SAPIENZA
UNIVERSITÀ DI ROMA

Robustness Of Deep Neural Networks Using Trainable Activation Functions

Computer Science - Informatica LM-18

Corso di Laurea Magistrale in Computer Science

Candidate

Federico Peconi

ID number 1823570

Thesis Advisor

Prof. Simone Scardapane

Academic Year 2019/2020

Thesis defended on Something October 2020
in front of a Board of Examiners composed by:

Prof. Nome Cognome (chairman)

Prof. Nome Cognome

Dr. Nome Cognome

Robustness Of Deep Neural Networks Using Trainable Activation Functions
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: September 25, 2020

Author's email: peconi.1823570@studenti.uniroma1.it

Dedicated to
Donald Knuth

Abstract

This document is an example which shows the main features of the LaTeX 2_ε class `sapthesis.cls` developed by with the help of GuIT (Gruppo Utilizzatori Italiani di TeX).

Acknowledgments

Ho deciso di scrivere i ringraziamenti in italiano per dimostrare la mia gratitudine verso i membri del GuIT, il Gruppo Utilizzatori Italiani di TEX, e, in particolare, verso il prof. Enrico Gregorio.

Contents

Chapter 1

Introduction

1.1 Intriguing Properties of Neural Networks

Here we informally state the problem of adversarial attacks in ML models especially wrt to Neural Networks. Why is it of fundamental importance for the progress of the field from both practical (nns cant yet be deployable in critical scenarios for such reasons) and theoretical (Madry arguments around interpertability and robustness) perspectives

1.2 Smooth Activation Functions and Robustness

Recently a link has been proposed between activation functions and the robustness of Neural Networks (Smooth Adversarial Training). In particular, authors showed how they managed to improve the robustness by replacing the traditional Rectified Linear Units activation functions with smoother alternatives such as ELUs, SWISH, PReLUs

Building up from this result we thought we could find benefits by laveraging recently proposed smooth trainable activation functions called Kernel Based Activation Functions (Scardapane et al.), which already showed great results in standard tasks, in the context of adversarial attacks.

1.3 Structure of the Thesis

This paper studies the robustness of defenses to adversarial examples. Readers familiar with the relevant literature and notation (Szegedy et al., 2014; Carlini Wagner, 2017b; Madry et al. Athalye et al., 2018a) can continue with Section 3 where we describe our methodology. In this first part, the building blocks needed in order to understand the main arguments of the thesis are introduced. Even if, at first sight, the concepts that are going to follow may appear apart, the scope of this work is to make an attempt at discovering connections that might instead lie between them. For this reason, we will extensively introduce what Neural Networksis important that we cover the basicsDue to the deep and wide nature depth and the wide Pointers to more appropriate and detailed resources on the topics are given throughout

Description of the remaining chapters

Part I

Fundamentals

Chapter 2

Neural Networks

Broadly speaking, the field of Machine Learning is the summa of any algorithmic methodology whose aim is to automatically find meaningful patterns inside data without being explicitly programmed on how to do it. Well known examples are: Decision Trees (ref.), Support Vector Machines (ref.), Clustering (ref.), Neural Networks (Ref.) and, more recently, Deep Neural Networks (ref.). During the last two decades Deep Neural Networks have gained a lot of attention for their outstanding performances in different tasks like image classification (ref. ImageNet, over human level), speech and audio processing(ref).

2.1 Definition

Neural Networks (NNs) are often used in the context of Supervised Learning where the objective is to model a parametric function $f_\theta \colon \mathcal{X} \to \mathcal{Y}$ given n input-output pairs $S = \{(x_i, y_i)_{i=1}^n\}$ with $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$ such that

$$f_\theta \sim f \tag{2.1}$$

where f is assumed to be the real input-output distribution that we want to learn. In plain words, this means that we want to find the best set of parameters θ^* for the model such that, for any unseen input x_{new} we have that $f_{\theta^*}(x_{new})$ is as close as possible to $f(x_{new})$

For the sake of explanation, assume the input, which in practice can be very complex and unstructured e.g. made of: graphs, text, sounds, ecc, to be embedded in an input space $\mathcal{X} = \mathbb{R}^d$. The simplest form of a neural network is then given by

$$f_{W,b}(x) = \sigma(Wx + b) \tag{2.2}$$

where the parameters of the network are the elements of a $u \times d$ matrix W and a u-dimensional vector called bias. The last element σ applied at the end is a function which consists of a non-linear function acting element-wise and is the key component to introduce non linearity in NNs allowing them to model highly non-linear functions. We call it *activation function*. 2.2 can then be rewritten:

$$f_{W,b}(x) = [\sigma(W_1^\intercal x + b_1), \sigma(W_2^\intercal x + b_2), \ldots, \sigma(W_u^\intercal x + b_u)], \tag{2.3}$$

where W_i and b_i are respectively the i-th row of W and the i-th element of the bias.

Historically, the whole picture was somehow biologically inspired and had an intuitive explanation. Indeed, if we think at W as weights i.e. w_{ij} as the importance the model gives to the input x_i for how much it contributes to the $f_W(x)_j$-th component and define σ to be

$$\sigma(W_j^\intercal x + b_j) = \begin{cases} 1 & W_j^\intercal x \geq -b_j \\ 0 & W_j^\intercal x < -b_j \end{cases} \tag{2.4}$$

then it is easy to see that here the bias is acting like a threshold which discrminates between *activating* or not the j-th component depending on how much importance was given. Due to this analogy with the behaviour of neurons in the brain we call each component *neuron*, non-linearities activation functions and the whole model neural network.

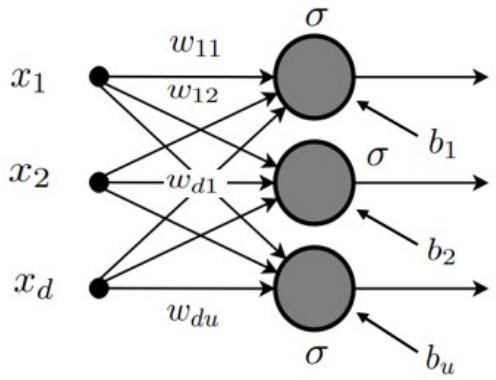


Figure 2.1. Graphic representation of a one layer NN also know as MLP(ref.)

In general, the idea of a layer of neurons can be recursively extended by stacking more layers together, all of which are described by a matrix of weights, a bias and an activation function and letting the output of one becoming the input of the subsequent. The resulting model is the mathematical composition of the layers, thus if we let L be the number of layers, $z_0 = x$ and $z_l = \sigma_l(W_l z_{l-1} + b_l)$ we write a L-layered $f_{\mathbf{W},\mathbf{b}}$:

$$f_{\mathbf{W},\mathbf{b}} = z_L = \sigma_L(W_L \sigma_{L-1}(\dots(W_2\sigma_1(W_1 z_0 + b_1) + b_2)\dots) + b_L)$$

With $\mathbf{W} = \{W_1, \dots, W_L\}$ and $\mathbf{b} = \{b_1, \dots, b_L\}$. We will call the first layer *input layer*, the middle layers *hidden layers* and the last layer *output layer*. This general but still basic form of Neural Network is known as *Feed Forward Neural Network* Fig. 2.2.

2.2 Training

There are still a couple of important pieces left to define to develop a properly working Neural Network. For instance, how are parameters computed? And in particular, with respect to what we compute them?

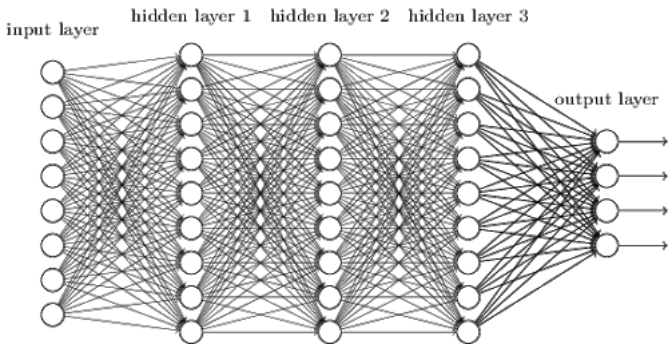


Figure 2.2. A Feedforward Neural Network with 3 hidden layers(ref. Michael A. Nielsen, Neural Networks and Deep Learning, Determination Press', 2015)

2.2.1 Loss Function

Before we realized that our goal is to maximize the approximation of the ground-truth input-output relation that lies under the data, therefore there is a need to introduce some metric to quantify this approximization. Call *loss function* $L(f_\theta(x), y) \colon \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ such metric. Common choices are (ref.):

- Least Square: $\|f_\theta(x) - y\|^2$ for regression tasks
- Binary Cross-Entropy: $y \log(f_\theta(x)) + (1-y)\log(1 - f_\theta(x))$ for binary classification
- Categorical Loss Function: $-\sum_{c=1}^{C} y_c \log\left(f_\theta(x)_c\right)$ for multicategory classification with C classes.

Intuitively, a good loss function will map bad approximations to high values and good approximations to smaller ones. Nevertheless, those are only point-wise estimates of the error, hence the best empirical solution learnable from the training set S would be

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} L\left(f_\theta(x_i), y_i\right) \tag{2.5}$$

which is called *empirical risk minimization.*

2.2.2 Gradient Descent

So far we have described, given an input x, how we can compute the output of a feedforward neural network by means of compositions of dot products and non-linear transformations between matrices, starting from the first to the very last of the layers in what is called a *forward pass.* As it turns out, to attempt to solve 2.5 we need to follow the exact opposite path. Indeed, every optimization algorithm used in practice makes use of the same subroutine called Backpropagation (ref.) introduced in the 1970s, which allows to compute, starting from the output layer and going backwards, the partial derivative of the loss function with respect to each weight in the network. Moreover it does so efficiently requiring only one *backward pass.*

Let $i^l = W_l z_{l-1} + b_l$ be the weighted input to the l-th layer, then the key observation is that the only way W_l can affect the loss function is by affecting

linearly the next layer which in turn affects its next layer and so on. In particular assume we add a little change Δi_j^l to the j-th element of i^l so that the neuron will output $\sigma\left(i_j^l + \Delta i_j^l\right)$, this change will eventually propagates in the network causing the overall loss LS to change by an amount $\frac{\partial LS}{\partial i_j^l}\Delta i_j^l$. For brevity, denote the gradient of the weighted input on the j-th neuron $\delta_j^l = \frac{\partial LS}{\partial i_j^l}$, then the following holds:

$$\delta_j^L = \frac{\partial LS}{\partial i_j^L} = \frac{\partial LS}{\partial z_j^L}\frac{\partial z_j^L}{\partial i_j^L} = \frac{\partial LS}{\partial z_j^L}\sigma'_L(i_j^L) \tag{2.6}$$

and equally, taking into account the whole output layer

$$\delta^L = \nabla_{z^L} LS \odot \sigma'(i^L) \tag{2.7}$$

where $\odot$ is the element-wise product and ∇_x the vector of the partial derivatives $\partial LS/\partial x$. That is, the gradient with respect to the weighted input to the last layer is given, using the chain rule, by the gradient with respect to the activation of the last layer times the derivative of the last activation function. Similarly, for any hidden layer l we note that:

$$\delta^l = \left((W^{l+1})^T \delta^{l+1}\right) \odot \sigma'(i^l) \tag{2.8}$$

When we apply the transpose weight matrix, $(W^{l+1})^T$, think intuitively of this as moving the previous layer's gradient backward, giving a measure of the gradient at the output of the l-th layer. We then take the product $\sigma'(i^l)$ which again moves the gradient backward through the activation function in layer l, giving the gradient of the weighted input to layer l.

By combining 2.7 with 2.8 we can compute the gradient δ^l for any layer in the network. We start by using 2.7 to compute on the last layer, then apply equation 2.8 to compute δ^{L-1}, then the same equation again to compute δ^{L-2}, and so forth, all the way back until the input layer. Since our intent is to retrieve the gradients for every weights of the network, we are left to show how δ^l relates to them, here we provide such relation without giving an explicit proof which instead can be found in many texts like (ref Nielsen chapter 2).

$$\frac{\partial LS}{\partial b_j^l} = \delta_j^l \tag{2.9}$$

$$\frac{\partial LS}{\partial w_{i,j}^l} = z_i^{l-1}\delta_j^l. \tag{2.10}$$

Remark how we already know how to compute each element on the right sides of these equations, moreover, given that the activation function and its derivative is efficiently computable, we will be able to efficiently get the seeked gradients in just one pass. It is worth mention that Backpropagation is actually a special case of a more generic set of programming techniques that go under the name of *Automatic Differentiation* (Ref.) to numerically evaluate the derivative of a function specified by a computer program. Such techniques are usually implemented in modern numerical libraries building variations of a data structure called *computational graph*. Well known examples are *Autograd* in *Pytorch* (Ref.) or *GradientTape* in *TensorFlow* (Ref.).

What does it mean to be able to compute partial derivatives of the loss? It means being able to understand where and how the loss decreases and thus we can exploit such information to find better and better weights solutions. This is the idea behind the Gradient Descent algorithm (Ref.). In particular, the gradient of a weight is nothing but the direction inside the weight-space where the loss function is increasing, therefore what we want to do is to follow the opposite direction. Formally, this translates in the following weight update rules:

$$w_l^t \rightarrow w_l^{t+1} = w_l^t - \frac{\eta}{n} \sum_j \frac{\partial LS_{x_j}}{\partial w_l^t} \tag{2.11}$$

$$b_l^t \rightarrow b_l^{t+1} = b_l^t - \frac{\eta}{n} \sum_j \frac{\partial LS_{x_j}}{\partial b_l^t} \tag{2.12}$$

where w_l^t are the values of the weights for layer l-th during the t-th pass, η is a small positive constant called *learning rate* chosen by the user accordingly and the gradients are averaged among all samples in the training set. Most importantly,

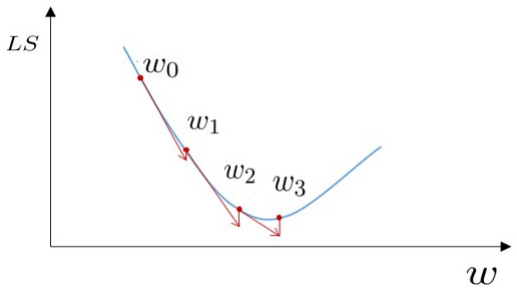


Figure 2.3. A 1-D representation of 4 gradient descent steps

note how we take the negative of the gradients, meaning that we are following the direction in which the loss decreases Fig. 2.3. The distance between two consecutives weights Δ_{w^t} well be directly proportional to both the learning rate picked and the averaged gradient.

In practice, however, very often we are deling with thousands or milions of data points, becomes unfeasable to compute every pass over the entire training set, thus what is done is to split the data into so called *mini-batches* and then apply the update rules on each mini-batch until we scanned the whole data. The entire scan is called *epoch* and the resulting algorithm Stochastic Gradient Descent (Ref.). Lastly, the size of a mini-batch is another hyperparameter that should be tuned by the developer, keeping in mind that the bigger the size the more stable will be our training the smaller the size the faster the training.

As stated earlier, neural networks can model extremely non-convex input-output functions therefore in principle there is no guarantee that Gradient Descent will find the optimal solution to 2.5 Fig. 2.4. Indeed, the optimal solution would be the global minimum of our weighted loss function but there is no apparent way for the algorithm to distinguish between global, local minimum or saddle points Fig. 2.5. However, it turns out that in practice Gradient Descent works fairly well once we correctly tune hyperparameters and run the algorithm from different initial values.

