy in a stealthy way.

D. Shift Attack

The shift attack (cf. section V-B2) gives more fine-grained control over the network that the victim trains. For fully connected networks, the shift parameter is equivalent to the number of active neurons in the network. We depict our results with a shift parameter of 4 and 8 on MNIST and Fashion-MNIST in fig. 11. Both decrease the accuracy significantly but without making the network fail completely.

As expected, as the shift decreases and less neurons are available to the network, the networks' performance decreases as well. On Fashion-MNIST, we observe an increase in training time of around 50 epochs. This is less clear for MNIST, where several networks are failing, and achieve their best (random guess) accuracy in epoch one. We observe that on these two datasets, a shift parameter greater than 12 does not decrease the accuracy. For smaller shifts, the networks obtain a lower accuracy after the same or a longer training time.

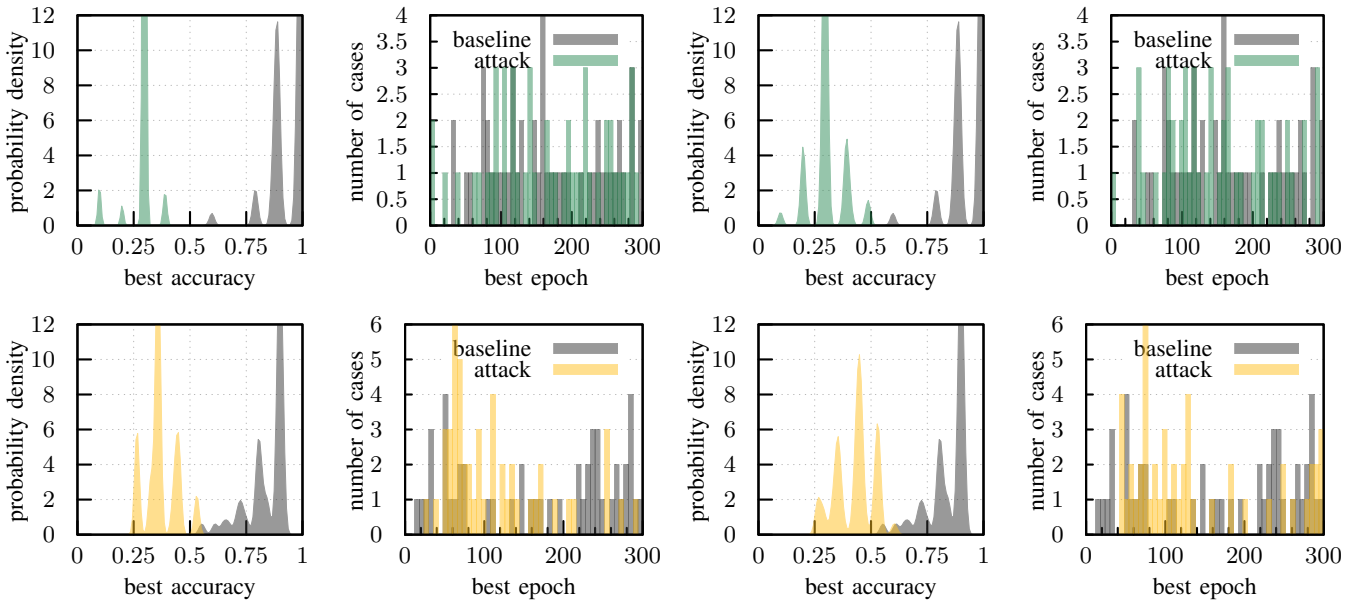
We additionally depict the results on convolutional networks on the CIFAR10 dataset in fig. 12. We once again apply a shift of either four or eight and set the number of filters this shift is applied to to either one or sixteen. As for the fully connected networks, we observe a strong decrease in accuracy.

We observe that most networks fail for a shift of four, independently of the number of filters that are affected. With a shift of eight, the networks obtain an average accuracy around 43% if one filter is affected and around 50% if the number of filters is increased to 16. In contrast to the previous attacks on dense networks, we mostly observe a decrease in training time. An exception to this is a shift of four applied to sixteen filters, where the training time is either very short or rather long.

