Neural Networks with Structural Resistance to Adversarial Attacks

Luca de Alfaro

Computer Science and Engineering Department
University of California, Santa Cruz
luca@ucsc.edu

September 24, 2018
Technical Report UCSC-SOE-18-13
School of Engineering, University of California, Santa Cruz

Abstract

In adversarial attacks to machine-learning classifiers, small perturbations are added to input that is correctly classified. The perturbations yield adversarial examples, which are virtually indistinguishable from the unperturbed input, and yet are misclassified. In standard neural networks used for deep learning, attackers can craft adversarial examples from most input to cause a misclassification of their choice.

We introduce a new type of network units, called RBFI units, whose non-linear structure makes them inherently resistant to adversarial attacks. On permutation-invariant MNIST, in absence of adversarial attacks, networks using RBFI units match the performance of networks using sigmoid units, and are slightly below the accuracy of networks with ReLU units. When subjected to adversarial attacks, networks with RBFI units retain accuracies above 90% for attacks that degrade the accuracy of networks with ReLU or sigmoid units to below 2%. RBFI networks trained with regular input are superior in their resistance to adversarial attacks even to ReLU and sigmoid networks trained with the help of adversarial examples.

The non-linear structure of RBFI units makes them difficult to train using standard gradient descent. We show that networks of RBFI units can be efficiently trained to high accuracies using *pseudogradients,* computed using functions especially crafted to facilitate learning instead of their true derivatives. We show that the use of pseudogradients makes training deep RBFI networks practical, and we compare several structural alternatives of RBFI networks for their accuracy.

1 Introduction

Machine learning via deep neural networks has been remarkably successful in a wide range of applications, from speech recognition to image classification and language processing. While very successful, deep neural networks are affected by adversarial examples: small, especially crafter modifications of correctly classified input that are misclassified [SZS+13]. The trouble with adversarial examples is that the modifications to regular input are so small as to be difficult or impossible to detect for a human: this has been shown both in the case of images [SZS+13, NYC15] and sounds [KGB16a, CW18]. Further, the adversarial examples are in some measure transferable from one neural network to another [GSS14, NYC15, PMJ+16, TPG+17], so they can be crafted even without precise knowledge of the weights of the target neural network. Together, these two properties make it possible to craft malicious and undetectable attacks for a wide range of applications that rely on deep neural networks. At a fundamental level, it is hard to provide guarantees about the behavior of a deep neural network, when every correctly classified input is tightly encircled by very similar, yet misclassified, inputs.

Thus far, the approach for obtaining neural networks that are more resistant to adversarial attacks has been to feed to the networks, as training data, an appropriate mix of the original training data, and adversarial examples [GSS14, MMS$^+$17]. In training neural networks using adversarial examples, if the examples are generated via efficient heuristics such as the *fast gradient sign method,* the networks learn to associate the specific adversarial examples to the original input from which they were derived, in a phenomenon known as *label leaking* [KGB16b, MMS$^+$17, TKP$^+$17]. This does not result in increased resistance to general adversarial attacks [MMS$^+$17, CW17b]. If the adversarial examples used in training are generated via more general optimization techniques, as in [MMS$^+$17], networks with markedly increased resistance to adversarial attacks can be obtained, at the price of a more complex and computationally expensive training regime, and an increase in required network capacity.

We pursue here a different approach, proposing the use of neural network types that are, due to their structure, inherently impervious to adversarial attacks, even when trained on standard input only. In [GSS14], the authors connect the presence of adversarial examples to the (local) linearity of neural networks. In a purely linear form $\sum_{i=1}^n x_i w_i$, we can perturb each x_i by ϵ, taking $x_i + \epsilon$ if $w_i > 0$, and $x_i - \epsilon$ if $w_i < 0$. This causes an output perturbation of magnitude $\epsilon \sum_{i=1}^n |w_i|$, or $n\bar{w}$ for $\bar{w}$ the average modulus of w_i. When the number of inputs n is large, as is typical of deep neural networks, a small input perturbation can cause a large output change. Of course, deep neural networks are not globally linear, but the insight of [GSS14] is that they may be sufficiently locally linear to allow adversarial attacks. Following this insight, we develop networks composed of units that are highly non-linear.

The networks on which we settled after much experimentation are a variant of the well known *radial basis functions* (RBFs) [BL88a, CCG91, Orr96]; we call our variant RBFI units. RBFI units are similar to classical Gaussian RBFs, except for two differences that are crucial in obtaining both high network accuracy, and high resistance to attacks. First, rather than being radially symmetrical, RBFIs can scale each input component individually; in particular, they can be highly sensitive to some inputs while ignoring others. This gives an individual RBFI unit the ability to cover more of the input space than its symmetrical variants. Further, the distance of an input from the center of the Gaussian is measured not in the Euclidean, or ℓ_2, norm, but in the infinity norm ℓ_∞, which is equal to the maximum of the differences of the individual components. This eliminates all multi-input linearity from the local behavior of a RBFI: at any point, the output depends on one input only; the n in the above discussion is always 1 for RBFIs, so to say. The "I" in RBFI stands for the infinity norm.

Using deeply nonlinear models is hardly a new idea, but the challenge has been that such models are typically very difficult to train. Indeed, we show that networks with RBFI units cannot be easily trained using gradient descent. To get around this, we show that the networks can be trained efficiently, and to high accuracy, using *pseudogradients.* A *pseudogradient* is computed just as an ordinary gradient, except that we artificially pretend that some functions have a derivative that is different from the true derivative, and especially crafted to facilitate training. In particular, we use pseudoderivatives for the exponential function, and for the maximum operator, that enter the definition of Gaussian RBFI units. Gaussians have very low derivative away from their center, which makes training difficult; our pseudoderivative artificially widens the region of detectable gradient around the Gaussian center. The maximum operator has non-zero derivative only for one of its inputs at a time; we adopt a pseudogradient that propagates back the gradient to all of its inputs, according to their proximity in value to the maximum input. Tampering with the gradient may seem unorthodox, but methods such as AdaDelta [Zei12], and even gradient descent with momentum, cause training to take a trajectory that does not follow pure gradient descent. We simply go one step further, devising a scheme that operates at the granularity of the individual unit.

We show that with these two changes, RBFIs can be easily trained with standard random (pseudo)gradient descent methods, yielding networks that are both accurate, and resistant to attacks. Specifically, we consider *permutation invariant MNIST,* which is a version of MNIST in which the 28×28 pixel images are flattened into a one-dimensional vector of 784 values and fed as a feature vector to neural networks [GSS14]. On this test set, we show that for nets of 512,512,512,10 units, RBFI networks match the classification accuracy of networks of sigmoid units ($(96.96 \pm 0.14)\%$ for RBFI vs. $(96.88 \pm 0.15)\%$ for sigmoid), and are close to the performance of network with ReLU units ($(98.62 \pm 0.08)\%$). When trained over standard training sets, RBFI networks retain accuracies well over 90% for adversarial attacks that reduce the accuracy of ReLU and sigmoid networks to below 2% (worse than random). We show that RBFI networks trained on normal input are superior to ReLU and sigmoid networks trained even with adversarial examples. Our experimental results can be summarized as follows:

- In absence of adversarial attacks, RBFI networks match the accuracy of sigmoid networks, and are slightly lower in accuracy than ReLU networks.
- When networks are trained with regular input only, RBFI networks are markedly more resistant to adversarial attacks than sigmoid or ReLU networks.
- In presence of adversarial attacks, RBFI networks trained on regualar input provide higher accuracy than sigmoid or ReLU networks, even when the latter are trained also on adversarial examples, and even when the adversarial examples are obtained via general projected gradient descent [MMS+17].
- RBFI networks can be successfully trained with pseudogradients; the training via standard gradient descent yields instead markedly inferior results.
- Appropriate regularization helps RBFI networks gain increased resistance to adversarial attacks.