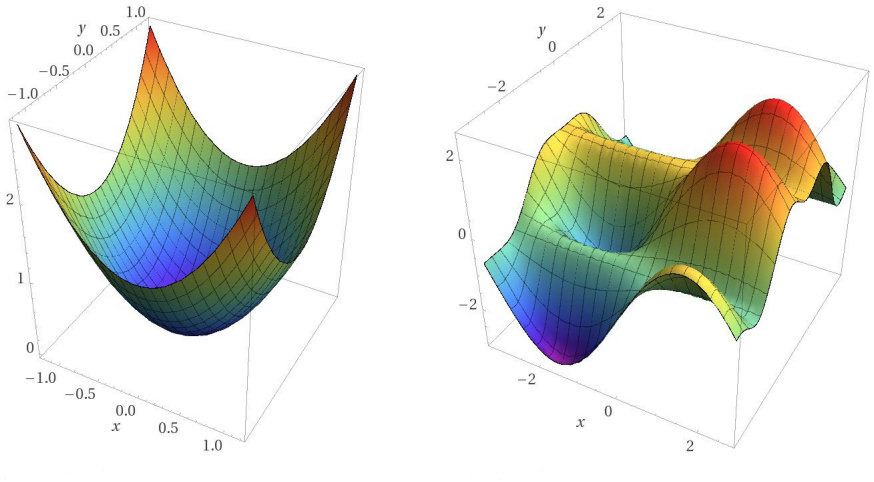


Figure 2.4. Convex and Non-convex optimization landscapes

(Ref .) Moreoever, lately authors have been proposed different methods to improve the convergence and the efficiency by smart changes of the learning rate during the training process (ref . cyclical learning rates)

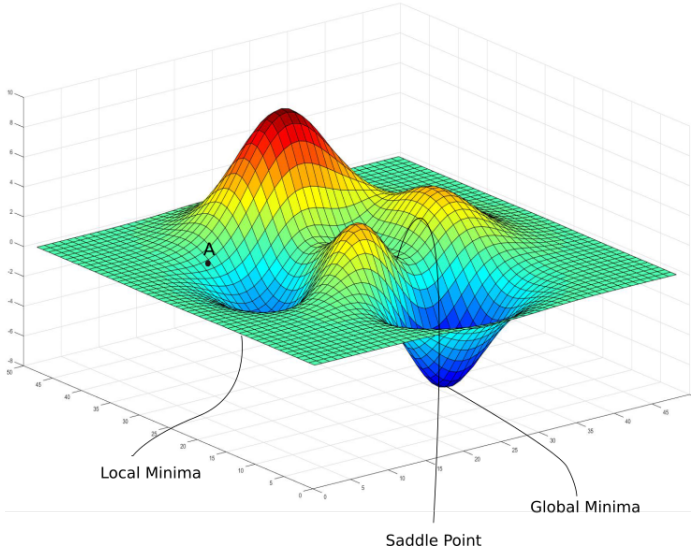


Figure 2.5. Visualization of global, local and saddle points. How can A reach the global minimum?

2.3 Activation Functions

We have seen how the learning process works optimizing the empirical risk by means of gradients computation. Therefore to be able to optimize anything, a neural network needs to have only differentiable components. However, before in 2.4 we

discussed the so called *step function*, an activation function that, even if biologically inspired and easier to justify, is not differentiable at the origin and the derivative is 0 elsewhere. Thus if we think again about how Backpropagation works, we see that employing such activation function would make the weight updates impossible since already δ^L would be either undefined or 0.

To deal with this issue, one of the first proposed activation functions was an approximation of the step function known as *sigmoid* (Ref.)

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{2.13}$$

which is differentiable everywhere with continuous derivatives (property that we will refer as *smoothness* through the chapters) and maps to $[0, 1]$ values. Nevertheless,

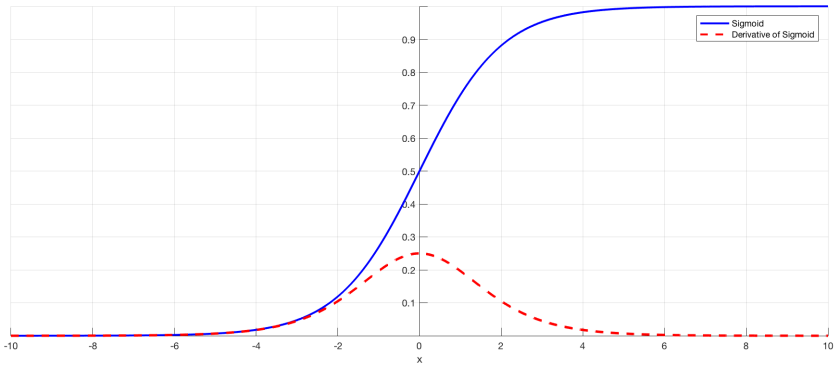


Figure 2.6. Plot of the Sigmoid function and its first derivative.

as the number of layers increases and the network becomes sufficiently deep, the sigmoid suffers from the *vanishing gradient* problem (Ref.) Fig. 2.6 due to its derivative being $[0, 0.25]$ bounded. Mainly for this reason it is not widely adopted in practice.

More in general, the sigmoid lies inside a class of activation functions known as *squashing* i.e. monotonically non-decreasing functions Σ that satisfy

$$\lim_{x\to-\infty} \sigma(x) = c, \quad \lim_{x\to\infty} \sigma(x) = 1. \tag{2.14}$$

For example, another kind of activation function of this type is the *hyperbolic tangent*, defined as

$$\tanh(x) = \frac{\exp\{x\} - \exp\{-x\}}{\exp\{x\} + \exp\{-x\}}, \tag{2.15}$$

which was find to allow for universal expressivness of nets (Ref.). However, as for the sigmoid, squashing functions tend to be prone to vanishing and exploding gradients (Ref.).

Nowadays, the most used activation function in neural networks for different applications is the *rectifier linear unit* (ReLU), first introduced in (Ref. Hahnloser et al. in 2000) and defined as the positive part of its argument

$$\text{ReLU}(x) = \max(0, x), \tag{2.16}$$

allows for efficient training and alleviates the exploding gradient problem (having derivative either 0 or 1), introducing only one point of non-differentiability. Moreover, it promotes *sparsness* in the network, which is usually beneficial (Ref.). One problem with ReLUs though is that the neuron's value, that get pushed to a big negative number, might stay stucked in 0 for essentially all inputs, in a so called *dead state*. If many neurons in the network die this can afflict the model capacity and can be seen as a form of vanishing gradient problem. To overcome this problem, a slighlty different activation functions can be used, called *leaky ReLU* (Ref.):

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} , \end{cases} \tag{2.17}$$

where $\alpha > 0$ is a user-defined constant usually set to small values such as 0.01. Even if this solutions solves the dying neurons problem, it does affect the sparseness property of ReLUs.

By definition, the mean of ouput values from a ReLU is always positive. *Exponential linear unit* (ELU) try to normalize their inputs:

$$\text{ELU}(x) \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp\{x\} - 1) & \text{otherwise} , \end{cases} \tag{2.18}$$

saturating negative values at a user-defined value $-\alpha$ which is usually set to 1. Conversly to ReLU and LeakyReLUs, the derivative is continuous therefore the function is smooth and for the negative values is defined as $ELU(x) + \alpha$.

Finally, one more recent activation function which gained a lot of attention is the *Swish* function (Ref.):

$$\text{swish}(x) = \text{sigmoid}(x) * x, \tag{2.19}$$

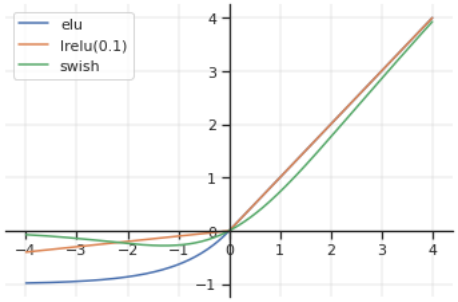


Figure 2.7. ELU, LeakyReLU($\alpha = 0.1$) and Swish functions plotted together. It can be seen that they mostly differ for negative values whereas behaving very similar to ReLU for positive arguments.

which again does look like another approximation of a ReLU Fig. 2.7, but in this case it manages in not loosing any useful property. Indeed, since it also saturates at 0 for negative values, it allows for sparsity and being smooth around 0 it helps reducing dying neurons. Lastly, around 0 negative values are kind of preserved which may still be relevant to patterns in the underlying data. Swish function proved to

consistently match if not outperform ReLU networks in different domains (Ref.), and more recent studies showed how this function can also help in training more *robust* networks (Ref. Smooth Adversarial training).

Along with this list of more 'traditional' activation functions, which we will call *fixed* activation functions, there is a whole branch of more sophisticated solutions, where the idea is to *learn* the function's optimal shape employing suitable parametric functions. Such functions can then be trained together with other weights of the net using backpropagation and gradient descent. Moreover, following the terminology introduced in (Ref Scardapane t al KafNets) we can again distinguish between two classes of this learnable activation functions: the so called *parametric activation functions* and *Non-parametric activation functions*. The former being usually a parametrization of a fixed activation function involving few constant parameters, whereas the latter is called non-parametric due to the number of parameters that can in principle grow without a bound and involves more complex shapes. At the end of this chapter we introduce a recently proposed class of non-parameteric activation functions, called *Kernel-Based Activation Functions* which will then be the main tool used in the following chapters to try to build more robust neural networks.

2.4 CNNs: Convolutional Neural Networks

In computer vision, in particular for image processing tasks, we can make the assumption that the input to the model will be an image. How well do feedforward neural networks adapt to such inputs? It turns out that we need to introduce several changes in the architecture in order to expect them to work properly. Take for example the famous dataset *ImageNet* which consists of more than 14 milions of images, all of which made of $256 \times 256 \times 3$ pixels. Every fully connected neuron in the first layer would have $256 * 256 * 3 = 196608$ weights, thus for a neural network with 1000 of such neurons, which is a very small number of units in practice, we would already need to train almost 200 milions of parameters, which requires a lot of resources. Therefore feedforward neural networks do not scale well to bigger images. More importantly, assume our task is to classify an image, from the point of view of such models, if we take an image x and perform a translation to, lets say, the right for few pixels, with high probability it will look like a completly different image from the point of view of the net and will probably be classified differently, even if from our point of view is clearly the same image. In some sense, there is no apparent way in which fully connected neural networks can take advantage of concepts such as *locality* or *translation invariance* that are intrinsic to images.

To circumvent these limitations, reasearchers have developed a specific architecture targeted for computer vision tasks called *Convolutional Neural Network* (CNN) (Ref.). In a standard CNN, every layer is 3-dimensional ($Width \times Height \times Dept$) to reflect the fact that we are always dealing with images and each neuron is connected only to a constant number of nearby neurons in the previous layer, shrinking down the number of total weights required. Layers can be either *convolutive layers* or *pooling layers* or *fully-connected layers*, the latter being a normal hidden layer.

A convolutive layer takes in input $W \times H \times C_{in}$ neurons from the previous layer and outputs $W \times H \times C_{out}$ neurons, where C_{out} is the number of filters used by

the layer. A filter F is a $K \times K \times C_{in}$ matrix of trainable weights, with $K > 0$ typically a small integer, which is used to compute a 2-dimensional activation map by sliding (convolving) across the width and height of the input. At each slice of input $\mathbf{x_{ij}} = (x_{ij}^1, x_{ij}^2, \ldots, x_{ij}^{C_{in}})$ that it touches, it computes the dot product $F^T X_{ij}$ where X_{ij} is the $K \times K \times C_{in}$ window centered at $\mathbf{x_{ij}}$. Intuitively, through backpropagation

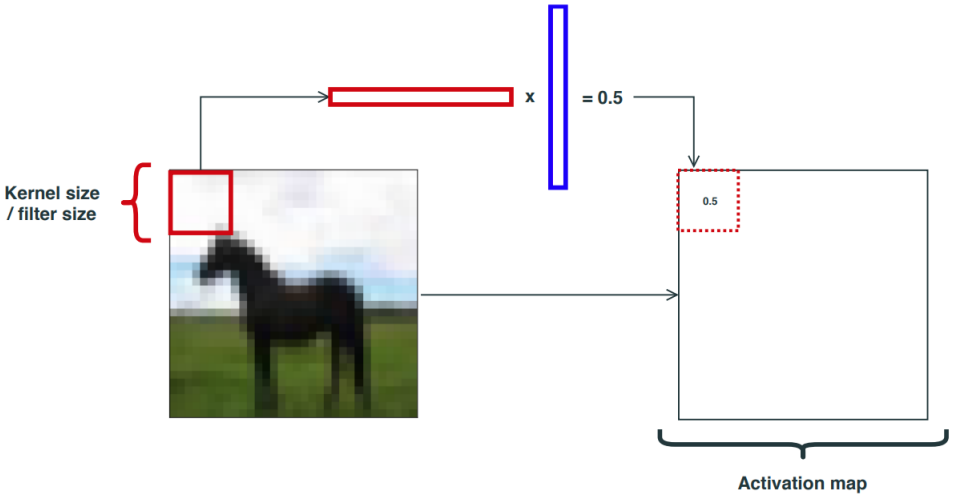


Figure 2.8. A convolution that produces the first element of the activation map for the given filter.

we learn filters tha are capable of recognizing specific shapes in the image, which we can see as features, starting from very concise ones in the early layers to more global ones towards the end layers as the receptive field gets larger(Ref.). Typically, as for normal neural networks, we apply a non-linear transformation after each convolutive layer and by stacking many of these Convolutional layers we get a convolutional network.

Going deeper in the network, as we learn global features, it might be convenient to reduce the width and the height dimensions. For this reason CNNs employ pooling layers that filters the inputs by some aggregation metric, such as average or max values. Similarly to a convolution, we specify a $K \times K$ window on which we apply the chosen metric. For example, let z^{l-1} be a $(64, 64, 12)$ dimensional input to a max pooling layer with window size 2×2, then, sliding again across width and height of z^{l-1}, we take the max value for each 2×2 input patch that we touch. The output will then be a $(32, 32, 12)$ dimensional vector of max valued neurons Fig. 2.9.

The complete architecture of a textbook CNN is a composition of 2 subarchitectures:

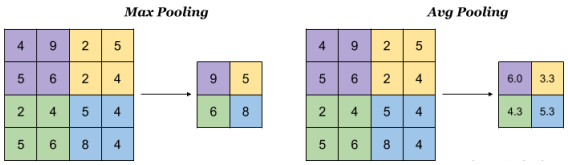


Figure 2.9. Max (left) and Average (right) pooling layer with 2x2 window size.

- A sequence of interleaved convolutive and max-pooling layers
- A *flatten* layer to reduce the last convolution to a 1-dimensional vector, followed by a sequence of fully connected layers to obtain the final score vector.

put together resulting in a *two-staged architecture*:

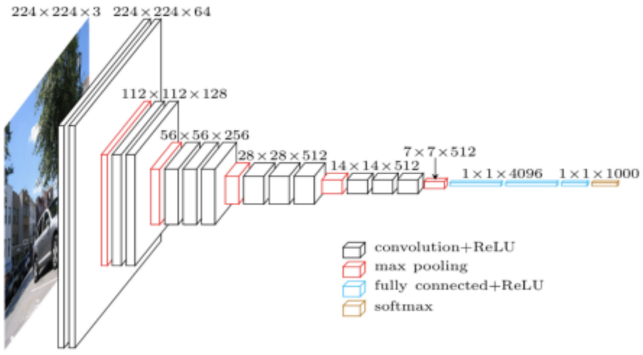


Figure 2.10. General Architecture of CNN for ImageNet (Ref.).