E. Conclusion on Attacks

To conclude the section, we first want to discuss the parameter choices of the attacker. As we have seen, small shifts and intermediate values of r ruin learning entirely, whereas large shifts or extreme values of r often delay learning or have no effect. In accordance with what could be expected, we further observe that a task like CIFAR10 needs more active neurons than for example MNIST. This is a potential problem for our attack because the number of deactivated neurons has to be adapted in order the both stealthy and effective. Yet, the victim gives us some knowledge about the difficulty of the task when she initializes the network: A fully connected network for Spam is much smaller than one for MNIST. The attacker could thus apply a simple rule that activates an appropriate fraction of connections. Even if the victim enlarges the network and finally succeeds, the attacker still achieves her goal: The network is unnecessarily large and thus needs more time to train.

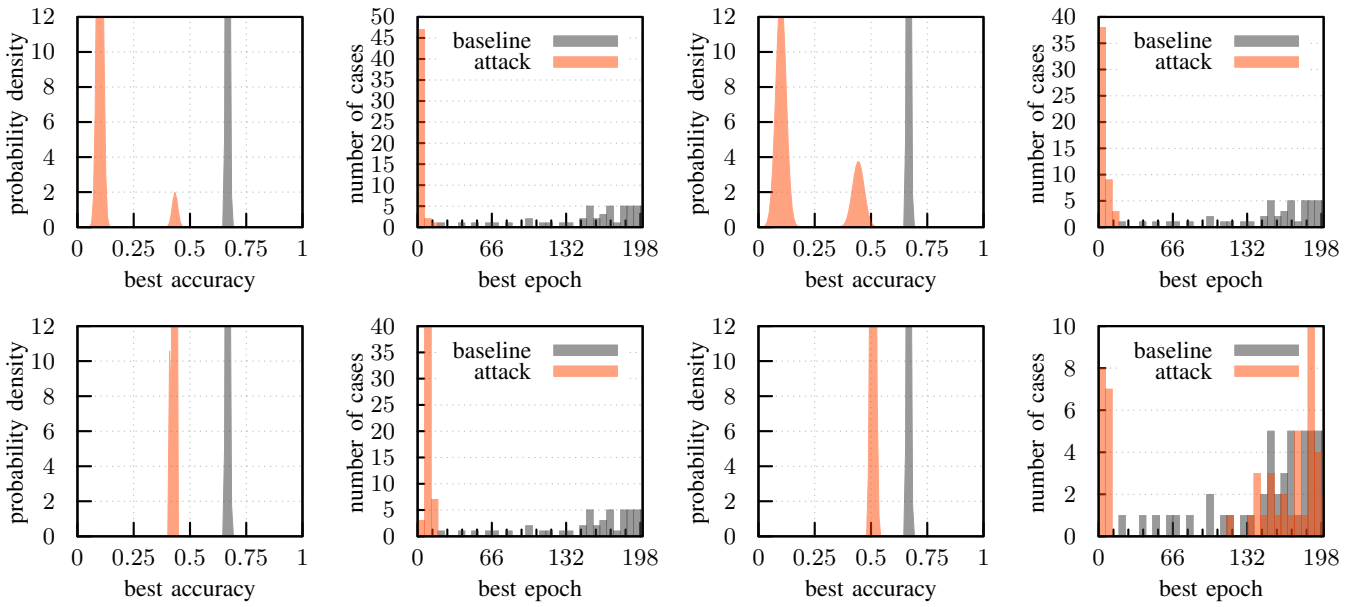
All in all, we conclude that it is indeed possible to harm a network by changing the initialization with relatively straightforward attacks, even if no knowledge about the data or task is available. At the same time, our evaluation shows that network convergence is sometimes bad even for the natural, untampered initializations. The baselines in fig. 11 on MNIST and Fashion-MNIST are an example, where some untampered networks only achieve 60% or 80% accuracy. This might imply the existence of more stealthy attacks. More research is needed



(a) Shift set to 4.

(b) Shift set to 8.

Fig. 11: The shift attack on the MNIST (above) and Fashion-MNIST (below) tasks.



(a) Shift applied to one filter.

(b) Shift applied to sixteen filters.

Fig. 12: The shift attack on convolutional architectures. We vary both the shift (upper plots' shift is four, lower plots' shift is eight) and the number of filters the shift is applied to (left and right plots). Evaluation dataset is CIFAR10.

to gather knowledge about symptoms and defenses in these cases.

VII. Defending Initialization Attacks

In this section, we want to discuss defenses against the previously introduced and evaluated, harmful attacks. We start with a comment on algorithms that are by their nature immune to our attacks and then dive into direct defenses applicable when no other algorithm can be used.