Of course, much work remains to be done, including experimenting with convolutional networks using RBFI units for images. However, the results seem promising, in that RBFI seem to offer a viable alternative to current adversarial training regimes for applications where resistance to adversarial attacks is important.

To conduct our experiments, we have implemented RBFI networks on top of the PyTorch framework [PGC+17]; all the code used in their implementation and in the experiments performed in this paper is available at `https://github.com/lucadealfaro/rbfi`.

2 Related Work

Adversarial examples were first noticed in [SZS+13], where they were generated via the solution of general optimization problems. In [GSS14], a connection was established between linearity and adversarial attacks. A fully linear form $\sum_{i=1}^{n} x_i w_i$ can be perturbed by using $x_i + \epsilon \, \text{sign}(w_i)$, generating an output change of magnitude $\epsilon \cdot \sum_{i=1}^{n} |w_i|$. In analogy, [GSS14] introduced the *fast gradient sign method* (FGSM) method of creating adversarial perturbations, by taking $x_i + \epsilon \cdot \text{sign}(\nabla_i \mathcal{L})$, where $\nabla_i \mathcal{L}$ is the loss gradient with respect to input i. The work also showed how adversarial examples are often transferable across networks, and it asked the question of whether it would be possible to construct non-linear structures, perhaps inspired by RBFs, that are less linear and are more robust to adversarial attacks. This entire paper is essentially a long answer to the conjectures and suggestions expressed in [GSS14].

It was later discovered that training on adversarial examples generated via FGSM does not confer strong resistance to attacks, as the network learns to associate the specific examples generated by FGSM to the original training examples in a phenomenon known as *label leaking* [KGB16b, MMS$^+$17, TKP$^+$17]. The FGSM method for generating adversarial examples was extended to an iterative method, I-FGSM, in [KGB16a]. In [TKP$^+$17], it is shown that using small random perturbations before applying FSGM enhances the robustness of the resulting network. The network trained in [TKP$^+$17] using I-FSGM and ensemble method won the first round of the NIPS 2017 competition on defenses with respect to adversarial attacks.

Carlini and Wagner in a series of papers show that training regimes based on generating adversarial examples via simple heuristics, or combinations of these, in general fail to convey true resistance to attacks [CW17a, CW17b]. They further advocate measuring the resistance to attacks with respect to attacks found via more general optimization processes. In particular, FGSM and I-FGSM rely on the local gradient, and training techniques that break the association between the local gradient and the location of adversarial examples makes networks harder to attack via FGSM and I-FGSM, without making the networks harder to attack via general optimization techniques. In this paper, we follow this suggestion by using a general optimization method, projected gradient descent (PGD), to generate adversarial attacks and evaluate network robustness. [CW16, CW17b] also shows that the technique of *defensive distillation,* which consists in appropriately training a neural network on the output of another [PMW$^+$16], protects the networks from FGSM and I-FGSM attacks, but does not improve network resistance in the face of general adversarial attacks.

In [MMS$^+$17] it is shown that by training neural networks on adversarial examples generated via PGD, it is possible to obtain networks that are genuinely more resistant to adversarial examples. The price to pay is a more computationally intensive training, and an increase in the network capacity required. We provide an alternative way of reaching such resistance, one that does not rely on a new training regime.

3 RBFI Units and RBFI Networks

Consider a classifier f that given an input feature vector $\mathbf{x} \in \mathbb{R}^n$ generates a classification $f(x) \in \{1, \ldots, K\}$, where n is the dimension of the input, and K is the number of classes. An *adversarial attack* for a correctly-classified input $\mathbf{x}$ consists in an input $\mathbf{x}'$ close to $\mathbf{x}$ in a metric of choice, and such that $f(\mathbf{x}) \neq f(\mathbf{x}')$. We choose in this paper the infinity norm, following [GSS14, MMS$^+$17], so that an η-adversarial attack is one in which $\|\mathbf{x} - \mathbf{x}'\|_\infty \leq \eta$.

3.1 RBFI Units

In [GSS14], the adversarial attacks are linked to the linearity of the models. For this reason, we seek to use units that do not exhibit a marked linear behavior, and specifically, units which yield small output variations for small variations of their inputs measured in infinity norm

A linear form $g(\mathbf{x}) = \sum_i x_i w_i$ represents the norm-2 distance of the input vector x to a hyperplane perpendicular to vector $\mathbf{w}$, scaled by $|\mathbf{w}|$. In our quest for robustness, we may seek to replace this norm-2 distance with an infinity-norm distance. However, it turns out that the infinity-norm distance of a point from a plane is not a generally useful concept: it is preferable to consider the infinity-norm distance between points.

Hence, we define our units as variants of the classical Gaussian *radial basis functions* [BL88b, Orr96]. We call our variant RBFI, to underline the fact that they are built using infinity norm. An RBFI unit $\mathcal{N}(\mathbf{u},\mathbf{w})$ for an input in $\mathbb{R}^n$ is parameterized by two vectors of weights $\mathbf{u} = \langle u_1, \ldots, u_n \rangle$ and $\mathbf{w} = \langle w_1, \ldots, w_n \rangle$ Given an input $\mathbf{x} \in \mathbb{R}^n$, the unit produces output

$$\mathcal{N}_\gamma(\mathbf{u},\mathbf{w})(\mathbf{x}) = \exp\left(-\|\mathbf{u} \circ (\mathbf{x} - \mathbf{w})\|_\gamma^2\right) \ , \tag{1}$$

where $\circ$ is the Hadamard, or element-wise, product, and where $\|\cdot\|_\gamma$ indicates the γ-norm. In (1), the vector $\mathbf{w}$ is a point from which the distance to $\mathbf{x}$ is measured in γ-norm, and the vector $\mathbf{u}$ provides scaling factors for each coordinate. Without loss of expressiveness, we require the scaling factors to be non-negative, that is, $u_i \geq 0$ for all $1 \leq i \leq n$. The scaling factors provide the flexibility of disregarding some inputs x_i, by having $u_i \approx 0$, while emphasizing the influence of other inputs. As we are interested here in robustness with respect to the infinity norm, our RBFI units are obtained by taking $\gamma = \infty$, in which case (1) can be written as:

$$\mathcal{N}_\infty(\mathbf{u},\mathbf{w})(\mathbf{x}) = \exp\left(-\max_{1\leq i\leq n}\big(u_i(x_i - w_i)\big)^2\right) \ . \tag{2}$$

The output of a RBFI unit is close to 1 only when $\mathbf{x}$ is close to $\mathbf{w}$ in the coordinates that have large scaling factors. Thus, the unit is reminiscent of an And gate, with normal or complemented inputs, which outputs 1 only for one value of its inputs. Logic circuits are composed both of And and of Or gates. Thus, we introduce an Or RBFI unit by:

$$\mathcal{N}_\infty^-(\mathbf{u},\mathbf{w})(\mathbf{x}) = 1 - \exp\left(-\max_{1\leq i\leq n}\big(u_i(x_i - w_i)\big)^2\right) \ . \tag{3}$$

We construct neural networks out of RBFI units using three types of layers:

- *And layer:* all units in the layer are And units, defined by (2).
- *Or layer:* all units in the layer are Or units, defined by (3).
- *Mixed layer:* the units of the layer are a mix of Or and And RBFI units. When the network is initialized, the And or Or-ness of each unit is chosen at random, and henceforth it is kept fixed during training and classification.

We will provide a comparison of the performance of networks using different types of layers. Obviously, layers using RBFI units can be mixed with layers using other types of units.