Even if we can already reach good accuracies with such a basic architecture, modern CNNs employ many smart variations to improve even more the performance, as we shall see in the next section.

2.5 From Neural Networks to Deep Neural Networks

Assume to develop a CNN as we have just seen to perform an image classification task. If the built network is sufficiently large, and the chosen dataset limited in number of samples, it might happen that our network will *memorize* the entire trainset instead of learning anything useful from it (Ref. UNderstanding DL requieres rethinking generalization). The described scenario is an infamous problem in learning theory and goes under the name of *overfitting*, i.e., instead of trying to learn the ground-truth distribution underlying the data, our model somehow tries to interpolate the training set, resulting in poor generalization capabilities. Early techniques to takle overfitting involve detection methods such as *early stopping* (Ref. On Early Stopping in Gradient Descent Learning) or basic prevention methods such as *regularization* (Ref. Regularization Theory and Neural Networks Architectures), where we try to penalize learning big-valued weights, which are syntoms of overfitting, by carefully adding penalization terms inside the optimization function.

Despite the fact that the presented techniques are very effective in practice and can help mitigating the problem, they are usally not enough for high perfomances. Another form of regularization can be induced performing *data augmentation* (Ref. AlexNet paper) which consists in virtually increasing the size of the train set applying , for each example in a mini-batch, one or more randomly sampled image transformation such as flipping, cropping, ecc. and then train on the resulting augmented trainset. Another idea to make the robust against slighlty perturbations hence more likely to generalize well is *Dropout* (Ref. Alexnet paper). Droput extends the idea of data augmentation to the network itself perturbing the hidden layers

instead of the inputs by randomly dropping some of the neurons. More formally, assume z^l be the output of a generic layer l, then applying Dropout to the ouput means replacing z^l during training with:

$$\hat{z^l} = z^l \odot m, \tag{2.20}$$

where m is a binary vector with entries taken from a Bernoulli distribution with probability p. It is important that Dropout gets applied only during training whereas at inference time the output of the layer is replaced with its *expected* training value:

$$\mathbb{E}[\hat{z^l}] = p \cdot z^l. \tag{2.21}$$

Both data augmentation and Dropout were key components of *AlexNet*, the first CNN to win an image classification contest by a big margin (Ref. AlexNet p).

In 2014, the Oxford's Visual Group realized that they were able to reach better performances than AlexNet by stacking *blocks* of layers instead of many single layers one after the other. In particular, they proposed a block made of multiple convolution layers with 3×3 kernel size and same number of filters, followed by a 2×2 max-pooling, periodically doubling the number of filters for deeper blocks (Ref. VGG). The resulting architecture, known as in literature as *VGG*, was however considerably demanding in terms of resources and this drove researchers to look for solutions that matched the performances whereas decreasing the number of weights. Such goal was achived soon after with *GoogleNet* which made use of two novel modules inside the network: the *inception block* and the *global average pooling* (Ref. GoogleNet). The former being the first attempt at process in parallel the same input with different levels of granularity, somehow allowing to embed multiple layers within a single one Fig. 2.11 , and the latter used as a substitute for the flatten

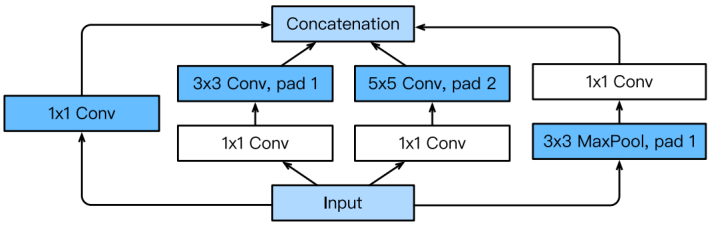


Figure 2.11. Inception Block overview, 1×1 convolutions are used to reduce the number of filters lowering complexity. Source: Dive into Deep Learning, chapt. 7.4

layer by taking the average value in each channel and then vectorizing them into a 1-dimensional vector. This last step drastically reduces the number of weights needed in the second stage of the CNN.

Less than one year later, another breakthrough technique was developed: the *Batch Normalization* (BN), a simple heuristic that allowed to train deep neural nets significantly better (Ref.). BN works by normalizing and learning to scale the mean and the variance of a layer's output in the following way: consider $i_1, i_2, \ldots, i_B$ to be the values of a generic given neuron during a mini-batch. Then with Batch Normalization, we first normalize them by:

$$i_j = \frac{i_j - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{2.22}$$

with μ and σ being respectively the mean and the variance of the mini-batch values. Then, we rescale them by:

$$i_j = \alpha i_j + \beta, \tag{2.23}$$

where α and β are trainable parameters computed with respect to every neuron in the layer. Nowadays, it is believed that the reason behind the effectiveness of BN is likely due to its effect on the optimization landscape (Ref.), which gets smoothed, hence the speed-up in convergence and better generalization properties.

Having developed tools that bypass exploding and vanishing gradients, that allow for faster convergence and better generalization, one may be tempted in see what happens when we keep stacking more and more layers. After all, the intuition we gained from the general trend in CNNs is that the deeper the network, the better the learning. However, this belief was not matched by experiments. Indeed, for very deep straightforward neural networks, we are likely to experiment a *degradation*(Ref. ref made by resnetpaper) of accuracy perfomances which is not caused by overfitting i.e. it leads both to higher test and training error Fig. 2.12. To approach the problem

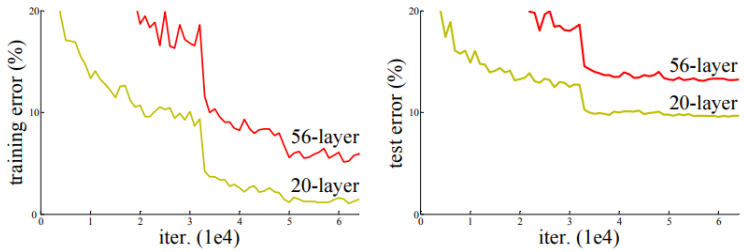


Figure 2.12. Source: Deep Residual Learning for Image Recognition

we can start with the following remark: if we assume a shallow network $\mathcal{F}(x)$ for a given task reaches an accuracy a, then by adding identity layers on such network i.e. layers implementing the identity function, will result in a deeper network with again accuracy a, it can't get much worse. Building upon this argument, authors in (Ref. ResNet paper) showed that a deep network will rather learn a better mapping starting from $\mathcal{F}(x) + x$ than from $\mathcal{F}(x)$. For this reason, they introduce the idea of *skipping connections* or *residual connections* where, as the name suggests, we link the input of an earlier layer to the output of a deeper layer, skipping over the layers in between Fig. 2.13. If x has different dimensionality, we can rescale it using a

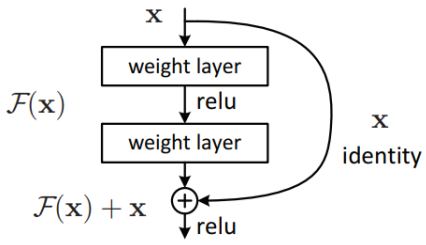


Figure 2.13. Source: Deep Residual Learning for Image Recognition

1×1 convolution. A neural network that makes use of many residual connections is called *ResNet* Fig. 2.14.

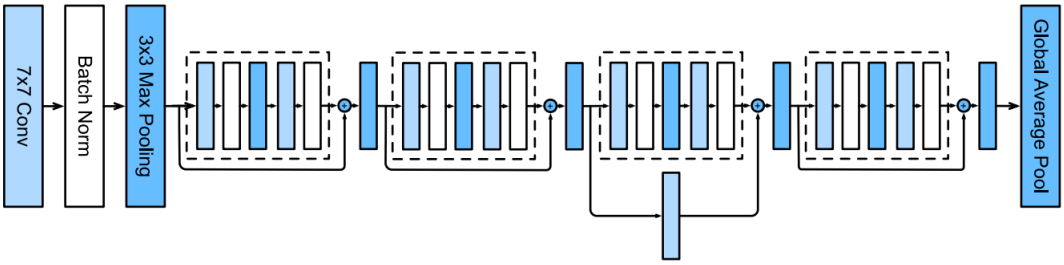


Figure 2.14. Source: Dive Deep into Deep Learning

Thanks to the methodologies introduced so far, we are capable of building non-degenerative CNNs that scale up to hundreds of layers and are currently state-of-the-art in many domains.

Chapter 3

Adversarial Examples Theory

In recent years, AI-based systems are finding an ever growing number of applications in the industry, ranging from medical, multimedia, telecomunications, to even military, political and legal sectors. As a consequence of the importance of such systems to the modern world, it is currently reasonable to think that they might become potential threats to the eyes of malicious agents such as hackers, business rivals as well as governments, which may seek to circumvent them. For simplicity, we name any of these malicious entity: *adversary*. Inside academia, there is a vast literature on the topic and many attacks and defenses have been devised by researchers towards different types of intelligient systems for both supervised (Ref. Intriguing Properties) and unsupervised models (Ref. s data clustering in adversarial settings secure?) with no exception for Neural Networks. Even better, since, as we have seen, DNNs reach best performances among ML systems in many applications, recent research efforts are especially directed in assessing the *robustness* of DNNs. In this thesis we will stick to the same trend.

In the context of classification, one of the many distintions that we can make about types of attacks is wether the objective of the attack is to just fool the classifier making it mislabel a given sample x, or targetting the classification towards a specific class $\hat{y}$ where, given a sample-label pair (x, y) with $\hat{y} \neq y$, our classifier will be tricked in believing that the correct classification is indeed $\hat{y}$. We name this two different scenarios *indiscriminate* and *targeted* attacks respectively. Moreover, despite the fact that the attack is targeted or not, we define three inherently different attacks types against a classifier:

- *Data Poisoning:* here the adversarial introduces *poisoned examples* into the data. Poisoned examples can either be mislabeled examples, with the examples correctly belonging to the domain space described by the data, or they can be anomalies for the domain. For instance, if data describes birds, a ship should be considered very odd and thus poisoned (Ref.).

- *Reverse Engineering:* usually crafted against rule-based classifiers, such attack consists in querying the model to retrieve sensible information about its decision rule or the data on which it was trained (Ref.).

- *Test Time Evasion:* as the name suggests, here the attack is performed at test time by a careful *perturbation* of the the sample in a way that the

transformation is neither human percetible nor easily detected by the system, but as powerfull that the classifier's decision now disagrees with a human consensus (Ref. Explaining and harnessing adversarial examples).

As shown for the first time in (Ref. Intriguing Properties of..), DNNs are drastically prone to Test Time Evasion attacks and, more importantly, unaware networks can easily be fooled in a matter of few lines of code by anyone who knows the basics of any modern deep learning framework. For this reason, we will dedicate this section to a formal introduction of the problem, and the rest of the thesis in the development of a new approach that aims to improve the resiliency - in jargon, *Robustness* - of Neural Networks against today's Test Time Evasion attacks.

3.1 Another Optimization problem

To get a Test Time Evasion attack working, any adversary needs to know the true class of the input he is manipulating. Indeed, the perturbation needs to be done in such a way that the resulting perturbed input will cross the true class decision region, in the output space of the model, to move to another decision region (which is specific to the targeted category in case of a targeted attack). However, due to an high number of weights and non-linearities involved in a forward pass, we usually don't know how DNNs actually make their predictions, instead we delegate the job of learning how to make decisions to Gradient Descent during training. Therefore, how does an adversary learn how to craft such untangible yet precise perturbations? Well, he relies again on Gradient Descent, more precisely, on back propagation. Recall that, in the previous section, we learned how to compute the gradient $\frac{\partial LS}{\partial w^l_{i,j}}$ with respect to any weight $w^l_{i,j}$ of the network, nevertheless, nothing prevents us to push even further automatic differentiation and compute the gradient of the loss with respect to the input x with just as much effort. This quantity will tell us how small changes to the image itself affect the loss function.

Since the goal of the adversary is to make the classifier mislabelling the input, and since we can optimize a function with respect to the input, we can devise an indiscriminate attack by just solving the following optimization problem:

$$\max_{\hat{x}} \mathrm{LS}\left(f_\theta\left(\hat{x}\right), y\right), \tag{3.1}$$

where $\hat{x}$ is called *adversarial example* and is nothing else that an approximation of the original input x. However we also need to characterize the fact that $\hat{x}$ must be very close to x. In fact, with this settings we could simply transform completly the input to make it equal to another input x' which belongs to a different class and would still be a valid solution. But this is clearly in contrast with the principle of being an human imperceptible perturbation! Thus denote $\delta \in \mathcal{X}$ to be the perturbation applied to the input $\hat{x} = x + \delta$, then the adversary will actually want to solve:

$$\max_{\delta \in \Delta} \mathrm{LS}\left(f_\theta\left(x + \delta\right), y\right), \tag{3.2}$$

where Δ denotes the set of any admissible small perturbation. Again, this is something we cannot implement straightaway since it is not clear from a mathematical

perspective how to explicitly construct the set of all valid small perturbations, even if this is what the adversary is ideally trying to achieve. In practice, what is done is to stick to some mathematical metric such as a specific norm for real vector spaces. For example, an effective metric which allows to fool many NNs, even with super small perturbations is the L_∞ norm. The L_∞ norm for a generic vector $z \in \mathbb{R}$ is defined to be:

$$\|z\|_\infty = \max_i |z_i| . \tag{3.3}$$

Thus the space of allowed perturbations becomes:

$$\Delta = \{\delta : \|\delta\|_\infty \leq \epsilon\}, \tag{3.4}$$

where ϵ is the size of the biggest perturbation allowed, i.e. if for example we are dealing with images, any pixel will be $[-\epsilon, \epsilon]$ perturbed and if ϵ is chosen sufficiently small, the resulting image will be visually indistinguishable to the original one. However, other norms such as L_2 are also very common.

How do we perform targeted attacks within this framework? Intuitively, the adversary will want to minimize the loss with respect to the targeted class but at the same time, he also wants to be sure that the network will give the smallest confidence to the correct class. This translates into:

$$\max_{\delta, \|\delta\|_\infty \leq \epsilon} (\mathrm{LS}(f_\theta(x+\delta), y) - \mathrm{LS}(f_\theta(x+\delta), y_{target})). \tag{3.5}$$

3.1.1 Fast Gradient Sign Method

For the sake of discussion, we will now see how to actually solve the proposed maximization problems, restricting ourself to indiscriminate attacks, since the same solutions will also work painlessy for the case of targeted attacks.

In general, the basic idea behind every adversarial attack is to use Gradient Descent to maximize our objective until we converge towards a satisfying solution δ^*, just as we did when we were training the network. Furthermore, in this case, we also have to take into account the bounds on the perturbation, which can be implemented by a projection to the $[-\epsilon, \epsilon]$ norm-bounded space. In order to maximize loss, we want to adjust delta in the direction of this gradient, i.e., take a step:

$$\delta^{t+1} = \delta^t + \alpha \cdot \nabla_{\delta^t} \mathrm{LS}\left(f_\theta(x+\delta^t), y\right), \tag{3.6}$$

for some step size α. Then, we clip δ^{t+1} to ensure the norm constraints, so in the case of L_∞-norm:

$$\delta^{t+1} = \mathrm{clip}(L_\infty, \delta^{t+1}, [-\epsilon, \epsilon]), \tag{3.7}$$

where clipping acts by projecting δ^{t+1} back to the ϵ-bounded L_∞ ball it moved outside.