Algorithms like support vector machines or one layer neural networks (logistic regression) are based on a convex optimization problem. They are hence not as vulnerable to a bad initialization as deep neural networks. Yet, algorithms such as support vector machines tend to not scale well to large amounts of data [4]. In addition, deep learning is particularly well suited to perform certain tasks like image classification and is currently state of the art for many tasks [13]. Due to that, switching to other algorithms is usually not an option. However, if deep learning is applied, we need to make sure that malicious initializations do not occur.

The typical diagnostics are not enough for that. During training, developers typically monitor the loss and some statistics of the network, as for example the distribution of the weights. By definition, the latter is not changed by our attacks. The loss, on the other hand, looks unsuspicious as well because the network is still trained in the regular way, just with a reduced capacity. An example is shown in fig. 13. The curves are clearly different but they might both be the result of an untampered learning process. The victim might conclude that the learning rate is too small or that the network is not expressive enough but all that will of course not fix the underlying problem of an artificially reduced capacity of the network.

Beyond the standard diagnostics, the resorting is clearly visible to the victim if the weight matrices are plotted, as we show in fig. 14. Yet, the full weight matrices are typically not monitored, as they are assumed to be random anyway and as dumps of all the weights can quickly become very storage intensive due to the size of modern neural networks. Analogously, our attack on convolutional networks can be discovered by visualizing the filters that were learned, as it is shown in fig. 15.

Last but nor least, an indirect defense to this kind of attacks is provided by well tested datasets such as MNIST, where achievable accuracies are documented: A decrease in accuracy will quickly be noticed in this context. This might lead to an arms-race, though, when the attacker does not target a network that fits the structure of these datasets.

VIII. Conclusion

In this work, we presented a new threat model for deep learning models that we call *adversarial initialization*. We showed how an attacker can exploit the lack of understanding and the typical practices of deep learning: A mere permutation of any randomly chosen initial weights is sufficient to decrease accuracy or increase training time significantly

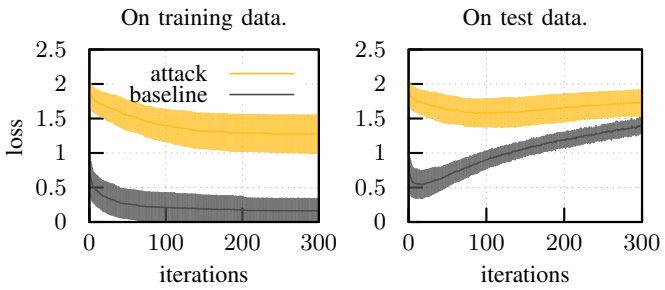


Fig. 13: Visualization of the loss over time during training. We apply the shift attack with shift 8 to a fully connected network and train on Fashion MNIST. We show both the training loss (left) and the test loss (right).

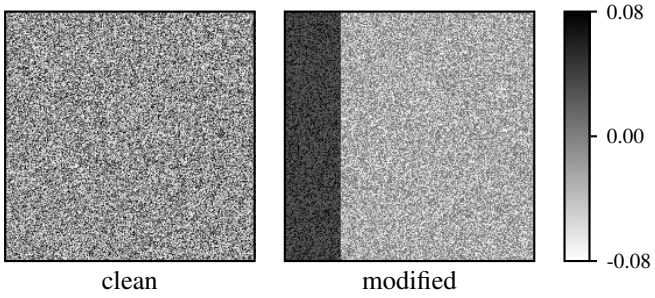


Fig. 14: Comparison of clean and modified weight matrices from actual experiments.