3.2 Sensitivity Bounds for Adversarial Attacks

To form an intuitive idea of why networks with RBFI units might resist adversarial attacks, it is useful to compute the sensitivity of individual units to such attacks. For $x \in \mathbb{R}^n$ and $\epsilon > 0$, let $B_\epsilon(x) = \{x' \mid \|x - x'\|_\infty \leq \epsilon\}$ be the set of inputs within distance ϵ from x in infinity norm. Given a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, we call its *sensitivity to adversarial attacks* the quantity:

$$s = \sup_{x\in\mathbb{R}^n} \limsup_{\epsilon\to 0} \frac{\sup_{x'\in B_\epsilon(x)} |f(x) - f(x')|}{\epsilon} \ . \tag{4}$$

The sensitivity (4) represents the maximum change in output we can obtain via an input change within ϵ in infinity norm, as a multiple of ϵ itself. For a single ReLU unit with weight vector $\mathbf{w}$, the sensitivity is given by

$$s = \sum_{i=1}^n |w_i| = \|\mathbf{w}\|_1 \ . \tag{5}$$

The formula above can be understood by noting that the worst case for a ReLU unit corresponds to considering an x for which the output is positive, and taking $x'_i = x_i + \epsilon$ if $w_i > 0$, and $x'_i = -\epsilon$ if $w_i < 0$, following essentially the analysis of adversarial examples in [GSS14]. Similarly, for a single sigmoid unit with weight vector $\mathbf{w}$, as the worst case corresponds to the unit operating in its linear region, we have $s = \frac{1}{4}\|\mathbf{w}\|_1$, where the factor of 1/4 corresponds to the maximum derivative of the sigmoid. For a RBFI unit $\mathcal{N}(\mathbf{u}, \mathbf{w})$, on the other hand, we have:

$$s = \frac{2}{e} \cdot \max_{1 \leq i \leq n} u_i^2 = \frac{2}{e} \cdot \|\mathbf{u}\|_\infty^2 \; . \tag{6}$$

Comparing (5) and (6), we see that the sensitivity of ReLU and Sigmoid units increases linearly with input size, whereas the sensitivity of RBFI units is essentially constant with respect to input size. This helps understand why attacks that rely on small changes to many inputs work well against networks using ReLU and Sigmoid units, but are not very effective against networks that use RBFI units, as we will show experimentally in Section 6.3.

Formulas (5) and (6) can be extended to bounds for whole networks. For a ReLU unit, given upper bounds $\hat{s}_i$ for the sensitivity of input i, we can compute an upper bound for the sensitivity of the unit output via

$$\hat{s} = \sum_{i=1}^{n} \hat{s}_i |w_i| \; .$$

Of course, this is only an upper bound, as the conditions that maximize the effect on one input will not necessarily maximize the effect on other inputs. In general, for a ReLU network with K_0 inputs and layers of $K_1, K_2, \ldots, K_M$ units, let $W^{(k)} = [w_{ij}]^{(k)}$ be its weight matrices, where $w_{ij}^{(k)}$ is the weight for input i of unit j of layer k, for $1 \leq k \leq K_M$. We can efficiently compute an upper bound $\hat{s}$ for the sensitivity of the network via:

$$\hat{\mathbf{s}}^{(0)} = \mathbf{1} \; , \qquad \hat{\mathbf{s}}^{(k)} = |W^{(k)}|\hat{\mathbf{s}}^{(k-1)} \; , \qquad \hat{s} = \|\hat{\mathbf{s}}^{(M)}\|_\infty \; . \tag{7}$$

The formula for Sigmoid networks is identical except for the 1/4 factors. Using similar notation, for RBFI networks we have:

$$\hat{\mathbf{s}}^{(0)} = \mathbf{1} \; , \qquad \hat{s}_j^{(k)} = \frac{2}{e} \cdot \max_{1 \leq i \leq K_{k-1}} \hat{s}_i^{(k-1)} \left(u_{ij}^{(k)} \right)^2 \; , \qquad \hat{s} = \|\hat{\mathbf{s}}^{(M)}\|_\infty \; . \tag{8}$$

The interest in (7) and (8) is not due to the fact that these formulas provide an accurate characterization of network sensitivity to adversarial attacks. They do not, and we will evaluate performance under adversarial attacks experimentally in Section 6.3. Rather, by connecting in a simple way the sensitivity to attacks to the network weights, these formulas suggest the possibility of using weight regularization to achieve robustness: by adding $c\hat{s}$ to the loss function for $c > 0$, we might be able to train networks that are both accurate and robust to attacks. We will show in Section 6.7 that such a regularization helps train more robust RBFI networks, but it does not help train more robust ReLU networks. Incentives do not replace structure, at least in this case.

4 Training RBFI Networks

The non-linearities in (2) make neural networks containing RBFI units difficult to train using standard gradient descent. Indeed, we will show that on permutation-invariant MNIST, that is, on a version of MNIST in which each 28 × 28 pixel image is flattened into a 784-long feature array, training the network using gradient descent yields at most about about 85% accuracy (see

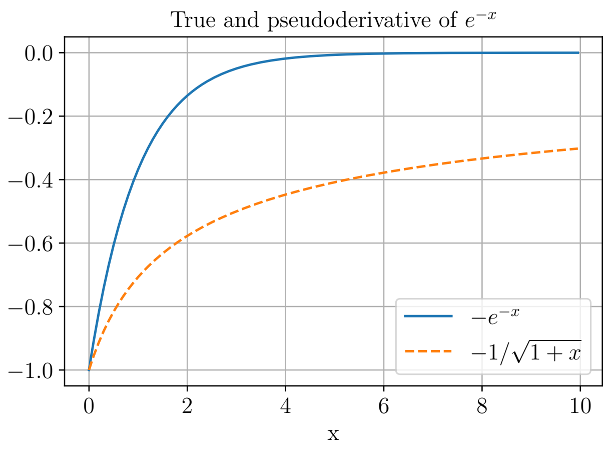


Figure 1: True derivative of e^{-x}, and pseudoderivative used in training.

Section 6.6). The problem lies in the shape of RBF functions. Far from its peak for $\mathbf{x} = \mathbf{w}$, a function of the form (2) is rather flat, and its derivative may not be large enough to cause the vector of weights $\mathbf{w}$ to move towards useful places in the input space during training.

The solution we have found consists in performing gradient descent using not the true gradient of the function, but rather, using a gradient computed using *pseudoderivatives.* These *pseudoderivatives* are functions we use in the chain-rule computation of the loss gradient in lieu of the true derivatives, and are shaped in a way that facilitates learning. Employing such pseudoderivatives, our RBFI networks will be easy to train to high levels of accuracy.

4.1 Pseudogradients

In order to back-propagate the loss gradient through (2), indicating $y = \mathcal{N}_\infty(\mathbf{u}, \mathbf{w})(\mathbf{x})$, we need to compute the partial derivatives $\partial y/\partial x_i$, $\partial y/\partial w_i$, and $\partial y/\partial u_i$, for $1 \leq i \leq n$. The true partial derivatives are computed, of course, applying the chain rule of derivation to (2). To obtain networks that are easy to train, we replace the derivatives for exp and max with alternate functions, which we call *pseudoderivatives.* In the computation of the loss gradient, we will use these pseudoderivatives in place of the true derivatives, yielding a *pseudogradient.*

Exponential function. In computing the partial derivatives via the chain rule, the first step consists in computing $\frac{d}{dz}e^{-z}$, which is of course equal to $-e^{-z}$. The problem is that this derivative is quite small when z is large, and z in (2) is the square of the infinity norm of the scaled distance between $\mathbf{x}$ and $\mathbf{w}$, which can be large. Hence, in the chain-rule computation of the gradient, we replace the true derivative $-e^{-z}$ with the alternate "pseudoderivative" below:

$$-\frac{1}{\sqrt{1+z}} \,. \tag{9}$$

The shape of the true and pseudoderivative is compared in Figure 1. As z increases, the pseudoderivative decreases much more slowly than the true derivative. We experimented with using $-(1+z)^{-\alpha}$ as pseudoderivative, and we found that it works well for values of α between 0.2 and 0.8; we settled on $\alpha = 1/2$, yielding (9).