Now, if we want to climb the slope of the loss as much as possible we will want to take a very large step size. However, by doing so, we are probably going to stick out the L_∞ ball and thus our delta will be either be projected to ϵ or $-\epsilon$ with high probability. Based on this principle, one of the first proposed attack simply considered the following update rule for delta:

$$\delta = \epsilon \cdot sign(\nabla_x \mathrm{LS}\left(f_\theta(x), y\right)), \tag{3.8}$$

and is known as the *Fast Gradient Sign Method*(FGSM) (Ref.). FGSM is a single step attack and works on the assumption that, in a very close neighbourhood of x, a DNN can be approximated with the behaviour of a linear model. Consider a simple linear model $g(x) = W^T x$, then $g(x + \delta) = W^T x + W^T \delta$ and thus, if we want to maximize the effect of the perturbation $|g(x) - g(x + \delta)|$ under $\|\delta\|_\infty \leq \epsilon$ we better define $\delta = \epsilon \cdot sign(W^T)$. Even if the per-component shift is small, the overall shift can increase way more if x is high dimensional.

3.1.2 Projected Gradient Descent

The idea that the effect of any perturbation inside the $\|\delta\|\infty \leq \epsilon$ ball can be approximated, if not upperbounded, by taking the perturbation at the boundary which is given by the direction where the loss is most increasing might be a little too strong as an assumption. Indeed, as previously seen, the optimization landscape is often extremely non-linear, even for very small neighbourhoods, thus, if we want a stronger attacks, we likely want to consider better methods at maximizing the loss function than a single projected gradient step.

A more effective and natural approach comes from iterating 3.6 many times with small step sizes and projecting back whenever needed. The general algorithm is called *Projected Gradient Descent* (PGD)(Ref.):

procedure PGD(α, ϵ)
  $\delta \leftarrow 0$
  for $i \leftarrow 1, n$ **do**
    $\delta \leftarrow \delta + \alpha \cdot \nabla_\delta \mathrm{LS}\left(f_\theta(x + \delta), y\right)$
    $\delta \leftarrow \mathcal{P}(\delta)$
  end for
  return δ
end procedure

Where $\mathcal{P}$ denotes the projection for the specific metric used. Nowadays, PGD, or slighlty variations of it are standard methods when it comes to evaluate the robustness of a network.

As for Gradient Descent, PGD is still limited by the possibility of getting stuck inside local maximum of our objective 3.2. Mitigations of the problem might arise adding random restarts, i.e., running PGD multiple times from randomly picked starting deltas (within our norm restricted ball). It is infact important to note that it is likely that there are local optima which will be found if we start with $\delta = 0$ and that could be avoided with randomization. Conversly, running multiple PGDs increases the runtime by a factor proportional to the number of restarts and it might not be practical in real world scenarios, especially when it is used as a subroutine to train robust models (Ref. Free Adversarial Training).

3.1.3 White, Grey and Black Box Attacks

In the aforementioned attacks, various implicit assumptions are made about the knowledge of the adversary. Biggio et al. in (Ref. Security evaluation of pattern classifiers under attack) , in alignment with the wider field of modern Cryptography, have advised making such assumptions explicit also for publications concerning the

security of ML models. In particular, in any of the previously introduced attacks such as FSGM or PGD, we let the adversary the possibility to exploit the gradients and thus the entire model (peraphs with some hyperparametes excluded) to perform the attack. Such described scenario, where there is full knowledge about the resources and the model of the defender, is known as a *White Box* attack. Moreover, note as this scenario should be the preferred one since it allows us to devise more effective defenses and is actually compliant with the basic principle of "not doing security by obscurity". On the contrary, no knowledge of the classifier results in *Black Box* attacks. Several attacks were nevertheless devised even in such conditions (Ref.). A more realistic scenario though involves the so called *Grey Box* attacks (Ref. Adversarial examples are not easily detected: Bypassing ten detection methods) where the adversary might have knowledge of the model but it has no direct access to the training set that was used to train the classifier and another surrogate classifier is trained using surrogate data (Ref. Survey on current defenses).

3.2 Defenses

Although one may be tempted in the quest for a better understanding of the problem of Adversarial Examples to shed some light on profound questions like: why such brittleness of DNNs exists in the first place, what is that it takes to develop environment resilient intelligent systems, how are adversarial examples and the interpretability problem related to each other (Ref. Madry) and so on, it is undeniable that the security concerns are, from a practical perspective, the most immminent ones, since, as the integration of ML applications becomes more and more present in the modern world, these issues are starting to threaten several sectors. An attack may, for example, fool an autonomous veichle which is trying to recognize a road sign (Ref.), cause a drone to falsely target a civilian (Ref. “Cooperative unmanned aerial vehicles with privacy preserving deep vision for real-time object identification and tracking,), or grant authentication to illegitimate people for entering buildings, systems (Ref.), ecc. . Therefore, is no surprise that, as soon as Adversarial Examples were first discovered, researchers started to come up with many different ideas on how to defend ML models against adversaries. In the following, we are going to briefly describe some of the most promising attempts that were recently made to develop more robust as well as performant DNNs.

3.2.1 Detection Methods

Detection methods usually involve attaching a 'patch' (or detector) to the original network that we want to make robust. The overall network then gets trained, on both original and perturbed samples if the detection is supervised, and, with some strategy implementation, the detector learns how to spot adversarial examples from normal ones.

In case of a supervised detection mechanism, we want to augment our data with labelled samples crafted with known attacks and build a binary classifier which is able to distinguish between vanilla and altered inputs. The key point here is then to test such classifier on new, previously unseen attacks and check how well it adapts. Papernot er al. in (Ref. On the (statistical) and Feinman et al. in (Ref. Detecting

adversarial samples from artifacts) developed two sophisticated early versions of such detection type, but both failed to detect ad-hoc CW (Ref.) attacks on CIFAR-10. Better results against CW were achieved by Metzen *et al.* (Ref. On Detecting Adversarial Perturbations), whose method works by feeding DNN layer's activations as features to a detector Fig. 3.1. In particular, in detecting CW on CIFAR-10 were 81% TPR and 28% of FPR. However they reported that they were not able to generalize well to other attacks, which shows a limitation of their method (or even to any supervised detection?), that is, to likely overfit on the attacks used for training (Ref. BYpassing 10 Detection Methods).

In literature, as well as supervised detection mechanisms, there have been also many detection classifier which were not trained on adversarial examples, thus performing sort of unsupervised detection. They instead rely on explicit null hypothesis and statistical models to work, such as based on PCA (Ref. Early methods for detecting adversarial images), which again however showed uneffective against CW for CIFAR-10, (Ref. Towards open set deep networks), or analyzing the joint density of a DNN's layer feature vector (Ref. Detecting adversarial samples from artifacts). In particular, based on the latter, recently Miller *er al.* in (Ref. Anomaly detection of attacks (ADA) on DNN classifiers at TEST TIME,) managed to develop the current state-of-the-art for detection methods as stated in (Ref. 2020 Survey on Defenses.). The description of such method is however quite involving and esulate from the objective of this thesis.

3.2.2 Robust Optimization and Adversarial Training

What if we train a DNN such that, together with minimizing the loss we also train to minimize the effects of adversarial examples? That is, how do we go and train a DNN which performs well on both clean and perturbed inputs? As it turns out, to perform such training, we need to solve the following, intuitive, min-max problem:

$$\min_{\theta} \frac{1}{|S|} \sum_{x,y \in S} \max_{\delta \in \Delta} \mathrm{LS}\left(f_\theta(x+\delta), y\right), \tag{3.9}$$

that goes under the name of *Robust Optimization.*

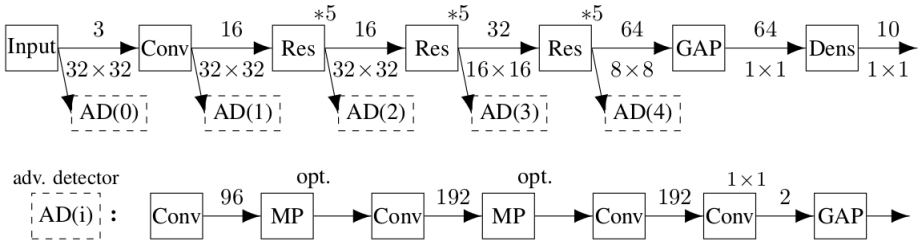


Figure 3.1. Metzen *et al.* (Ref.) used a ResNet as original classifier, plus dections is made thanks to many interleaving detectors between each residual block. Each detector is implemented as a DNN that learns how to spot the presence of an attack by looking at the activations layer's distributions during training.

The order of the optimizations is important here. The maximization is inside the minimization, this intuitively means that we are training in a way that: even if the adversary knows the parameters of the model θ and performs his best attack over it, we contrast its effects by minimizing the empirical risk on such attack, as with standard training. Moreover, notice that we just learned how to compute strong attacks e.g. with randomized PGD, thus we can already go and implement Robust Optimization, which, to be precise, it is mostly reffered as *Adversarial Training* (AT) when we approximate the solution instead of computing it exactly (Ref. https://arxiv.org/pdf/2007.00753.pdf).

In practice, even if the training is performed employing random PGD to compute the inner maximization, i.e., a very specific form of attack, it does generalize well to other attacks (Ref. Madry article in Robust Optimization), provided that we consider attacks under the same metric. Indeed, there is no real guarantee that AT done under, say, $\|\cdot\|_2$ will result in a model also robust against $\|\cdot\|_\infty$ based attacks. To achive defenses against multiple metrics, we need to incorporate multiple attacks under the inner maximization, however, as previously discussed, characterizing a priori every possible metric of perturbation seem to be a difficult task, hence the complexity of devising an universal defense mechanism.

The main drawback with AT, which is so far regarderd as the most effective method developed agains adversarial Examples (Ref.), is its computational demand. Indeed, due to the double optimization that needs to be computed for each weight update on each sample, it requires a lot of resources, especially when it comes to large networks and large datasets. For this reason, lately, different works tried to takle the problem of speeding-up AT, proposing approximations and variations of it (Ref. Free AL)(Ref. Fast is better than free).

3.2.3 Provable Robustness

Provable defenses try to theoretically find certificates in distances or probabilities to certify the robustness of DNNs. Can 3.9 be exactly solved? Namely, can we find the optimal set of weights such to minimize the error on Adversarial Examples? This is in principle a legitimate question to ask, and several strategies have been proposed, all of which somehow works providing an upperbound for the inner maximization. In this way, defined the threat model, we can make stronfer statements about the guarantees for the defense. If we rewrite the innner maximization as the following adversarial loss:

$$\mathcal{L}_{adv} = \max_{\sigma\in\Delta} \left\{ \max_{i\neq y} f_\theta(x)\,(x+\delta)_i - f_\theta(x)\,(x+\delta)_y \right\}, \tag{3.10}$$

then, if we are capable to define an always larger certificate $C(x, f_\theta) > \mathcal{L}_{adv}$ and prove

$$C(x, f_\theta) < 0 \tag{3.11}$$

under certain constraints, the we are sure that, under the same constraints, e.g., on the bound of the perturbation, our model will always predict the correct class. Employing this argument, [112] transforms the problem into a linear programming problem and [111] derives the certificate using semidefinite programming.

Differently, !!Pippo!! *et al.* (Ref. Intriguing) pointed out the relation that exists between the so called *Lipischitz Constant* and the sensitivity of the network with respect to input perturbations. Specifically, denote $f_\theta(x) = f_{\theta_L}(f_{\theta_{L-1}}(\cdots(f_{\theta_1}(x))\cdots))$ then the Lipschitz Constant of $f_{\theta_i}(x)$ with respect to the norm $\|\cdot\|_p$ is defined to be the smallest L_i such that, for any $x, r \in \mathcal{X}$:

$$\|f_{\theta_i}(x) - f_{\theta_i}(x+r)\|_p \leq L_i \|r\|_p. \tag{3.12}$$

Notice how L_i by definition is an upperbound for the sensitivity of layer i with respect to any perturbation r. Following the composition property of the Lipschitz Constant, the overall network Lipschitz Constant will be, the smallest L such that:

$$\|f_\theta(x) - f_\theta(x+r)\|_p \leq L \|r\|_p, \tag{3.13}$$

where $L = \prod_{i=1}^{L} L_i$. It is important to note how this is only an upperbound, thus a conservative measure of the possible unstability of the network. For this reason, no conclusion about the existence of Adversarial Examples can be derived even from large Lipischitz Constants. We are however guaranteed that for very small Lipischitz Constants no Adversarial Example will exist. This is the idea behind many regularization techniques that seek to penalize Lipschitz bounds on networks's components as in (Ref.) (Ref.)

Many other approaches try to provide certificates such as *Randomized Smoothing*, *estimation of the lower bound* (Ref. https://arxiv.org/pdf/2007.00753.pdf) . However, provable Robustness struggles to scale up to real-world applications due to the complexity of computing such bounds that are usually based on generally intractable methods or ad-hoc methods.

Chapter 4

Non-Parametric Activation Functions

As anticipated shortly in the section concerning Activation Functions, it is possible to increase the flexibility of a Neural Network replacing a fixed activation function with a parametrized, differentible, non-linear function. These transformations can in fact be trained along with the remaining weights of the network allowing each neuron to model its own optimal shape.

Scardapane *et al.* in (Ref. KafNets) grouped together different works on such activation functions distinguishing, on the high level, between the number of parameters they involve in their formualtions. In particular, whenever we add a constant number of weights to a fixed activation function, authors say we are dealing with 'parametric activation functions'. Some examples of this class of functions are: the *Generalized Hyperbolic Tangent* (Ref.) , a parametric Leaky ReLU introuced by He *et al.* (Ref.) or the more flexible S-shaped Relu (SReLU) (Ref. Jin et al.):

$$\text{SReLU}(x) = \begin{cases} t^r + a^r (x - t^r) & \text{if } x \geq t^r \\ x & \text{if } t^r > x > t^l \\ t^l + a^l \left(x - t^l\right) & \text{otherwise} \end{cases} , \tag{4.1}$$

parametrized by $\{t^l, a^l, t^r, a^r\}$. Depending on the values assumed by the left (l) and right (r) parameters, SReLU can assume both convex and non convex shapes.

What happens if we give to activation functions greater modeling capabilities, what if we allow them to model any continuous segment? This question is instead addressed by the class of 'non-parametric activation functions'. These methods, usually introduce a further global hyper-parameter, allowing to balance the number of parameters that can in principle grow without a bound, hence the name.

In this section, we give an overview of three early proposals, describing the general idea and some of the drawbacks, if any. After that, we focus on a more recent kind of non-trainable activations called kernel-based activation Functions (KAFs), highlighting their properties and experimental results.