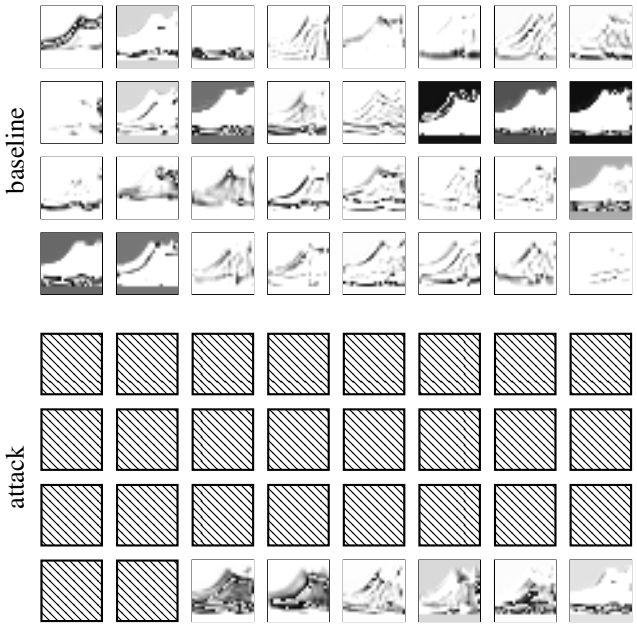


Fig. 15: Comparison of the convolutional filter output for one sample on a untouched (above) and a network initialized by an adversary (below). Filters with zero output are hatched.

without altering any of the typical diagnostics in a way that would invoke suspicions. We complemented our empirical study with a formal analysis which backs up our findings. Our results further hint towards the existence of even more stealthy attacks: More research is needed to identify, asses and mitigate these. For the attacks in this work, however, defenses are straight forward: plotting weight matrices or visualizing the learned convolutional filters suffices to detect an attacker.

IX. Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the project CISPA_AutSec (FKZ: 16KIS0753). Marius Mosbach acknowledges partial support by the German Research Foundation (DFG) as part of SFB 1102.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[2] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

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

[4] C. M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[5] N. Cheney, M. Schrimpf, and G. Kreiman. On the robustness of convolutional neural networks to internal architecture and weight perturbations. *arXiv preprint arXiv:1703.08245*, 2017.

[6] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma. Adversarial classification. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 99–108, New York, NY, USA, 2004. ACM.

[7] T. Denoeux and R. Lengell. Initializing back propagation networks with prototypes. *Neural Networks*, 6(3):351363, Jan 1993.

[8] G. Drago and S. Ridella. Statistically controlled activation weight initialization (scawi). *IEEE Transactions on Neural Networks*, 3(4):627631, Jul 1992.

[9] R. Giryes, G. Sapiro, and A. M. Bronstein. Deep neural networks with random gaussian weights: A universal classification strategy? *IEEE Transactions on Signal Processing*, 64(13):3444–3457, Jul 2016.

[10] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[11] B. Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 580–589. Curran Associates, Inc., 2018.

[12] B. Hanin and D. Rolnick. How to start training: The effect of initialization and architecture. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 569–579. Curran Associates, Inc., 2018.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[14] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[15] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[16] J. Kadmon and H. Sompolinsky. Transition to chaos in random neuronal networks. *Phys. Rev. X*, 5:041030, Nov 2015.

[17] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] M. Lichman. UCI machine learning repository, 2013.

[21] Y. Liu, S. Ma, Y. Aafer, W. Lee, J. Zhai, W. Wang, and X. Zhang. Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[22] Y. Liu, L. Wei, B. Luo, and Q. Xu. Fault injection attack on deep neural network. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 131–138. IEEE Press, 2017.

[23] C. D. Manning. Computational Linguistics and Deep Learning. *Computational Linguistics*, 41(4):701–707, Dec 2015.

[24] S. Mei and X. Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *AAAI*, pages 2871–2877, 2015.

[25] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 2017.

[26] D. Mishkin and J. Matas. All you need is a good init. *CoRR*, abs/1511.06422, 2015.

[27] S. J. Oh, M. Augustin, M. Fritz, and B. Schiele. Towards reverse-engineering black-box neural networks. In *International Conference on Learning Representations*, 2018.

[28] S. Park, J.-K. Park, S.-J. Shin, and I.-C. Moon. Adversarial dropout for supervised and semi-supervised learning. *arXiv preprint arXiv:1707.03631*, 2017.

[29] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[30] B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein, and S. Ganguli. Exponential expressivity in deep neural networks through transient chaos. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3360–3368. Curran Associates, Inc., 2016.

[31] B. I. Rubinstein, B. Nelson, L. Huang, A. D. Joseph, S.-h. Lau, S. Rao, N. Taft, and J. D. Tygar. Antidote: understanding and defending against poisoning of anomaly detectors. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2009.

[32] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *10th International Conference on Malicious and Unwanted Software, MALWARE 2015, Fajardo, PR, USA, October 20-22, 2015*, pages 11–20, 2015.

[33] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *NIPS*, 2018.