Max. The gradient of $y = \max_{1 \le i \le n} z_i$, of course, is given by:

$$\frac{\partial y}{\partial z_i} = \begin{cases} 1 & \text{if } z_i = y \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

In training, the problem with (10) is that it transmits feedback only to the largest of the inputs to max. We found it profitable to transmit feedback not only to the largest input, but also to those that are close in value to it. Hence, we obtain the pseudogradient for max by replacing the right-hand side of (10) with:

$$e^{z_i - y} \; . \tag{11}$$

In this way, some of the feedback is transmitted to inputs z_i that approach y. Again, we experimented with functions of the form $e^{\beta(z_i - y)}$, and we found that we could train networks effectively for many values of β; we settled on $\beta = 1$ for simplicity.

One may be concerned that by using the loss pseudogradient as the basis of optimization, rather than the true loss gradient, we may then take optimization steps that cause an increase in true loss. This may occur, of course — but so it may when the amplitude and direction of steps is influenced by algorithms such as AdaDelta [Zei12], or even gradient descent with momentum. Further, the pseudogradient can be zero even when the true gradient is not, and this would cause the training to settle in a position that is not a (local) minimum or saddle point of the cost function. However, in practice this is exceedingly rare, and it is a much smaller risk than simply ending up in a suboptimal local minimum. Ultimately, the efficacy of these training strategies is best evaluated experimentally.

4.2 Bounding the Weight Range

In training RBFI networks, it is useful to bound the range of the $\mathbf{u}$ and $\mathbf{w}$ weight vectors. If no lower bound is used for the individual components u_i of $\mathbf{u}$, gradient descent with discrete step sizes can cause the components to become negative, which is useless, as the sign of u_i does not matter, and even counterproductive, as an impulse to lower the value of u_i could result in an overshoot, ending up with an updated value $u'_i < 0$ with $-u'_i > u_i$. The upper bound to the components of $\mathbf{u}$ performs an important role, as it bounds the sensitivity of the unit outputs with respect to changes in their inputs, as the slope of (2) with respect to the inputs depends on the magnitude of $\mathbf{u}$.

For the first network layer, the weights w should be chosen with a range that matches the input range. For subsequent layers, we found that bounding the components of w to the interval $[0, 1]$, which is the output range of a RBFI unit, works well.

5 Generating Adversarial Examples

We will evaluate the robustness of neural networks with respect to both adversarial attacks, and input noise. Consider a network trained with cost function $J(\theta, \mathbf{x}, \mathbf{y})$, where θ is the set of network parameters, $\mathbf{x}$ is the input, and $\mathbf{y}$ is the output. Indicate with $\nabla_{\mathbf{x}} J(\theta, \mathbf{x}', \mathbf{y})$ the gradient of J wrt its input $\mathbf{x}$ computed at values $\mathbf{x}'$ of the inputs, parameters θ, and output $\mathbf{y}$. For each input $\mathbf{x}$ belonging to the testing set, given a perturbation amount $\epsilon > 0$, we produce an adversarial example $\tilde{\mathbf{x}}$ with $\|x - \tilde{\mathbf{x}}\|_\infty \le \epsilon$ using the following techniques.

Fast Gradient Sign Method (FGSM) [GSS14]. If the cost were linear around $\mathbf{x}$, the optimal ϵ-max-norm perturbation of the input would be given by $\epsilon \operatorname{sign}(\nabla_{\mathbf{x}} J(\theta, \mathbf{x}, \mathbf{y}))$. This suggests taking as adversarial example:

$$\tilde{\mathbf{x}} = [\![\mathbf{x} + \epsilon \operatorname{sign}(\nabla_{\mathbf{x}} J(\theta, \mathbf{x}, \mathbf{y}))]\!]_0^1 \ , \tag{12}$$

where $[\![\mathbf{x}]\!]_a^b$ is the result of clamping each component of $\mathbf{x}$ to the range $[a, b]$; the clamping is necessary to generate a valid MNIST image.

Iterated Fast Gradient Sign Method (I-FGSM) [KGB16a]. Instead of computing a single perturbation of size ϵ using the sign of the gradient, we apply M perturbations of size ϵ/M, each computed from the endpoint of the previous one. Precisely, the attack computes a sequence $\tilde{\mathbf{x}}_0, \tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_M$, where $\tilde{\mathbf{x}}_0 = \mathbf{x}$, and where each $\tilde{\mathbf{x}}_{i+1}$ is obtained, for $0 \leq i < M$, by:

$$\tilde{\mathbf{x}}_{i+1} = \left[\!\!\left[\tilde{\mathbf{x}}_i + \frac{\epsilon}{M} \operatorname{sign}(\nabla_{\mathbf{x}} J(\theta, \tilde{\mathbf{x}}_i, \mathbf{y}))\right]\!\!\right]_0^1 \ . \tag{13}$$

We then take $\tilde{\mathbf{x}} = \tilde{\mathbf{x}}_M$ as our adversarial example. This attack is more powerful than its single-step version, as the direction of the perturbation can better adapt to non-linear cost gradients in the neighborhood of $\mathbf{x}$ [KGB16a].

Projected Gradient Descent (PGD) [MMS$^+$17]. For an input $\mathbf{x} \in \mathbb{R}^n$ and a given maximum perturbation size $\epsilon > 0$, we consider the set $B_\epsilon(\mathbf{x}) \cap [0, 1]^n$ of valid inputs around $\mathbf{x}$, and we perform projected gradient descent (PGD) in $B_\epsilon(\mathbf{x}) \cap [0, 1]^n$ of the negative loss with which the network has been trained (or, equivalently, projected gradient ascent wrt. the loss). By following the gradient in the direction of increasing loss, we aim at finding mis-classified inputs in $B_\epsilon(\mathbf{x}) \cap [0, 1]^n$. As the gradient is non-linear, to check for the existence of adversarial attacks we perform the descent multiple times, each time starting from a point of $B_\epsilon(\mathbf{x}) \cap [0, 1]^n$ chosen uniformly at random.

Noise. In addition to the above adversarial examples, we will study the robustness of our networks by feeding them inputs affected by noise. For a testing input $\mathbf{x}$ and a noise amount $\epsilon \in [0, 1]$, we produce an ϵ-noisy version $\breve{\mathbf{x}}$ via

$$\breve{\mathbf{x}} = (1 - \epsilon)\mathbf{x} + \epsilon\chi \ ,$$

where χ is a random element of the input space. For permutation-invariant MNIST, the vector χ is chosen uniformly at random in the input space $[0, 1]^n$. This noise model is tailored to images: as ϵ grows, the resulting $\breve{\mathbf{x}}$ contains less signal, and more white noise. Of course, this is one of a multitude of reasonable noise models; nevertheless, it will provide at least a coarse indication of how the networks perform in presence of input perturbations.

FGSM and I-FGSM attacks are powerful heuristics for finding adversarial examples, but they are not general search procedures. Indeed, [CW17b] shows how many networks that resist these attacks still misclassify some adversarial examples. Hence, [CW17b] argues persuasively that a proper evaluation of susceptibility to adversarial attacks can only be done via general optimization techniques, which search for small input perturbations that cause misclassifications. The PGD attacks we have included in our experiments perform a general search and optimization of perturbations with high loss, which can lead to mis-classified examples. As in this paper we consider the infinity norm, these attacks are of generality and power comparable with the attacks described in [CW17b].

6 Experiments on Permutation-Invariant MNIST

6.1 Implementation of RBFI Networks

We implemented RBFI networks in the PyTorch framework [PGC+17]; the code is available at `https://github.com/lucadealfaro/rbfi`. The resulting implementation is well-suited to running on GPUs, which yield a very large speedup.