4.1 Adaptive Piece-Wise Linear Activation Functions

Introduced in (Ref. Agostinelli et al 2014), APL generalize the SReLU 4.1 activation function summing up S parametrized linear segments that are learned under the constraint that the resulting function is continuous:

$$\mathrm{APL}(x) = max\{0, x\} + \sum_{i=1}^{S} a_i \max\{0, -x + b_i\}, \tag{4.2}$$

where S is a user-defined value and a_is the parameters to learn. APL adds $2S$ new parameters per neuron i.e. introducing a linear number of parameters to the overall network, which is often feasable. APL cannot, however, model or approximate all piece-wise linear functions, but only saturating ones. Moreover, APLs introduce S non-differentiable points which may harm backpropagation.

4.2 Spline Activation Functions

If we want an activation function capable of interpolating S given points, we can devise the following polynomial interpolation:

$$\sigma(x) = \sum_{i=0}^{S} a_i x^i, \tag{4.3}$$

where we actually going to learn $S + 1$ coefficients to pass through the desired S points. Thus, in theory, we can at least approximate, by means of a sufficiently large S, any smooth function albeit in practice, due to the global effect that any parameter has on the global shape, such approximation is hard. Moreover, x^i can easily grow too much and encounter numerical problems.

Instead of using polynomial interpolation, (Ref.) proposed the use of *spline interpolation* that results in the so called spline activation functions (SAFs). Let $\{x_1, x_2, \cdots, x_S\}$ be an equally spaced sampling of real values, symmetric to the origin and with step size Δx and call *knots* the corresponding y-valules $\{\mathrm{SAF}(x_i)\}_{i=0}^{S}$. Denote $u = \frac{t}{\Delta x} - \lfloor \frac{t}{\Delta x} \rfloor$ to be the normalized ascissa value between to consecutive knots when the activation is t. Finally, we can define SAF at t:

$$\mathrm{SAF}(t) = \mathbf{u}^T \mathbf{B} \mathbf{q}_k, \tag{4.4}$$

where $\mathbf{u} = \left[u^P, u^{P-1}, \ldots, u^1, 1\right]^T$, P is a user-defined value (usually chosen to be 3), $\mathbf{q}_k$ the vector composed by the closest knot to t and the P rightmost neighbors knots and $\mathbf{B} \in \mathbb{R}^{(P+1)\times(P+1)}$ the *spline basis*. Different basis give rise to different interpolation schemes (Ref.).

Good news with SAFs is that each knot has only a local effect on the overall shape. Therefore, with respect to the initialized knots, their training allows for faster and better convergence to the optimal. Moreover, as with polynomial interpolation, SAFs can in principle approximate any smooth function. A drawback, however, comparing to ALF, is that regularization techniques cannot be explicitly implemented.

4.3 Maxout Functions

One slightly different non-parametric activation function, is given by the introduction of a whole new layer called *maxout function*. With maxout, we compute K different dot products on each neuron and then take the maximum:

$$\text{maxout}(x) = \max\{W_i^T x + b_i\}_{i=1}^{K}. \tag{4.5}$$

A DNN wich employs maxout functions as non-linearities is called *Maxout Network* (Ref.). As for APL, maxout introduces several points of non-differentiability and usually requires more parameters than previous approaches, as we scale by a factor

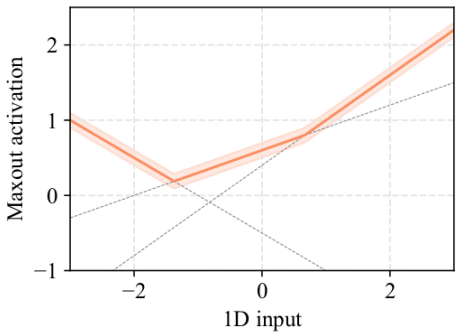


Figure 4.1. A maxout function with a one-dimensional input and K = 3. The three linear dot products are shown with light gray, while the max and thus resulting activation is shown in shaded red. Source: (Ref.)

of K the number of weights inside the net. Furthermore, maxout can only generate convex shapes and we lose the ability of plotting the activation function, except for input up to three-dimensional Fig. 4.1. Different variations aim to solve the smoothness problem for Maxout Networks (Ref.) (Ref.).

4.4 Kernel-Based Activation Functions

Introduced in (Ref. Kafnets), kernel-based activation functions (KAFs) is a class of non-parametric activation functions that leverage kernel expansions to adapt the shape on a per-neuron basis. Let D be a user-defined positive integer, then a KAF acting on activation $x \in \mathbb{R}$ has the following form:

$$\text{KAF}(x) = \sum_{i=1}^{D} \alpha_i \kappa\left(x, d_i\right), \tag{4.6}$$

where $\{\alpha_i\}_{i=1}^{D}$ are the usual parameters to adapt, called *mixing coefficients*, $\kappa \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a 1D kernel method (Ref.) and $\{d_i\}_{i=1}^{D}$ the kernel's dictionary elements. Dictionary elements are usually sampled from training data, but in case of activation functions this would tie the size of the expansion D to the specific dataset used.

Since generality is preferred, KAFs use fixed dictionary elements by selecting D equally spaced points on the x-axis centered at 0 with step size Δ, similary to SAFs. In particular, there is a vast literature on kernel methods with fixed dictionary elements (Ref. Snelson and Ghahramani, 2006). A Neural Network that makes use of KAFs is called *Kafnet*.

To allow for better optimization with Gradient Descent we want the kernel $\kappa(\cdot,\cdot)$ to be positive semi-definite and thus convex, i.e., for any choice of $\{\alpha_i\}_{i=1}^{D}$ and $\{d_i\}_{i=1}^{D}$ we have:

$$\sum_{i=1}^{D}\sum_{j=1}^{D}\alpha_i\alpha_j\kappa\left(d_i,d_j\right)\geq 0. \tag{4.7}$$

As for which kernel method one should use, to perform our experiments in the following chapters, we will stick to the original paper and use the 1-dimenstional *Gaussian kernel*:

$$\kappa\left(x,d_i\right)=\exp\left\{-\gamma\left(x-d_i\right)^2\right\}, \tag{4.8}$$

where the normalization term γ is called *kernel bandwidth* and is empirically chosen to be $\gamma=\frac{1}{6\cdot\Delta^2}$. The Gaussian kernel is easy to implement using vectorization libraries and quite cheap to compute provided that the dimensions don't grow too large. Moreover, concerning the backward pass, KAFs have pretty straightforward derivatives as well:

$$\frac{\partial g(x)}{\partial\alpha_i}=\kappa\left(x,d_i\right), \tag{4.9}$$

$$\frac{\partial g(x)}{\partial x}=\sum_{i=1}^{D}\alpha_i\frac{\partial\kappa\left(x,d_i\right)}{\partial x}. \tag{4.10}$$

Additionally, employing the Gaussian kernel has two more benefits: first, it allows for locality effects of parameters and second, the resulting (Gaussian) KAF is an universal approximator for continuous segments (Ref. Micchelli et al., 2006).

The mixing coefficients can either be initialized randomly from a normal distribution, giving no direction yet complete freedom in the trend that the modeled shape should follow Fig. 4.2, or, on the contrary, we can initialize mixing coefficients such that the KAF will start with an interpolation of any given function f Fig. 4.3. Let $\mathbf{t}=(t_1,t_2,\cdots,t_D)$ be the set of values assumed by f when acting on dictionary elements i.e. $\mathbf{t}=(f(d_1),f(d_2),\cdots,f(d_D))$. We can then initialize the mixing coefficients in the following way:

$$\alpha=\left(K+\epsilon I\right)^{-1}\mathbf{t} \tag{4.11}$$

where $K\in\mathbb{R}^{D,D}$ is the kernel matrix with $K_{i,j}=\kappa(d_i,d_j)$ entries, and we add a diagonal term $\epsilon>0$ to avoid degenerate solutions. To constrain and regularize the learned parameters, contrary to SAFs, KAFs also allow the use of common regularization techniques. It is also worth mention that we can improve the flexibility of KAFs by letting the network adapt, along with mixing coefficients, both the dictionary elements and the bandwidth. This further level of flexibility can indeed help reach better performances as will be shown in the Evaluation chapter.

In (Ref. KafNets) authors performed different experiments to assess the performances of the proposed KAFs. It turned out that KAFs managed to improve the

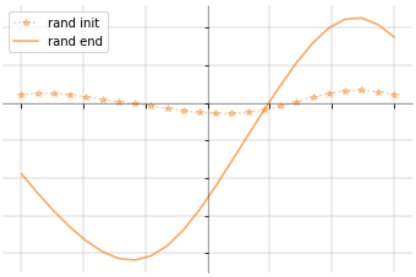


Figure 4.2. The shape of a KAF with randomly initialized coefficients is shown in light-starred orange. The final learned shape of the same KAF after training is shown with a continuous orange line. Notice the strong difference between the two functions and how the final one resembles a shifted tanh.

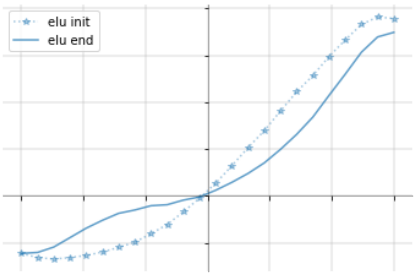


Figure 4.3. The shape of a KAF with coefficients initialized through ridge regression to approximate the ELU function is shown in light-starred blue. The final shape after training is shown with a continuous blue line. Notice how the difference between the two is less evident comparing to randomly initialized KAF (left). Both KAFs (left and right) were trained on the same neuron for the *MNIST* dataset using 5 epochs and same overall settings.

results obtained with fixed activation functions over several benchmarks ranging from simple datasets like the *Sensorless* dataset (Ref.) to larger ones such as *SUSY* (Ref. SUSY dataset), testing on shallow feedforward Neural Networks as well as deeper CNNs architectures and for both supervised and unsupervised tasks. A summary of the reported results is given in Table 4.1.

Scardapane et al. KAF Experiments					
Dataset	**Design**	**Metric**	**Fixed AF**	**Param. AF**	**Non-param. AF**
Sensorless	FNN*	Accuracy	tanh:99.18%	PReLU:99.30%	KAF:99.80%
SUSY	FNN**	AUC	ReLU:0.8739	PReLU: 0.8748	Maxout:0.8744 APL:0.8757 KAF:0.8758
CIFAR10	CNN***	Accuracy	ELU:78%	*n.a.*	ELUKAF:83%

Table 4.1. Each row summarize the results obtained using different activation functions on a specific task. Underlined entries stand for best result achieved. *: feedforward neural networks with 3 hidden layers of 100 neurons are used for fixed or parametric activation functions whilst a single hidden layer is used for KAF activation function. **: feedforward neural networks with 5 hidden layers and 300 neurons each for fixed or parametric activation functions whilst 2 hidden layers with the same number of neurons for non-parametric activation functions
***: CNNs made by stacking 5 convolutional blocks, each composed by (a) a convolutive layer with 150 filters, with a filter size of 5×5 and a stride of 1; (b) a max-pooling operation over 3×3 windows with stride of 2; (c) a dropout layer with probability of 0.25. See the original article for a full description of the architectures, hyperparams and training settings.

Part II

Robustness of Kafnets

Chapter 5

Related Works

Among t

5.1 K-Winners Take All

In (Ref. ENHANCING ADVERSARIAL DEFENSE BY k-W INNERS-TAKE-ALL), Xiao *et al.* advocate the use of a C^0-discontinuous activation function, called *k-Winners-Take-All* (k-WTA) activation, to improve, with no substantial overhead, the robustness of a neural network against gradient-based attacks such as PGD. k-WTAs work by acting on the whole layer $\mathbf{y}$, similarly to maxout, but in this case they filter the input retaining the k-largest values and deactivating to 0 the remaining ones. More formally, a k-WTA acting on an input vector of neurons $\mathbf{y} \in \mathbb{R}^n$ is a function $\phi_k(y) : \mathbb{R}^n \to \mathbb{R}^n$ such that:

$$\phi_k(\mathbf{y})_j = \begin{cases} y_j, & y_j \in \{k \text{ largest elements of } \boldsymbol{y}\} \\ 0, & \text{Otherwise.} \end{cases} \tag{5.1}$$

where $\phi_k(\mathbf{y})_j$ denotes the j-th element of the output. Notice how $\phi_k(\mathbf{y})$ is effectively parametrized by k but it cannot be regarded as a parametric activation function since k is user defined and not differentiable. Moreover, since it is likely to have layers with different shapes inside a net, we use the *ratio* λ of the number of neurons of each layer to compute the correct k, simply as $k = \lambda \cdot |layer|$.

K-WTAs foster robustness by making the computation of the gradient $\nabla_x f_\theta(x)$ *undefined*. Loosely speaking, this is achieved thanks to densely distributed discontinuities in the space of x. Indeed, this implies that, with very high probability, any perturbation moving x to it's neighbourhood will move over a non-differentiable spot Fig. 5.1. in the adversary objective, making gradient-based search unfeasable. The meticolous reader may at this point start to wonder how is thus even possible to perform training with such discontinuous functions. However, it turns out that the gradient with respect to the weights $\nabla_w f_\theta(x)$ is discontiuous but in a sparser way, presumably beacuse the parameter space where w leaves is much larger than the input space, allowing the training to succeed. To gain a better understanding of what is going on, the paper provides also a theoretical framework in which both the dense discontinuities and the trainability can be explained. Since k-WTAs are only

marginally used in the following parts of this thesis, we will not go in the details of such proofs.

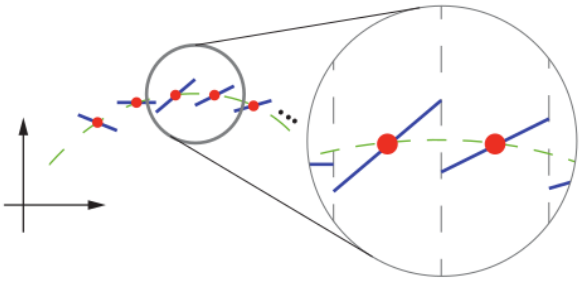


Figure 5.1. A plot of a 1-dimensional k-WTA interpolating a curve. As stressed by the figure, k-WTAs are piece-wise linar functions where points of non-differentiability are very much densely distributed. Any small change in the input results in an abrupt change in the function's value. Source: (Ref.)

Authors claim substantial improvements in the robustness of this defense against white-box attacks. Specifically, they were able to evaluate a ResNet-18 against PGD, C&W (Ref.) and DeepFool (Ref.) attacks, training both classically and with AT 3.9. Using CIFAR10 for the dataset, ResNet-18 with ReLUs achieved 0% and 43.6% accuracy under the most effective attack, respectively with standard and adversarial training. On the other hand, when k-WTAs were used, the same network improved to 13.1% with standard training and 50.7% with AT.

In a recent work (Ref.), Carlini *et al.* highlighted fundamentals flaws in the defense that was just claimed for k-WTAs. Authors were in fact capable of showing how this method falls within a broader category of attacks, known as gradient-masking defense methods (Ref. survey 2), which are already known in literature to be vulnerable. It is nevertheless true that, for unadaptive and straightforward implementations of current gradient-based attacks, k-WTAs still provide a fair protection, especially when the networks is adversarially trained. For this reason, in the following chapters we will make use of k-WTAs (for which we give a novel implementation in *TensorFlow2*), as a benchmark for our evaluations.

5.2 Smooth Adversarial Training

The current wisdom among researchers, suggests that there probably is a fundamental trade-off between accuracy and robustness that we must deal with (Ref. There is no free lunch in adversarial robustness). That is, improving the resiliency of a model against adversarial examples will almost certainly result in a less accurate model. In particular, whenever an effective technique to improve the robustness is found, it almost always coincides with the method harming the model performances (Ref. On the convergence and robustness of adversarial training) (Ref. Max-margin adversarial training) (Ref. Theoretically principled trade-off between robustness and accuracy).