[34] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. pages 3–18, 2017.

[35] R. Stevens, O. Suciu, A. Ruef, S. Hong, M. Hicks, and T. Dumitraş. Summoning Demons: The Pursuit of Exploitable Bugs in Machine Learning. *ArXiv e-prints*, Jan. 2017.

[36] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[37] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Proceedings of the 2014 International Conference on Learning Representations*. Computational and Biological Learning Society, 2014.

[38] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 601–618, 2016.

[39] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. 2017.

[40] Q. Xiao, K. Li, D. Zhang, and Y. Jin. Wolf in sheep's clothing-the downscaling attack against deep learning applications. *arXiv preprint arXiv:1712.07805*, 2017.

[41] Q. Xiao, K. Li, D. Zhang, and W. Xu. Security risks in deep learning implementations. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, pages 123–128, 2018.

[42] J. Y. Yam and T. W. Chow. A weight initialization method for improving training speed in feedforward neural network. *Neurocomputing*, 30(1-4):219–232, 2000.

APPENDIX

A. Evaluation on Alternative Attacks.

We described some attacks that might be alternatives to ours in section V and concluded that they would not work very well in practice due to aspects beyond their effect in a controlled setting. For the sake of completeness, we present the evaluation of some of these alternative attacks in this section. This evaluation follows the set up described in section VI-A. The results presented here can be compared against those for our main attacks.

For the first attack we change the variance of the weights: instead of offsetting it to the ideal value $2/\text{fan}_{\text{in}}$, we set it to $2/\text{fan}_{\text{out}}$. We report the results of our experiment on the Credit and Spam tasks in fig. 16. We observe an increase in training time, and an increase of non-converging networks on the credit data. The accuracy on the spam data does not change.

In a second experiment, we alter the leaning rate maliciously to slow down learning. Instead of the default 10^{-3}, we set the learning rate to 10^{-6} and depict the results in fig. 17. We observe both intended effects, as the training time increases and the accuracies decrease.

Finally, we consider the effect of choosing a very large dropout probability during training. We evaluate this on Spam (dropout rate 0.005) and MNIST (dropout rate 0.01). We depict the results of our experiments in fig. 18. We again observe both desired properties: an increase in training time and a decrease in accuracy.

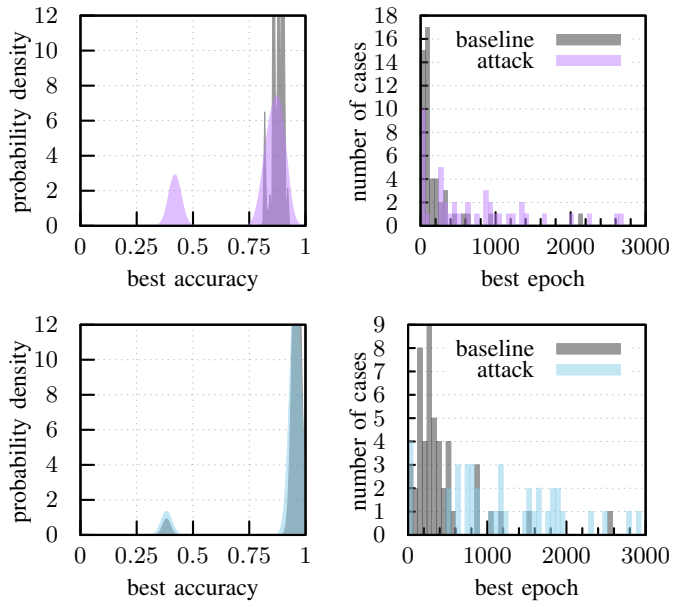


Fig. 16: Setting variance to non-ideal value on the credit (above) and the spam (below) task.