PyTorch provides automatic gradient propagation: one needs only define the forward path the feature vectors take in the networks, and the framework takes care of computing and propagating the gradients via a facility called AutoGrad [PGC+17]. Crucially, PyTorch makes it easy to define new functions: for each new function f, it is necessary to specify the function behavior $f(\mathbf{x})$, and the function gradient $\nabla_{\mathbf{x}} f$. This makes it very easy to implement RBFI networks. All that is required consists in implementing two special functions: a *LargeAttractorExp* function, which has forward behavior e^{-x} and backward gradient propagation according to $-1/\sqrt{1+x}$, and *SharedFeedbackMax,* which behaves forward like max, and backwards as given by (11). These two functions are then used in the definition of RBFI units, as per (2) and (3), with the AutoGrad facility of PyTorch providing then backward gradient propagation for the complete networks.

We also introduced to PyTorch *bounded parameters,* which have a prescribed range that is enforced during network training; these bounded parameters were used to implement our $\mathbf{u}$ and $\mathbf{w}$ weight vectors.

6.2 Experimental Setup

Dataset. We use the MNIST dataset [LBBH98] for our experiments, following the standard setup of 60,000 training examples and 10,000 testing examples. Each digit image was flattened to a one-dimensional feature vector of length $28 \times 28 = 784$, and fed to a fully-connected neural network; this is the so-called *permutation-invariant* MNIST.

Neural networks. We compared the accuracy of the following network structures.

- **Sigmoid.** We consider fully-connected networks consisting of sigmoid units. The last layer consists of 10 units, corresponding to the 10 digits. We train sigmoid networks using square-error as loss function; we experimented with cross-entropy loss and other losses, and square-error loss performed as well as any other loss in our experiments.

- **ReLU.** We consider fully-connected ReLU networks [NH10, KSH12], again whose last layer consists of 10 units. The output of ReLU networks is fed into a softmax, and the network is trained via cross-entropy loss.

- **RBFI.** We consider fully-connected networks consisting of RBFI units. As for networks of sigmoid units, the last layer has 10 units, and the network is trained using square-error loss. For a RBFI network with m layers, we denote its type as RBFI$(K_1, \ldots, K_m \mid t_1, \ldots, t_m)$, where $K_1, \ldots, K_m$ are the numbers of units in each layer, and where the units in layer i are And units if $t_i = \wedge$, Or units if $t_i = \vee$, and are a random mix of And and Or units if $t_m = *$.

Obviously, a network could mix layers consisting of sigmoid, ReLU, and RBFI units, or indeed the units could be mixed in each layer, but we have not experimented with such hybrid architectures. Unless otherwise noted, we use bounds of $[0.01, 3]$ for the components of the u-vectors, and $[0, 1]$ for the w-vectors, the latter corresponding to the value range of MNIST pixels.

Training and testing. We trained all networks with the AdaDelta optimizer [Zei12], which yielded good results for all networks considered. Unless otherwise noted, we performed 10 runs of each experiment with different seeds for the random generator used for weight initialization, in order to measure the mean and standard deviation of each result. When reporting an accuracy as $X\% \pm Y\%$, X is the mean value of the percent accuracy, and Y the standard deviation of the individual run outcomes (rather than the standard deviation of the mean). Error bars in the plots correspond to one standard deviation of individual run results.

Attacks. We performed the adversarial attacks as follows. For FGSM, and for noise, the attack is completely determined by its amplitude ϵ; the attacks were applied to all examples in the test set. In I-FGSM attacks, we performed 10 iterations of (13). The attack was applied to all data in the test set.

As PGD attacks are considerably more computationally intensive than the other attacks considered, for each experiment we select the neural network model trained by the first of our set of runs, and we compute the performance under PGD attacks over the first 5,000 examples in the test set, or half of it. To generate adversarial examples for input $\mathbf{x}$, we start from a random point in $B_\epsilon(\mathbf{x})$ and we perform 100 steps of projected gradient descent using the AdaDelta algorithm to tune step size; if at any step a misclassified example is generated, the attack is considered successful. For each input $\mathbf{x}$, we repeat this procedure 20 times, that is, our search is performed with 20 restarts. We experimented with using more than 100 steps of descent, but we observed only minimal changes in attack success rates. We also found that when the network accuracy in presence of PGD attacks was at least 80%, additional restarts contributed very little to the attack efficacy. When the attack succeeded over 20% of the time, or for networks whose accuracy under attack was below 80%, additional restarts did increase somewhat the overall attack success, so that our results should be considered upper bounds for classification accuracy. In any case, as we kept the attack configuration identical for all our experiments, our results will enable us to compare the performance of the different neural network architectures in presence of PGD attacks.

6.3 Performance of RBFI, ReLU, and Sigmoid Networks

Network	Accuracy ($\epsilon = 0$)	FGSM, $\epsilon = 0.3$	I-FGSM, $\epsilon = 0.3$	PGD, $\epsilon = 0.3$	Noise, $\epsilon = 0.3$
ReLU	$\mathbf{98.62 \pm 0.08}$	1.98 ± 0.42	0.06 ± 0.06	67.40	79.36 ± 2.60
Sigmoid	96.88 ± 0.15	0.71 ± 0.43	0.11 ± 0.11	38.78	56.57 ± 2.28
RBFI	96.96 ± 0.14	$\mathbf{94.90 \pm 0.35}$	$\mathbf{93.27 \pm 0.48}$	**93.32**	$\mathbf{96.23 \pm 0.08}$

Table 1: Performance of 512-512-512-10 for MNIST testing input, and in presence of adversarial examples and noise computed with perturbation size $\epsilon = 0.3$. The RBFI network has layers RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$.

We first give the results on the accuracy and resistance to adversarial examples for networks trained on the standard MNIST training set, without the benefit of adversarial examples. We conducted 10 measurement runs for ReLU and Sigmoid networks, and 5 for RBFI networks, which suffices to determine result variance. In each run, we trained the networks for 30 epochs on the MNIST training set, and we evaluated performance both on the original MNIST training set, and on the training set perturbed either by noise, or via adversarial examples computed via FGSM, I-FGSM, or PGD. We chose 30 epochs as all networks reached their peak performance by then. We give results for 4-layer networks, with 512, 512, 512, and 10 units. This is a size at which all the