To prevent from performance decay, one feasable (but expensive) strategy is to increase the size of the network by making it either deeper and/or wider (Ref.

Intriguing Properties of NNs at scale.). It seems nevertheless reasonable to believe that there could be other ways that would allows us to build more robust networks without sacrificing in neither accuracy nor resources. That is exaclty what was recently reported by Xie *et al.* in (Ref. Smooth Adversarial Training), where authors found a way to consistently improve robustness over different tasks, without giving up any other property of the model. Specifically, they found that, it is sufficient to prefer and adopt smooth activation functions over non-smooth ones, like the everywhere present ReLUs, to achieve better results in AT straightaway.

The rationale behind adopting smooth activation functions, i.e., non-linear functions that are continuous in their first derivative, lies in the observation that, during AL, we are computing many times the gradients of the network that are normally required during standard training. Specifically, in addition to compute gradients to update the network's parameters, adversarial training also needs gradients computation for generating training adversarial samples. Moreover, the presence of non differentiable points and abrupt changes in the gradient's values during backpropagation, may prevent from generating sufficiently strong adversarial samples. This particularly holds true in the case of ReLUs, where the gradient lacks of flexibility and gets a severe jump in the origin Fig. 5.2.

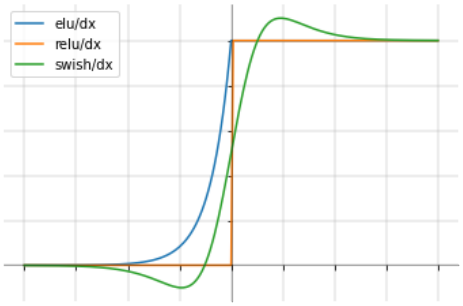


Figure 5.2. Plot of the first derivative for ReLU, ELU and Swish activations. Except for the rectified linear unit, both the ELU and Swish have continuous derivative.

To test their hypothesis, Xie et al. evaluated a ResNet50 on ImageNet for different AT runs, by switching activation function after each training. For their experiment, parametrized *Softplus* (Ref.), Swish, *GELU* (Ref.) and ELU were used as smooth activations, whereas ReLU for the baseline. Notice that, for the case of ELU, the parameter α 2.18 should be set equal to 1 in order to avoid the gradient being undefined at the origin. Compared to the baseline, all smooth activation functions substantially boosted robustness while keeping the accuracy nearly unchanged. In particular, any smooth activation function at least improved robustness by a factor of 5.7%. Best results were achieved with Swish, which enabled ResNet50 to achieve 42.3% robustness and 69.7% standard accuracy as shown in Fig. 5.3

To test further the issue with ReLU and generalize the smoothness conjecture, another experiment was made targetting ELUs. As we said, when we set the parameter α to be different to 1, the function becomes non-differentiable in the origin, with the gradient abruption increasing for larger values of α. With no surprise, if we perform the same evaluation with no change in the settings except for equipping the network with non-smooth ELUs, we observe that the adversarial robustness is

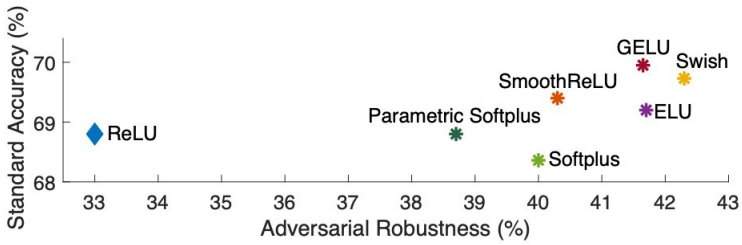


Figure 5.3. Compared to ReLU, all smooth activation functions showed a significant increase in robustness while some even improved accuracy. In the original work, 2 others smooth activation functions were used, namely SmoothReLU and Softplus. We did not include them since they are not commonly used in practice. Source: (Ref. Adversarlial Smoothness)

highly dependent on the value of α. The highest robustness is achieved when the function is smooth while for all other choices of α the robustness monotonically decreases when we gradually approache $\alpha = 2.0$. In particular, with α being 2.0, robustness drops to 33.2%, that is, 7.9% lower than that of smooth ELU. The observed results are consistent with previous conclusions on ReLU: non-smooth activation functions significantly weaken adversarial training.

From a broader perspective, the work on smooth activation functions might hints towards the draft of a more general design principle, which is that, architectural smoothness might play an essential role in enhancing adversarial rbustness, at least for what concerns defenses against gradient-based attacks.

With a combination of lessons just learned from both Xiao et al. and Xie et al., the focus of the following chapters will then be on the evaluation of a novel kind of activation functions. Functions that we believe might have , by laveraging, among other properties, strong smoothness and flexibility, the potential of surpassing both k-WTAs and Swish for what concerns robustness and, in particular, the trade-off between accuracy and robustness. Distinguished activation functions that satifsy such prerequisites are the previously introduced non-parametric kernel-based activation functions (KAFs).

Chapter 6

Solution Approach

To test novel architecture components or training techniques with real data, we first need to implement them as software. In our case, we use *Python* (Ref.) programming language and, in particular, experiments are conducted using *TensorFlow2* (Ref.) (TF2), a modern framework for differentiable-programming. In turn, TF2 integrates another famous library, *Keras* (Ref.), which provides high-level API to build neural networks based models. Specifically, Keras allows to describe customized layers hence it can be used to implement k-WTA and KAF activation functions. A working implementation is given in *activationsf.py*.

6.1 KAFs May Be Good Candidates

Once the code is done, we can begin and perform some early evaluations. As a first thing, we reasonably chose to test the robustness of simple neural networks and in particular, we can already try to understand if there is any meaningful difference when we change activation functions.

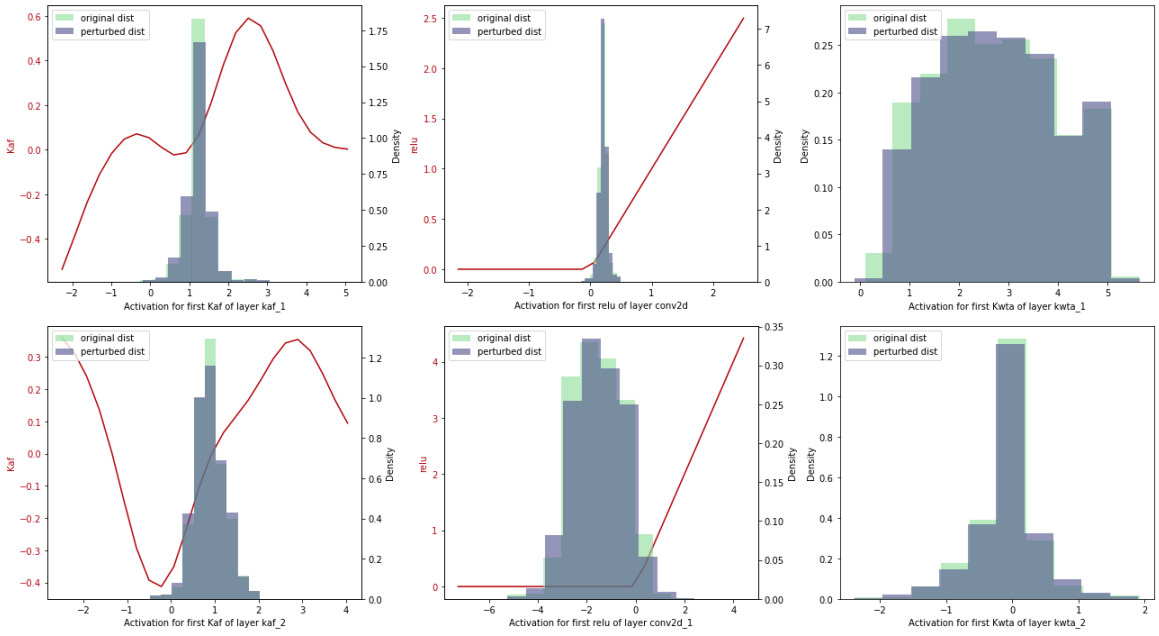
The light first-developed model is made of few parameters, to conduct quick experiments and perhaps find an initial direction to follow. The network is composed by two convolutional blocks, each of which holding two convolutional layers followed by batch normalization. After those two convolutions we halve the dimensions with a maxPooling and then apply regularization with Dropout. In the first block we use 32 filters, in the second 64, each with window size 3. Finally, a global average pooling is used followed by a fully-connected layers with 128 units which is in charge of computing the output layer. The dataset used, and the only one that we will be using for each evaluation, is CIFAR10. The code where such networks are built and trained can be found in *light_models.ipynb*.

Respectively, using ReLU, k-WTA and KAFs (with randomly initialized mixing coeffiecients and D=20) and using Adam optimizer, we obtained 77.96%, 65.33% and 68.80% standard accuracy. After that, we are ready to see how such trained nets behave against adversarial examples. Following the settings of (Ref. kWTA), a PGD attack is crafted under the L_∞ metric, with perturbation size $\epsilon = 0.031$, steps size 0.003 and 40 random restarts. Importantly, we only attack 1000 random samples taken from the test set, since, due to resources limitations we were not capable of computing the attack on the whole network in reasonable time and, moreover,

because we only wanted to a rough estimate to begin with. To actually implement such attacks we will use the *Adversarial Robustness Toolbox* (ART) from IBM (Ref. ? should i cite this?).

The first results are not really in line with what was expected. In particular, in (Ref. KWTA) a ResNet18 equipped with ReLUs results in 0% accuracy against the aforementioned attacks, whereas in our case the net manages, even if with small numbers, to correclty classify 11% of the 1000 samples attacked. This means that, even if we assume that every of the remaining samples were to be classified wrongly, we would have still classified in the right way the 1.1% of the test set, in contraddiction with the 0% gained in the original article. However, it plausible that this depends by the model being used, indeed, it is likely that ResNet18 equipped with ReLUs is more prone get fooled by the attack. Despite this inconsitency, k-WTAs on the other hand behave similarly, getting 13,2% of the samples correclty classified. Another surprise concerns KAFs as well: we managed to reach 15.7% of accuracy which already a big improvement if we consider k-WTAs were specifically designed to resist gradient based attacks and still, a straightforward implementation of the KAFs allowed us to overcome them.

Moved by this results, we decided to go more in depth with the analysis and look at what is actually happening inside these different implementations of our net when we give adversarial inputs. In particular, we wanted to examine, for each layer, the distributions of the activation values, i.e., the inputs on which activation functions act, to see if we could spot any evidence of the observed results. Fig. 6.1 shows plots for the distributions we obtained when we consider the first neuron of the layer, namely, the fist neuron of the first filter in the case of convolutional layers or the first neuron in the fully-connected layer before classification. Distributions are computed with respect to both perturbed (violet) and non perturbed (light-green) samples.



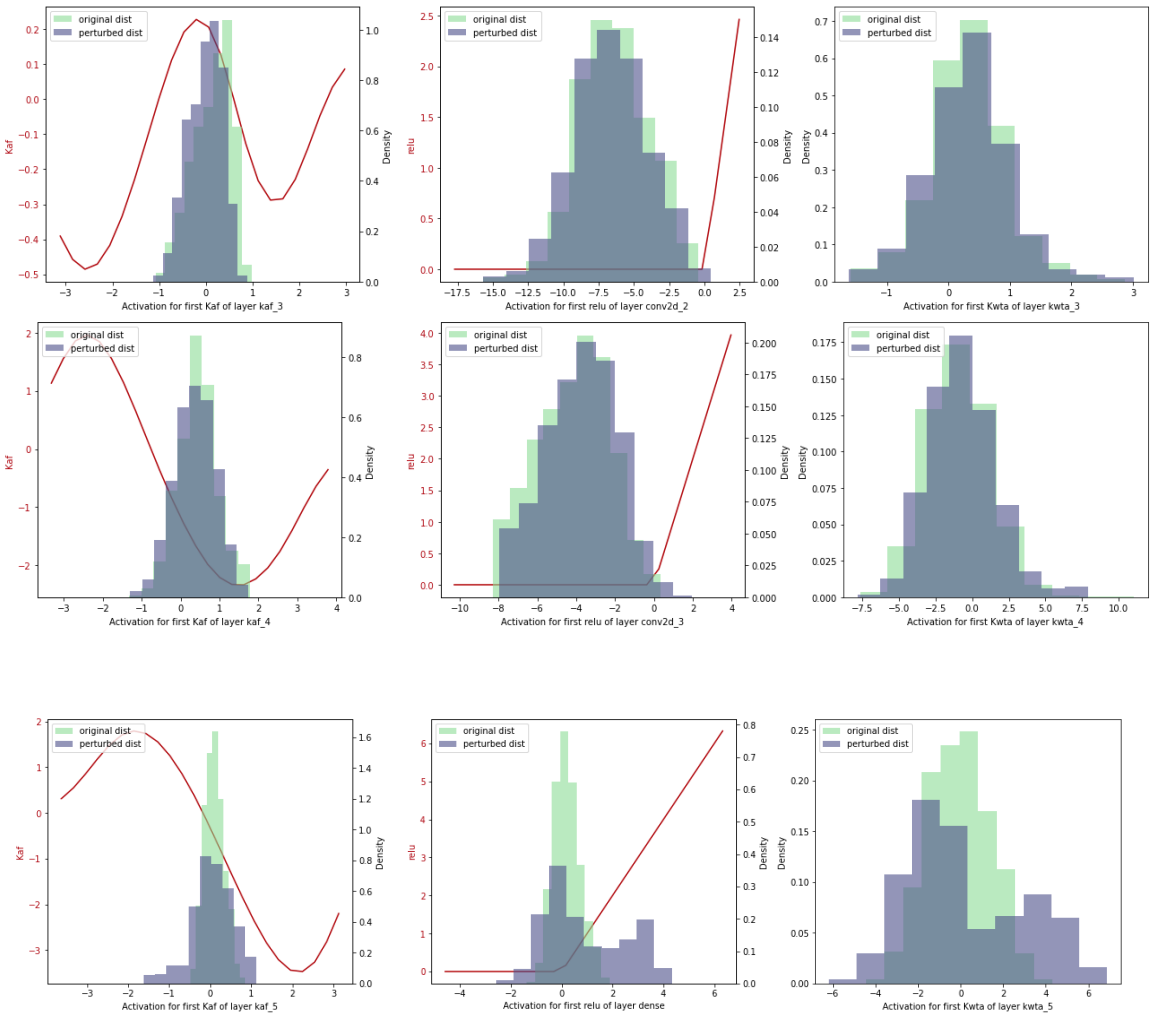
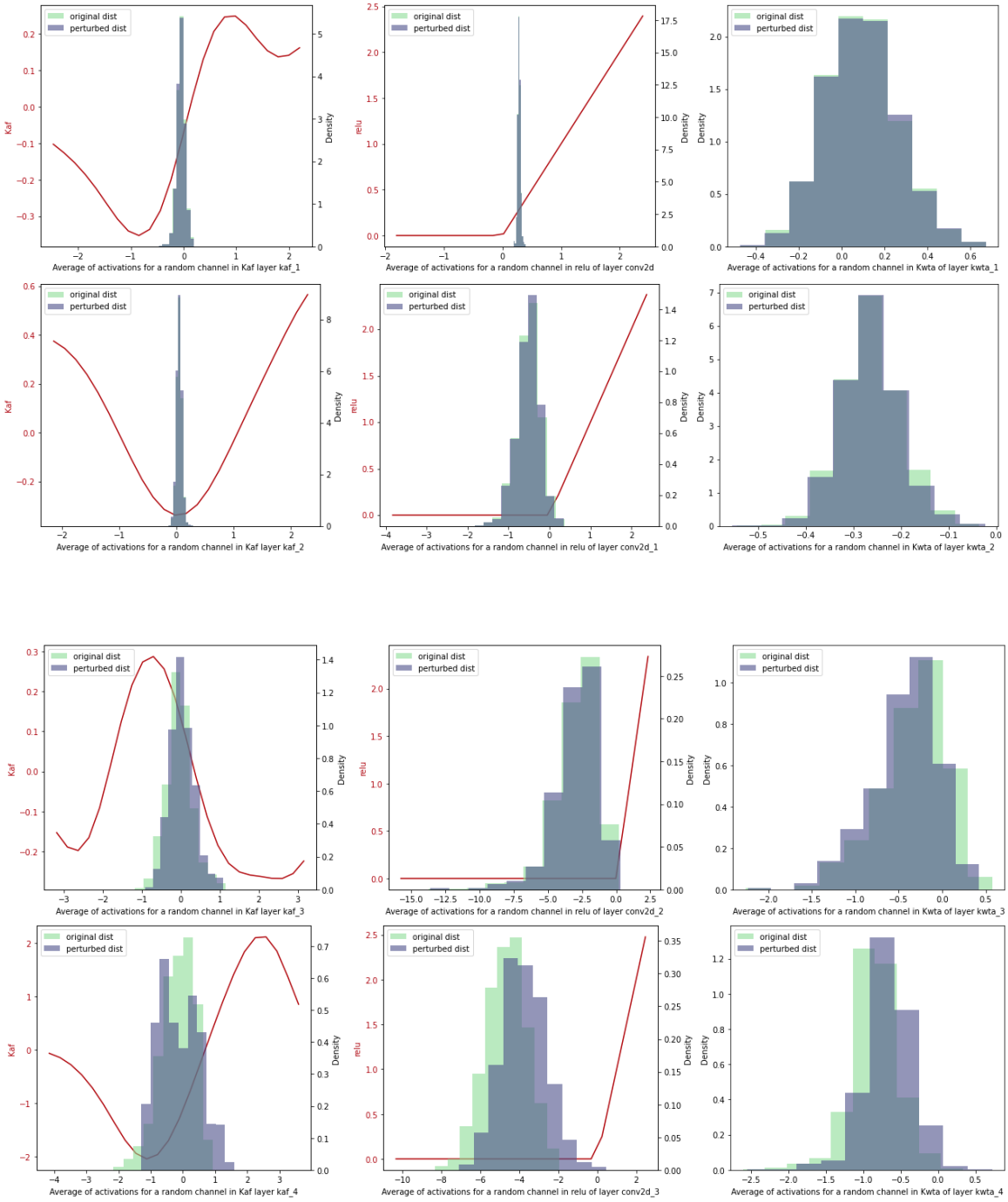