B. Evaluation with SGD and Glorot Initializer.

In the main evaluation, we focus on He initialization and the Adam optimizer. In this appendix, we add additional results using the Glorot initializer and the SGD optimizer to show that the difference in vulnerability is negligible. We again follow the set up described in section VI-A. We start with the effect of the shift attack on MNIST and Fashion-MNIST using the Glorot initializer in fig. 19. He performs better than the Glorot initializer under and without the attack. Additionally, we compare SGD and Adam optimizer on the optimization

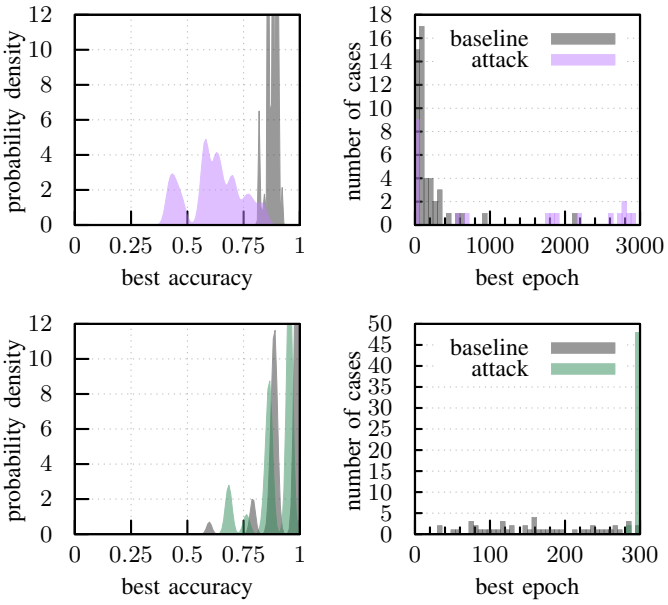


Fig. 17: Maliciously set learning rate on credit (above) and MNIST (below) task.

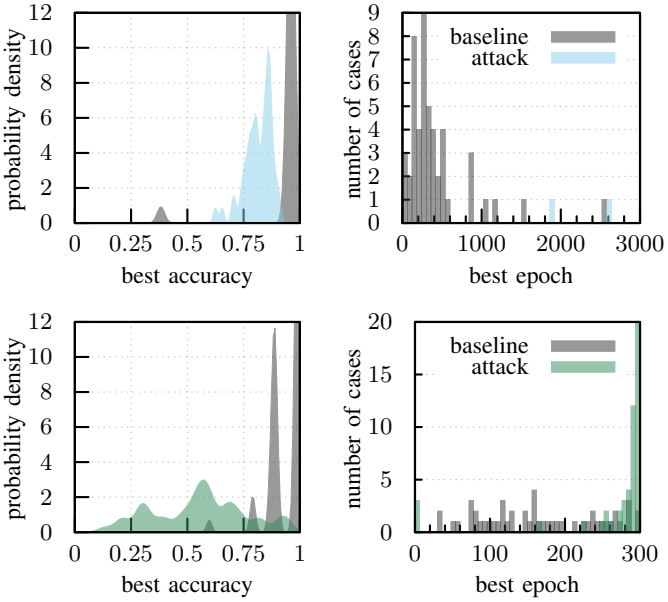
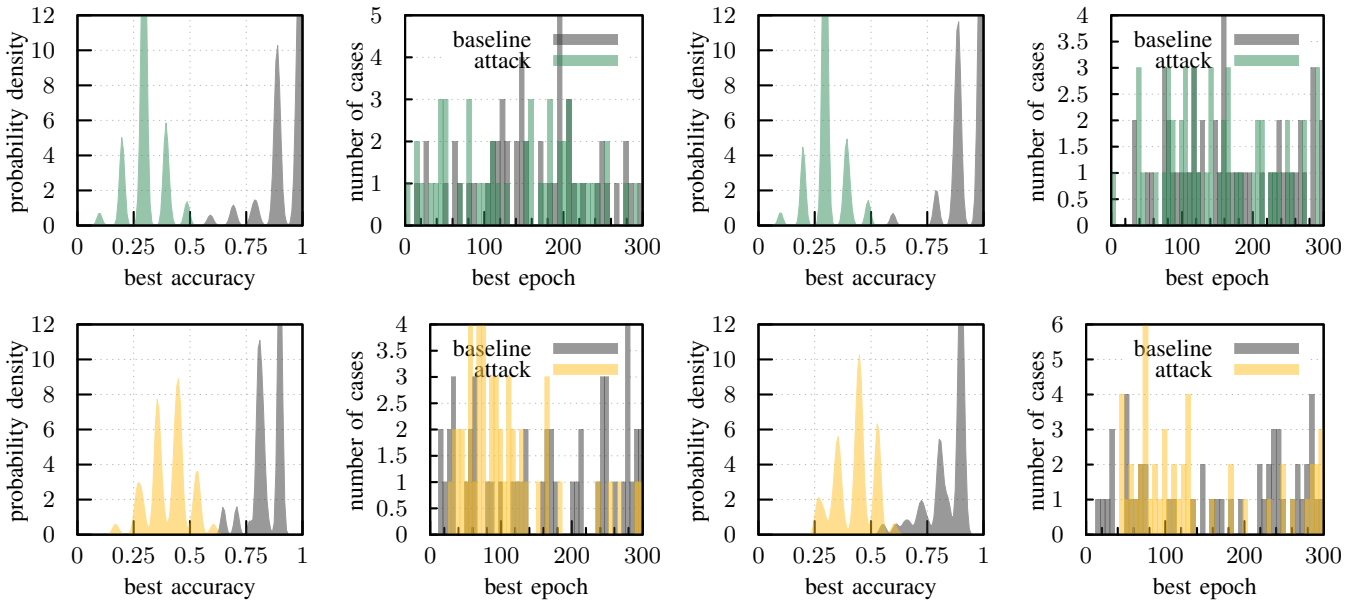