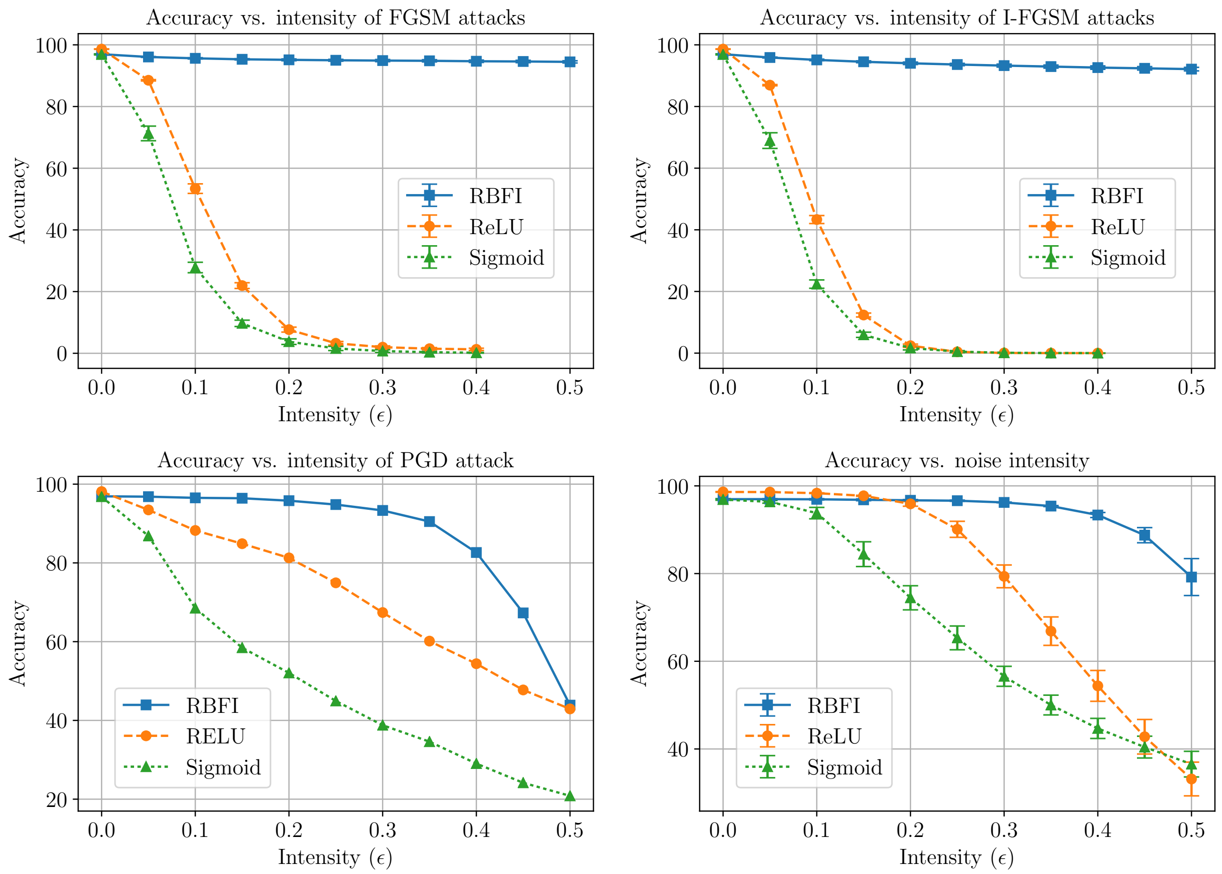


Figure 2: Performance of 512-512-512-10 networks in presence of adversarial examples and noise. The RBFI network has layers RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$.

network types have essentially reached their peak performance. For the RBFI network we chose geometry RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$; as we will detail in Section 6.5, using networks with alternating unit types gives marginally better results.

We report the results in Figure 2 and in Table 1. In absence of perturbations, RBFI networks lose $(1.66 \pm 0.21)\%$ performance compared to ReLU networks (from $(98.62 \pm 0.07)\%$ to $(96.96 \pm 0.14)\%$), and perform comparably to sigmoid networks (the difference is below the standard deviation of the results). When perturbations are present, in the form of adversarial attacks or noise, the performance of RBFI networks is markedly superior.

We note that the FGSM and I-FGSM attacks are not effective against RBFI networks. This phenomenon, observed in [CW17b] also for networks trained via defensive distillation [PMW$^+$16, CW16], is likely due to the fact that for RBFI networks the gradient in proximity of valid inputs offers only limited information about the possible location of adversarial examples. Indeed, even noise provides a better adversary than FGSM and I-FGSM. The PGD attacks are more powerful, as they explore $B_\epsilon(\mathbf{x}) \cap [0, 1]^n$ more thoroughly, sampling points in it at random before following the gradient.

For systems that exhibit marked linearity, such as ReLU and sigmoid networks, FGSM and I-FGSM attacks are instead more powerful than PGD attacks, at least if the latter are carried out with a moderate number of restarts, as in our experiments. In FGSM and I-FGSM, we perturb each input coordinate by ϵ, according to the gradient sign; this may lead to a more effective exploration of

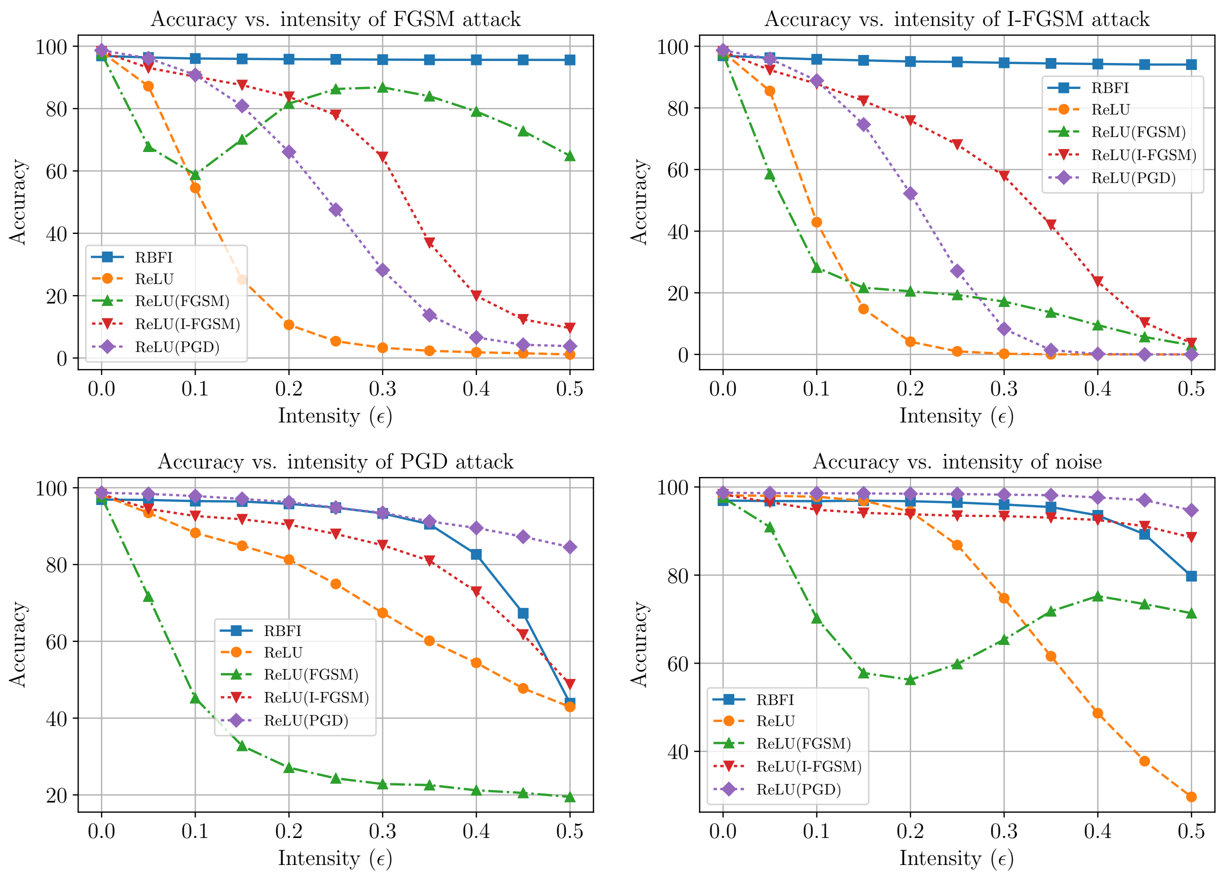


Figure 3: Performance of ReLU networks trained with adversarial examples, vs. performance of RBFI network trained normally, with respect to adversarial input and noise.

$B_\epsilon(\mathbf{x}) \cap [0,1]^n$ than by following the gradient, which tends to explore more the input coordinates for which the gradient component is larger in absolute value. Perhaps by using thousands or millions of restarts, PGD attacks could become as effective as FGSM and I-FGSM attacks for ReLU networks. In our experiments, going from 20 to 100 restarts did not appreciably increase the effectiveness of PGD attacks for $\epsilon \leq 0.3$.

6.4 Performance of Networks Trained With Adversarial Examples

Including adversarial examples in the training set is the most common method used to make neural networks more resistant to adversarial attacks [GSS14, MMS+17]. Here, we explore whether ReLU and Sigmoid networks trained via a mix of normal and adversarial examples can offer a resistance to adversarial attacks compared to that offered by RBFI networks trained on standard examples only. Specifically, we compared the performance of a RBFI network with that of ReLU network trained for 10 epochs as follows:

- **ReLU:** trained normally, on the pairs $(\mathbf{x}, t)$ of input $\mathbf{x}$ and class t in the standard training set.
- **ReLU(FGSM):** for each $(\mathbf{x}, t)$ in the training set, we construct an adversarial example $\tilde{\mathbf{x}}$ via (12), and we feed both $(\mathbf{x}, t)$ and $(\tilde{\mathbf{x}}, t)$ to the network for training.