Figure 6.1. Each histogram shows the distribution of the activation values for the first neuron of the first channel, for each layer. The distribution colored in violet represents the activations for the batch of perturbed samples while the light-green stands for the distribution of values the neuron gets when we feed the network with original samples. We also plot, when possible, the actual shape of the activation function acting on the neuron with a red line. The effects of the perturbation can be appreciated especially in the last layer where distributions differ the most.

The rationale behind these plots is to observe, the more we go deep in the network, how the two distributions, very similar in the initial stages, tend to diverge into two distant distributions. This behavior is precisely what we would excpect since the networks are making, for the majority of the 1000 samples, different predictions. However, despite the fact that our expectations are confirmed by these histograms, a more careful analysis would also find that the distance in the last KAF layer seem to be smaller in comparison to the distance (between distributions) that we see in the last layer for k-WTA and ReLU. Having said that, it is still plausible that what we concluded might hold true just for the first neuron of the layer while being false in general for the others. For this reason, the next step is to repeat the experiment,

this time considering the distribution over more neurons, that is, we consider the average mean for the activations of a given filter, in the case of convolutional layers, or taking the mean of the whole last layer's activations Fig. 6.2.



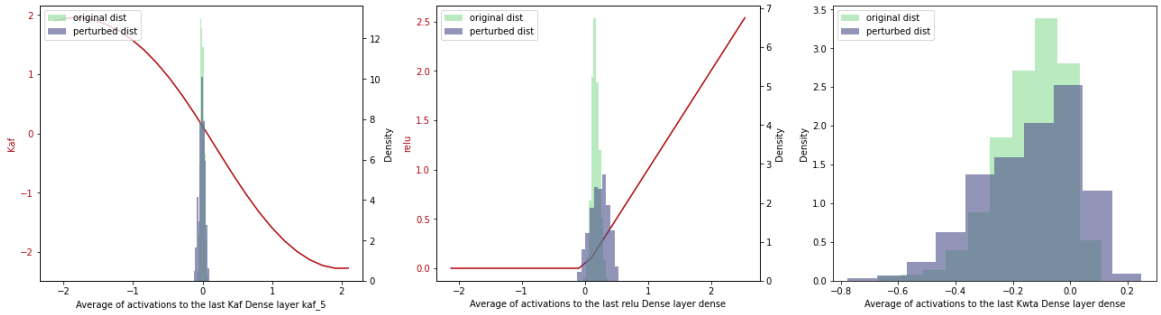


Figure 6.2. Each histogram shows the distribution of the mean value of the activations for a random layer's filter, in the case of a convolutive layer, or for every neurons in the layer, in the case of dense (last) layer. The effects of the perturbation are less visible when the net is equipped with KAF activation functions.

With more confidence, we can now observe that, what was previously happening in the case of a single neuron is actually happening in general: on average, the Kafnet seem to be less prone to big changes within its layers and its output, than when we consider K-WTA, or, in particular, ReLU activation functions. Specifically, if we want to quantify this this distance, given x the original input and x' its attacked version, we can compute the L_2 norm of the difference between perturbed and non-perturbed activations, for each layer. By doing so, and repeating it on many of the attacked inputs, for instance, when considering the first sample and its adversarial pair we get:

```
Perturbation Effects at layer 1: relu 0.00011200506560271606; KAF 0.0006243169773370028; KWTA 0.006895178463310003
Perturbation Effects at layer 2: relu 0.08248745650053024; KAF 0.026331476867198944; KWTA 0.011613165028393269
Perturbation Effects at layer 3: relu 3.3173251152038574; KAF 0.2809731364250183; KWTA 0.037524059414863586
Perturbation Effects at layer 4: relu 1.3507401943206787; KAF 26.69105339050293; KWTA 43.38331985473633
Perturbation Effects at layer 5: relu 175.467041015625; KAF 159.125; KWTA 175.42672729492188
```

, when considering the second pair:

```
Perturbation Effects at layer 1: relu 6.316676444839686e-05; KAF 0.0012307873694226146; KWTA 0.004430566914379597
Perturbation Effects at layer 2: relu 0.13316965103149414; KAF 0.01723913475871086; KWTA 0.014558049850165844
Perturbation Effects at layer 3: relu 8.279311180114746; KAF 0.3452574610710144; KWTA 0.07315781712532043
Perturbation Effects at layer 4: relu 2.254383087158203; KAF 36.55812072753906; KWTA 41.92384338378906
Perturbation Effects at layer 5: relu 207.2716522216797; KAF 157.625; KWTA 178.77764892578125
```

, the 50th pair:

```
Perturbation Effects at layer 1: relu 0.00012299319496378303; KAF 0.002121036173775792; KWTA 0.008412819355726242
Perturbation Effects at layer 2: relu 0.23596253991127014; KAF 0.013095371425151825; KWTA 0.02081015706062317
Perturbation Effects at layer 3: relu 13.298871994018555; KAF 0.2131684124469757; KWTA 0.050939373672008514
Perturbation Effects at layer 4: relu 3.958709716796875; KAF 32.653343200683594; KWTA 65.42059326171875
Perturbation Effects at layer 5: relu 284.0301513671875; KAF 159.125; KWTA 449.6789855957031
```

and so on. The Kafnet almost always results in smaller distance norms. The code for reproducing these experiments can be found in *insights_link_rob_activationsf.ipynb*.

Thanks to an in-depth analysis on the architectural statistics we were able, at least from this point of view, to justify why a Kafnet is winning against the other two implementations. Moreover, this results seem to suggest that KAFs, especially when considered together with their flexibility properties, might be good candidates for enhancing robustness through the lens of Robust Optimization. In particular, they seem to fit well when it comes to regularization techniques bounding the effects of perturbations, such as the previously introduced Lipschitz

constant regulatization 3.2.3. Even though this regularization technique might appear as the natural approach to follow, it exhibits several difficulties to overcome, such as the lack of efficient and easy to use implementations in current Deep Learning frameworks, and, more importantly, it is in general difficult to come up with an explicit bound on the Lipschitz constant for (Gaussian) KAF layers (Ref. https://arxiv.org/abs/1903.11990). Therefore, we opted to approach Robust Optimization with the well studied method of adversarial training 3.9 and perform the training laveraging also on the smoothness properties of Gaussian KAFs and the previously introduced results linking smoothness and AT together 5.2.

6.2 Fast is Better than Free Adversarial Training

Adversarial training (AT), can be an expensive procedure to accomplish, and it is not due to the lack of resources but rather to the way we implemented it. For example, a textbook implementation of Madry's PGD adversarial training (Ref.), as for 2019, required 4 days to run over CIFAR10 using a Titan X GPU, and, in order to scale to ImageNet, researchers had to deploy large clusters of GPUs at a scale far beyond the reach of many institutions (Ref. Adversarial Training for Free!). Consequently, many research has been devoted to improve this computational barrier. For instance, when performing multi-step PGD adversary in the inner maximization, it is possible to cut out redundant calculations during back propagation to gain additional speed-ups (Ref. Zhang et al. 2019). Or, again, insead of updating the network's weights after multiple forward and backwards passes needed to compute the PGD, we could update the weights together with the perturbation δ at each backward pass of the PGD, hence reducing the overall number of iterations needed to complete the training (Ref. Adversarial Training for Free!). The latter method is know in literature as *Free Adversarial Training* to refer to the apparent relief from the complexity of standard adaversarial training. Nevertheless, even employing these techniques can still result in prohibitive running times.

A recent breakthrough was claimed by Wong *et al.* in their article *Fast is Better than Free: Revisiting Adversarial Training*(Ref.), where authors showed they were able to train models that matched state of the art robustness against full-strenght PGD attacks, for both CIFAR10 and ImageNet, in 6 minutes and 12 hours respectively. The proposed method, which allowed them to achieve such results, consists in a mixture of: a clever way for computing the inner maximization in one step, and novel techniques used in efficient training of deep networks, such as *cyclical learning rates* (Ref. Smith et al.) and mixed-precision arithmetics. The letter being two frequently used key-components in top submissions for image classification competitions like DAWNBench (Ref.).

Regarding the method used to approximate the inner maximization, it was used a small variation of the FSGM 3.1.1 adversarial training. The original FSGM AT works by computing a single step FSGM attack on the mini-batch with step size ϵ to generate the adversarial sample.

procedure FGSM_AT_ORIGINAL($epochs = T$,$dataset\_size = M$,ϵ)
  for $t \leftarrow 1, T$ **do** ▷ For each epoch
    for $i \leftarrow 1, M$ **do** ▷ For each minibatch
      $\delta \leftarrow 0$
      $\delta \leftarrow \delta + \epsilon \cdot sign(\nabla_\delta \mathrm{LS}\left(f_\theta(x+\delta), y\right))$ ▷ Perform FGSM
      $\delta \leftarrow \mathcal{P}(\delta)$
      $\theta = \theta - \nabla_\theta \ell\left(f_\theta\left(x_i + \delta\right), y_i\right)$ ▷ Update model weights
    end for
  end for
end procedure

FSGM AT is generally known to be a weak defense against PGD attacks (Ref. Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world). Wong *et al.* extended this approach by adding a random initialization of the perturbation δ (within the allowed space Δ) and by increasing the step size α by a factor of 1.25, making it $\alpha = 1.25\epsilon$. The overall algorithm to adversarially train the model then becomes:

procedure FGSM_AT_WONG($epochs = T$,$dataset\_size = M$,ϵ)
  for $t \leftarrow 1, T$ **do** ▷ For each epoch
    for $i \leftarrow 1, M$ **do** ▷ For each minibatch
      $\delta \leftarrow \mathcal{U}(-\epsilon, \epsilon)$ ▷ Random init
      $\delta \leftarrow \delta + 1.25\epsilon \cdot sign(\nabla_\delta \mathrm{LS}\left(f_\theta(x+\delta), y\right))$ ▷ Perform extended FGSM
      $\delta \leftarrow \mathcal{P}(\delta)$
      $\theta = \theta - \nabla_\theta \ell\left(f_\theta\left(x_i + \delta\right), y_i\right)$ ▷ Update model weights
    end for
  end for
end procedure

Notice how, for a standard effective AT, that computes adversarial samples via PGD using, say, m steps, we need to compute $\mathcal{O}\left(mM\right)$ operations per epoch, whereas with $FSGM$ we shrink down to $\mathcal{O}(M)$ operations, i.e., asymptotically equal to number of operations required for standard training. Therefore, with only these two small variations to the original FGSM AT, as well as the employment of cyclical learning rates and mixed-precision arithmetic to accelerate convergence, it is possible to devise an adversarial training method, with no significant computational overhead to standard training, which achieves leading edge robustness-accuracy trade-offs.

Moved by the early results obtained on vanilla (i.e. classically trained) Kafnets, and by the feasability of performing strong adversarial trainings, we argue that an evaluation of the robustness of adversarially trained Kafnets is a direction worth taking. More importantly, as previously discussed, the smoothness of network's components, such as activation functions, can additionally boost the resiliency against adversarial examples, and that is precisely the case for Gaussian KAFs. Indeed, recall that the first derivative of a KAF with Gaussian 1D kernel 4.8 is given by:

$$\frac{\partial g(x)}{\partial x} = \sum_{i=1}^{D} \alpha_i \frac{\partial \kappa\left(x, d_i\right)}{\partial x}, \tag{6.1}$$

which is a sum of a finite number of continuous functions $\alpha_i \frac{\partial \kappa(x,d_i)}{\partial x}$, which makes the whole derivative continuous. Therefore, together with their adaptable nature, we conjecture KAFs will improve, with respect to traditional activation functions, both the inner product of adversarial training:

$$\delta = \delta + \alpha \cdot sign(\nabla_\delta \operatorname{LS}(f_\theta(x+\delta), y)), \tag{6.2}$$

in terms of crafting stronger attacks thanks to smoothness, as well as the outer minimization:

$$\theta = \theta + \beta \cdot \nabla_\theta \operatorname{LS}(f_\theta(x+\delta), y), \tag{6.3}$$

where the 'learning' of the attack will benefit from having trainable activation functions, as it happens with standard training 4.1.

This theoretically principled hypothesis on the potential advantages that might come from adversarially train Kafnets needs however an empirical validation. For this reason, the rest of this thesis will be directed into presenting the findings resulted from different experiments we made evaluating the robustness of Kafnets.