Fig. 18: Maliciously set dropout (attack) and benign training (baseline) on spam (above) and MNIST (below).

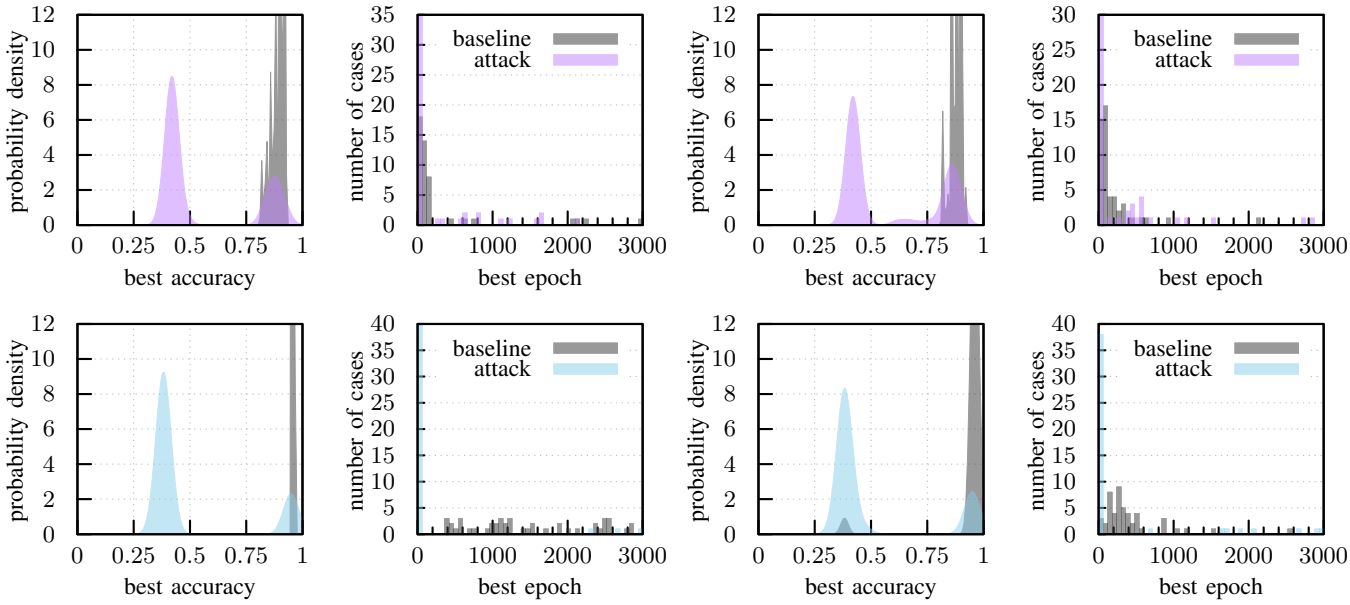
attack in fig. 20. Adam converges earlier on both datasets and yields higher accuracy both without and under attack. All in all the results indeed show that the attack does not hinge on a particular initializer.



(a) Glorot initializer.

(b) He initializer.

Fig. 19: The influence of the initializer on vulnerability to the shift attack. Datasets are MNIST (above) and Fashion-MNIST (below), shift is set to 8.



(a) SGD optimizer.

(b) Adam optimizer.

Fig. 20: The influence of the optimizer on vulnerability to the optimization attack. Datasets are credit (above) and spam (below).