RBFI$(128, 128, 10)$		RBFI$(64, 64, 64, 10)$		RBFI$(512, 512, 512, 10)$	
Layers	Accuracy	Layers	Accuracy		
$*, *, \vee$	95.00 ± 0.29	$*, *, *, \vee$	93.64 ± 0.30	$*, *, *, \vee$	96.79 ± 0.17
$*, *, \wedge$	94.22 ± 0.30	$*, *, *, \wedge$	93.66 ± 0.34	$*, *, *, \wedge$	96.87 ± 0.22
$\vee, \wedge, \vee$	94.94 ± 0.21	$\wedge, \vee, \wedge, \vee$	93.46 ± 0.45	$\wedge, \vee, \wedge, \vee$	96.96 ± 0.14
$\wedge, \vee, \wedge$	94.26 ± 0.32	$\vee, \wedge, \vee, \wedge$	93.64 ± 0.23		
$\wedge, \wedge, \wedge$	94.25 ± 0.21	$\wedge, \wedge, \wedge, \wedge$	93.30 ± 0.49		
		$\vee, \vee, \vee, \vee$	93.69 ± 0.24		

Table 2: Accuracy of networks trained with different layer kinds. The $128, 128, 10$ and $512, 512, 512, 10$ networks were trained for 30 epochs; the $64, 64, 64, 10$ networks were trained for 10 epochs.

- **ReLU(I-FGSM):** for each $(\mathbf{x}, t)$ in the training set, we construct an adversarial example $\tilde{\mathbf{x}}$ via (13), and we feed both $(\mathbf{x}, t)$ and $(\tilde{\mathbf{x}}, t)$ to the network for training.
- **ReLU(PGD):** for each $(\mathbf{x}, t)$ in the training set, we perform 100 steps of projected gradient descent from a point chosen at random in $B_\epsilon(\mathbf{x}) \cap [0, 1]^n$; denoting by $\mathbf{x}'$ the ending point of the projected gradient descent, we feed both $(\mathbf{x}, t)$ and $(\mathbf{x}', t)$ to the network for training.

We did not include sigmoid networks in the comparison, as they could be trained only with respect to FGSM-generated adversarial examples, for which they gave inferior results. We generated adversarial examples for $\epsilon = 0.3$, which is consistent with [MMS+17]; $\epsilon = 0.3$ also and that, with respect to the results reported in Figure 2 offers a middle value in the range of perturbations we use for testing.

Obviously, many variations are possible to the above adversarial training regime. It is possible to vary the proportion of normal and adversarial inputs. It is also possible to run PGD multiple times for every input, choosing the end-point with the highest loss, at a price of increased computational cost (the above PGD training, by requiring 100 steps of gradient descent for each input, already makes training 100 times more expensive). We could also generate adversarial examples for many values of ϵ, rather than $\epsilon = 0.3$ only. While the variations are many, we believe the above four attacks enable us to gain at least some understanding of how networks trained in adversarial fashion compare with RBFI-based networks.

As in the previous section, we use networks with 512-512-512-10 units, and we choose geometry RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$ for the RBFI network. Training was performed for 10 epochs for Sigmoid and ReLU networks, which seemed sufficient. The results are given in Figure 3. Overall, the best networks may be the simple RBFI networks, trained without the use of adversarial examples: for each class of attack, they exhibit either the best performance, or they are very close in performance to the best performer; this is true for no other network type. For PGD attacks, the best performance is obtained via ReLU(PGD) networks, trained on PGD attacks; however, ReLU(PGD) networks do not fare well in presence of FGSM or I-FGSM attacks.

We note that ReLU(FGSM) networks seem to learn that $\epsilon = 0.3$ FGSM attacks are likely, but they have not usefully generalized the lesson, for instance, to attacks of size 0.1. The S-shaped performance curve of ReLU(FGSM) with respect to FGSM or noise is known as *label leaking:* the network learns to recognize the original input given its perturbed version [KGB16b].

	Accuracy	
Network	Regular Gradient	Pseudogradient
$\text{RBFI}(512, 512, 512, 10 \mid *, *, *, \vee)$	86.35 ± 0.75	96.79 ± 0.17
$\text{RBFI}(256, 256, 256, 10 \mid *, *, *, \vee)$	85.12 ± 1.07	96.63 ± 0.16
$\text{RBFI}(128, 128, 128, 10 \mid *, *, *, \vee)$	83.25 ± 1.34	95.80 ± 0.20
$\text{RBFI}(256, 256, 10 \mid *, *, \vee)$	82.94 ± 1.64	95.60 ± 0.25
$\text{RBFI}(128, 128, 10 \mid *, *, \vee)$	82.40 ± 3.72	95.00 ± 0.29

Table 3: Accuracy achieved training RBFI networks over 30 epochs using regular gradients and pseudogradients.

6.5 RBFI Networks with And, Or, and Mixed Layers

Logic circuits are easy to design in alternating And-Or layers (with the possibility of complementing gate inputs, which RBFI units allow). Hence, we expected networks with alternating unit types to offer superior performance. The results of our experiments with layer types are presented in Table 2. We see that for small network sizes, there are some differences, but the differences generally become statistically insignificant (smaller than the standard deviation in the run results) as the networks become larger. We do not yet have a good insight in why accuracy differences are not larger.

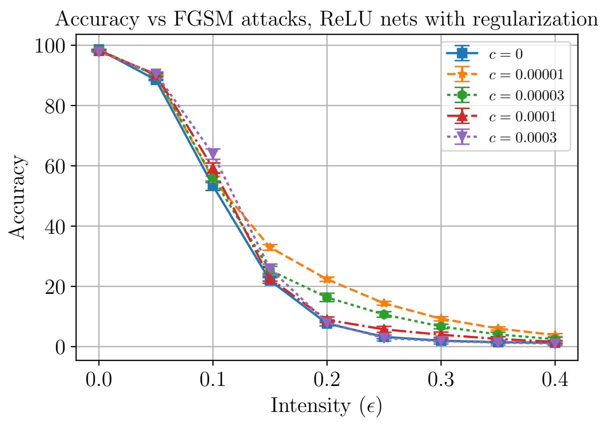
6.6 Pseudogradients vs. Standard Gradients

In order to justify the somewhat un-orthodox use of pseudogradients, we compare the performance achieved by training RBFI networks with standard gradients, and with pseudogradients. Table 3 compares the accuracy attained training RBFI networks using regular gradients, and pseudogradients. We see that training with the true gradient yielded markedly inferior results. The training with regular gradients is also slower to converge; the accuracy in Table 3 was measured after 30 training epochs, when the accuracy of training with both regular, and pseudogradients, had plateaued.

6.7 Learning with Regularization

ReLU networks. In this paper we suggest that neural network may be made more resistant to adversarial attacks via an appropriate choice of structure, namely, by using RBFI units. One may wonder whether the same effect could be achieved more simply by employing weight regularization and standard ReLU units. In Section 3.2 we described how to obtain bounds $\hat{s}$ for attack sensitivity in terms of network weights, via (7) or (7). It is thus natural to consider using $\hat{s}$ as regularization, adding to the loss used to train the networks $c\hat{s}$, for $c \geq 0$. The results of doing so for ReLU networks is illustrated in Figure 4. Moderate values of c yield small improvements, and larger values of c cause the network to cease learning, as keeping the weights low to minimize $c\hat{s}$ takes the precedence over reducing the accuracy-related component of the loss. In summary, at least in our attempts, regularization could not replace structure.

RBFI networks. The choice of upper bound for the components of the **u**-vector influences the resistance of the trained networks to adversarial examples, as indicated by (6) and from the bound (8). MNIST is a digit recognition dataset: pixels that do not contain digit portions are close to 0 in value, while pixels that contain digit portions tend to be close to 1. This provides some heuristic for choosing the bound for the components of **u**: our chosen bound of 3 enables to differentiate



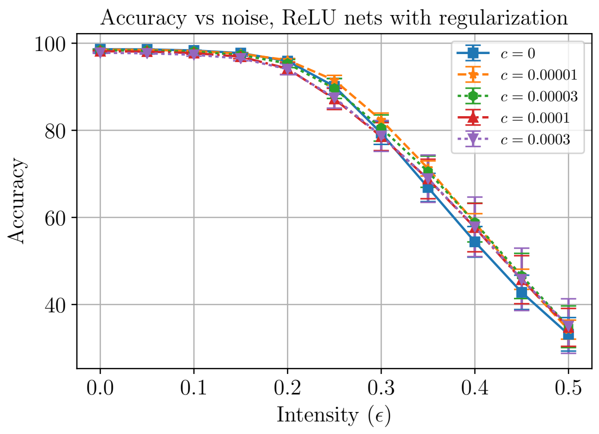


Figure 4: Performance of 512-512-512-10 ReLU networks trained for 30 epochs with an amount c of regularization. For $c \geq 0.001$, the ReLU network did not learn (performed no better than a random guess).

Network	$u_{\max}$	c	Sensitivity bound	
RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$	3	0	43.72	± 0.10
RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$	10	0	1,072.90	± 88.94
RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$	10	0.0001	231.90	± 28.01
ReLU(512,512,512,10)		0	275,296.24	$\pm 7,321.01$
ReLU(512,512,512,10)		0.0003	214.88	± 3.83
ReLU(512,512,512,10)		0.0001	383.30	± 7.83
ReLU(512,512,512,10)		0.00003	645.82	± 13.22
ReLU(512,512,512,10)		0.00001	985.57	± 11.41
Sigmoid(512,512,512,10)		0	1,872,558.46	$\pm 27,969.37$

Table 4: Sensitivity bounds, computed via (7) and (8) for different networks and training regimes.

between pixel value 0 and 0.3 by an amount $e^0 - e^{-(3 \cdot 0.3)^2} \approx 0.56$, and this seems sufficient sensitivity; indeed, the results of Section 6.3 confirm this. We may ask: would RBFI networks perform as well in settings where we do not have a good intuition for an upper bound for $\mathbf{u}$? The answer is yes, provided they are trained with a regularization that provides an incentive towards low $\mathbf{u}$ values.

We experimented with using $c\hat{s}$ as regularization loss, for a constant $c \geq 0$ and $\hat{s}$ as in (8). In Figure 5 we compare the robustness to PGD attacks of RBFI networks with u-bound 3, with that of RBFI networks with u-bound 10 trained without regularization and with $c = 0.0001$. We see that a small amount of regularization helps networks with higher u-bound to recover most of their robustness to attacks. We used PGD attacks, rather than FGSM or I-FGSM attacks, in the comparison, as RBFI networks are generally impervious to the latter. The regularization we used is fairly simple; it is likely that more sophisticated regularization would yield better results.

In Table 4 we report the sensitivity values computed via (7) and (8) for different networks. As we see, the bounds do not seem to be very useful in comparing the resistance to attacks of neural networks of different architectures. ReLU networks trained with regularization can have sensitivities in the few hundreds, comparable to those of RBFI networks, and yet they are very susceptible even to the simple FGSM attack, as shown in Figure 4.

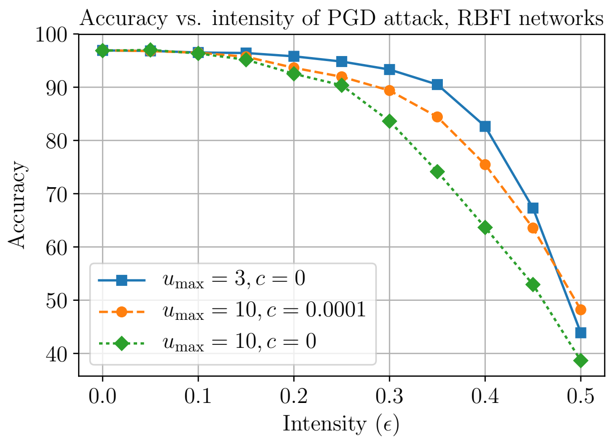


Figure 5: Performance of RBFI$(512, 512, 512, 10 \mid \wedge, \vee, \wedge, \vee)$ networks trained for 30 epochs, according to the $u_{\max}$ bound used for the $\mathbf{u}$ vectors, and according to the amount c of regularization used.

6.8 Deep RBFI Networks

Deeply non-linear models can be difficult to train in deep models consisting of many layers. To test the trainability of deep RBFI networks, we have experimented with training 128-128-128-128-128-10 networks for 20 epochs; for RBFI we used topology RBFI$(128, 128, 128, 128, 128, 10 \mid \vee, \wedge, \vee, \wedge, \vee, \wedge)$. ReLU and RBFI networks were easy to train: the former had $98.11 \pm 0.08\%$ accuracy, and the latter $95.53 \pm 0.13\%$ accuracy. At least with the methods we used (square-error loss and AdaDelta), which were the same for Sigmoid and RBFI networks, we were unable to train the Sigmoid network: the final accuracy was $11.24 \pm 0.34\%$, which is barely better than random.

6.9 RBFI vs. RBF Units

We have devoted the bulk of our attention to networks using RBFI units, rather than the more common RBF units defined by taking $\gamma = 2$ in (1). There are two reasons for this. The first is theoretical. The sensitivity of a single RBF unit is given by $s = \sqrt{2/e} \cdot \|\mathbf{u}\|_2$, and thus grows with the square-root of the number of unit inputs; this is better than the linear growth with ReLU or sigmoid units, but not as good as the essentially constant behavior of RBFI units. The second reason behind our preference for RBFI units lies in the fact that RBF units turned out to be harder to train than RBFI units, even when using our pseudogradient techniques; this was particularly evident when we tried to train deep networks. This surprised us, as we thought that the more regular nature of the norm-2 metric would have helped training, compared to the infinity norm.

7 Conclusions

In this paper, we have shown that non-linear structures such as RBFI can be efficiently trained using artificial, "pseudo" gradients, and can attain both high accuracy and high resistance to adversarial attacks.

Much work remains to be done. One obvious and necessary study is to build convolutional networks out of RBFI neurons, and measure their performance and resistance to adversarial attacks in image applications. Further, many powerful techniques are known for training traditional neural networks, such as dropout [SHK+14]. It is likely that the performance of RBFI networks can also be increased by devising appropriate training regimes. Lastly, RBFIs are just one of many conceivable highly nonlinear architectures. We experimented with several architectures, and our experience led us to RBFIs, but it is likely that other structures perform as well, or even better. Exploring the design space of trainable nonlinear structures is clearly an interesting endeavor.

References

[BL88a] David S. Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

[BL88b] David S. Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

[CCG91] Sheng Chen, Colin FN Cowan, and Peter M. Grant. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on neural networks*, 2(2):302–309, 1991.

[CW16] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. *arXiv preprint arXiv:1607.04311*, 2016.

[CW17a] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017.

[CW17b] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[CW18] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. *arXiv preprint arXiv:1801.01944*, 2018.

[GSS14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[KGB16a] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[KGB16b] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[MMS$^+$17] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. June 2017.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[NYC15] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

[Orr96] Mark JL Orr. *Introduction to Radial Basis Function Networks.* Technical Report, Center for Cognitive Science, University of Edinburgh, 1996.

[PGC$^+$17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[PMJ$^+$16] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium On*, pages 372–387. IEEE, 2016.

[PMW$^+$16] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[SHK$^+$14] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SZS$^+$13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv:1312.6199 [cs]*, December 2013.

[TKP$^+$17] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[TPG$^+$17] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv preprint arXiv:1704.03453*, 2017.

[Zei12] Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.