Chapter 7

Evaluation

In this chapter, experiments assessing the robustness of Kafnets are discussed. In a comparison with their traditional counterparts, we try to understand what is the preferable way to leverage KAFs in this context. If it is by using random initialization of the coefficients, or if we should use them to approximate a specific function and use ridge regression initialization. Or again, what happens if we push flexibility even further and we let the network train dictionary elements and kernel bandwidths too? Do we gain any improvement? Is there a more advisable architecture for robust Kafnets and how well do they perform when we scale to deeper networks? The follwing sections will cover each of these points. After several evaluations, we try to interpret the obtained results and highlight why they are or not consistent with our initial hypothesis, what could be the shortcomings of our work and, in particular, what are the implications.

To perform each test we make use of the same software environment presented before in 6, with the additon of *TensorFlow Addons* (Ref.), a library that contains niche functionalities such as the cyclical learning rate technique reccommended by Wong *et al.* in (Ref. FBF) to speed-up adversarial training. With respect to adversarial training, to the best of our knowledge, there is no implementation available for the extended fast FGSM AT compatible with TF2 at the time of writing. For this reason, we give a working solution in *fbfadvtrain.py* which can reproduce the results of the original paper. We choose *Google Colab* (Ref.) for the platform since it provides a built-in installation of the environment plus free access to fast GPUs such as NVIDIA Tesla T4, Tesla P100s that can significantly boost the running time. With these settings, each adversarial train or attack, the two most expensive operations, requires at most 4 hours to complete.

We propose two main experiments, where the only discriminating factor between the two is the model architecture. For the first one we are going to employ a VGG-inspired 2.5 convolutional neural network, whereas for the second one a ResNet20. In both cases, the task is to devise robust networks for CIFAR10, where robust optimization is approached using adversarial learning and specifically we will be using the extended FSGM adversarial training discussed above. Following, the resulting trained networks are attacked by means of a multi-step PGD attack whose parameters are taken from (Ref. FBF), i.e., with 50 iterations, max perturation $\epsilon = 8/255$, step size $\alpha = 2/255$, 10 random restarts and perturbation metric $\|\cdot\|_\infty$.

For the optimization algorithm we use *Stochastic Gradient Descent* (SGD) (Ref.), as it is the standard choice for adversarial training (Ref. Towards Evaluating). The learning rate is scheduled following the one-cycle triangular learning rate policy where we let the learning rate linearly increase starting from a minimum value in the first epoch to a maximum value in the middle of the training and then decrease back with the same ratio until it reaches again the minimum value in the last epoch. Additionally, the net can be trained for few more epochs were we decrese the learning rate even more from the minimum value by another order of magnitude Fig. 7.1. The minimum and maximum values are picked through a process called

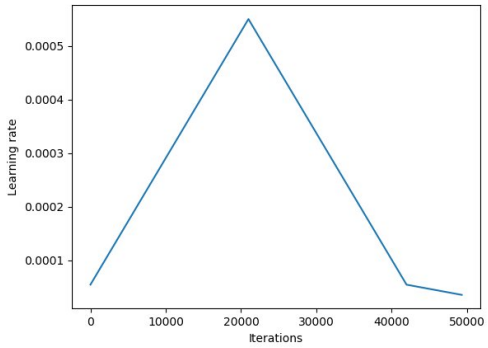


Figure 7.1. Learning rate evolution during a one-cyclic triangular learning rate schedule. Here the minimum value is 0.00005 and maximum value is 0.0005.

learning rate range finder(LRF) in which the network gets trained (in our case, adversarially trained) for few epochs and in the mean time the learning rate is exponentially increased starting from a very small value up to a very large one. The overall evolution of the loss during this learning rate escalation is then inspected: the minimum value is heuristically chosen to be the one from which the loss starts decreasing for the first time whereas the maximum value is picked when the loss reaches a plateau Fig. 7.2 Different LRF versions are available online, in our case

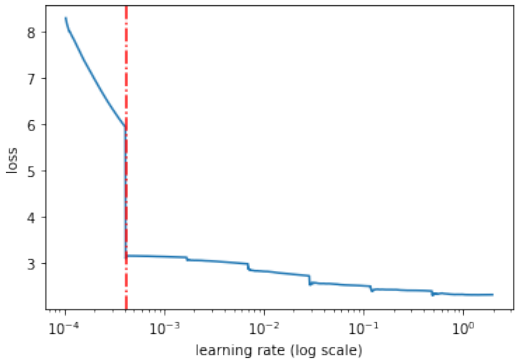


Figure 7.2. Loss evolution plot with respect to exponentially increasing learning rates. The vertical dot-dashed red line denote the learning rate value where the loss has the steepest decrease. While the minimum value matches with the smallest learning rate, the maximum one should be picked around 10^{-1}.

we use one from a public *GitHub* repository[1] since it offers integration with TF2.

Despite the fact that we previously used k-WTAs to show the potential of KAFs, it is worth to mention that such comparison was only meaningful for the purpose of our toy eperiments, but, in general, k-WTAs should not be used as a benchmark for assessing true robustness. Indeed, as shown in (Ref. Carilini https://arxiv.org/abs/2002.08347), only adding k-WTAs to the network is not an effective defense method at all, since they "make adversarial training worse". It turned out that it is actually possible to devise an adaptive, gradient-based attack capable of finding many adversarial examples, that brought down the accuracy of the original paper's k-WTA ResNet from 50% to 16%. Notice how this is not the case for other, standard, activation functions: there is currently no gradient-based attack which is known to break a proper adversarially trained network that uses, for instance, ReLUs activation functions. Therefore, K-WTAs should be considered unsecure. In the following sections, we will only be concerned in assessing KAF robustness against well-enstablished counterparts like ReLU, that will act as our baseline, or, as previously argued, the preferable ELU and Swish smooth activation functions.

7.1 VGG Inspired Architectures Results

To begin, we consider a CNN made from convolutional blocks, loosely inspired by the well-known VGG architecture (Ref. VGG). Each block is composed by two convolutional layers with the same number of filters, kernel size 3, interleaved by batch normalization applied before the non-linearities. After convolutions, we apply a 2×2 max pooling to reduce dimensionality and dropout for regularization, where the probability for a neuron to get dropped increases by 0.05 factor in each application starting from 0.2 in the first one to 0.4 in the last dropout layer. We use 3 convolutional blocks with 32 filters in the first one, 64 in the second one and 128 in the third. The second stage of the network begins with a 1-dimensional flattening layer applied after the last convolution. Then we fully connect the flattened vector to a 128 neurons dense layer, followed by dropout again, and finally the decision layer is another fully connected layer where the number of neurons is equal to the number of classes (10 for CIFAR10) and logits are generated applying a *softmax* application function Fig 7.3. Additionally, we employ *He-uniform* initialization on kernel weights and l_2 regularization on each dense layer with λ penality equal to 0.0012.

Once the basic structure is built, we only need to plug the specific activation function and test the robustness. We start with ReLU, which we consider our baseline. First of all, LRF is run and we find that the min and max learning rate bounds are, respectively, 0.0001 and 0.08, and then, we adversarially train the model for 60 epochs. At the end of the training the model with ReLUs achieves 78.1% of standard accuracy on the test set. Lastly, PGD is performed and the overall accuracy of our network against the perturbed images of the test set is 45.41%. Therefore, we were able to develop a relatively resilient networks against full strenght PGD attacks. To be precise, at this point we ran different version of our PGD by varying

[1] https://github.com/beringresearch/lrfinder

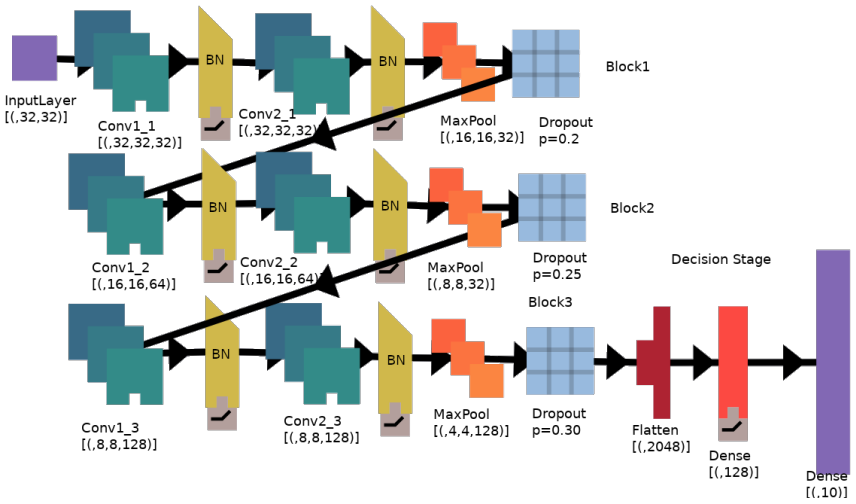


Figure 7.3. Visualization of the general structure of a VGG inspired CNN. During the experiments we will modify the activation function.

the *max_iter* parameter. It turns out that, for ReLU, just like for the following experiments, the attack converges around the 50th iteration. It is in fact not possible to harm significantly more even if we push the number of iterations up to 150 Fig 7.4. For this reason we are going to consider only PGD attacks with 50 steps when

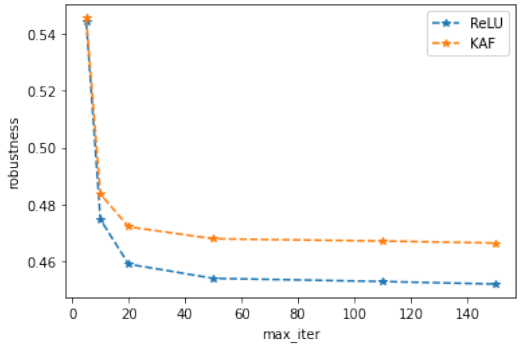


Figure 7.4. Increasing the number of attack iterations inside PGD stops being effective after a certain threshold. With our set-up, 50 iterations are enough to craft an optimal attack. Moreover, notice how the Kafnet improves robustness over the baseline.

evaluating.

We repeat the process equipping the same network with ELU non-linearities. This small change results in a minor improvement over the baseline in terms of robustness, with 45.49% of the perturbed examples correclty classified. However, the accuracy on clean samples drops to 73.99%, from an original 78.1% when using ReLUs. This latter result is apparently in contrast with the claims in (Ref. SAT), indeed, according the paper, we should have observed an increase in both accuracy and robustness. Nevertheless, this drop could also be explained with a wrong choice of learning rates bounds during LRF. To better understand why this is happening, we test other activation functions, this time using KAFs. In particular, KAFs with

randomly initialized mixing coefficients and dictionary size $D = 20$ are able to basically match ReLU accuracy, with 77.95%, and at the same time, to significantly improve robustness achieving 46.80% against our PGD.

It is reasonable to think that the promising gains that we experiment with KAFs might help us improving or even leveraging the properties of fixed activation functions to achieve better scores. Specifically, we build another network where activation functions are KAFs with same dictionary size as before but, in this case, mixing coefficients are ridge initialized to approximate the ELU function. We call such function KAF_ELU activation function. Evaluating the network with KAF_ELUs returns 74.93% accuracy and 45.4% robustness, i.e., even if slightly better than fixed ELU, these results are still worse than the baseline. Therefore, it is then reasonable to believe that the poor results with ELU and KAF_ELU are caused by the architecture itself, which is somehow less suitable to ELU like shapes.

We are then left to test the activation function which gained best results in (Ref. SAT): the Swish function. Accordingly to our expectation, with Swish the network boosts in robustness achieving 47.73%, hence more than 2 points better than with ReLUs, and also improving over KAFs results, making Swish the strongest activation function for this network against PGD, so far. Although, from the standard accuracy perspective, we are still performing slightly worse than ReLU and KAFs, obtaining 77%.

Similarly to the ELU case, also for Swish we developed a KAF approximation, that we denote as KAF_Swish. However, if with KAF_ELU we register close results with respect to the fixed counterpart, with KAF_Swish activation functions we get greater variations in the two metrics. Indeed, the KAF_Swish-equipped, adversarially trained network reaches top accuracy with a 78.85% but worsen the robustness achieving 46.33%.

Ultimately, we test another KAF version where kernel badwidth and dictionary elements are trainable parameters of the network, together with mixing coefficients. In this way, by doubling the number of parameters required for KAFs, we give in principle as freedom as possible in the modelling of an optimal activation function shape. Nevertheless, even with this last attempt, the model scores 77.54% and 46.2% in accuracy and robustness respectively, positioning these last "moving" KAFs (KAF_Free), in terms of perfomances, under the previously used KAF_Swish. In particular, KAF_Swish, KAFs and Swish activation functions resulted in best accuracy/robustness trade-offs, with Swish achieving strongest robustness and KAF_Swish strongest accuracy Fig. 7.5, Tab. 7.1. Code to reproduce the experiment and in-depth visualizations on KAF's shapes evolutions are located in *fbf_training_kafvgg.ipynb*

To what extent these results are accurate? Are we sure that KAFs cannot improve even more robustness? Is the Swish function hence the go-to transformation when it comes to robustness? These are all legitimate questions, in fact, somehow our initial hypothesis on the strong properties of KAFs seems to be only partially true. From our experiments it emerged that Swish function works better than KAFs if our only concern is robustness. However, a couple of things shoud be highlighted here. First of all, in many cases with KAFs and its variations (KAF_ELU, ..) we noticed, at the end of adversarial training, that there was still room for learning Fig. 7.6 and thus, 60 epochs might as well acted as a limitation in our evaluation

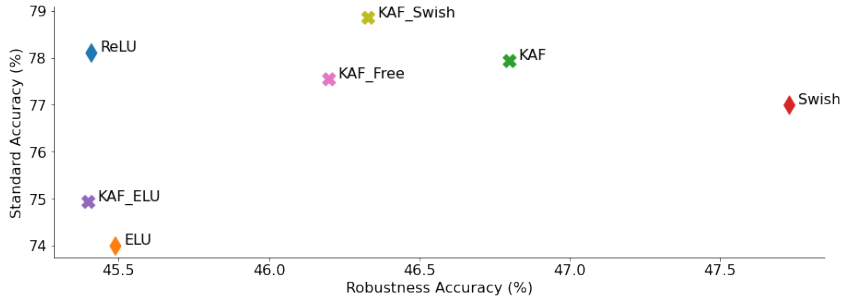


Figure 7.5. Robustness/Accuracy visualization for a VGG-inspired architecture equipped with different activation functions. Fixed activation functions are shown with a thin diamond whereas non-parametric KAFs with a filled x. Swish achieves best robustness and KAFs generally tend to improve both robustness and accuracy with respect to a ReLU baseline.

Activation	(min_lr, max_lr)	Accuracy	Robustness
ReLU	$(0.0001, 0.08)$	78.1%	45.41%
ELU	$(0.0001, 0.1)$	73.99%	45.49%
KAF	$(0.01, 1)$	<u>77.93%</u>	<u>46.8%</u>
KAF_ELU	$(0.0001, 0.1)$	74.93%	45.4%
Swish	$(0.0001, 0.3)$	76.99%	**47.73%**
KAF_Swish	$(0.0001, 0.8)$	**78.85%**	46.33%
KAF_Free	$(0.01, 2)$	77.54%	46.2%

Table 7.1. Summary of the results gained from changing activation functions in an adversarially trained VGG network. Bold denotes best result while underline best trade-off.

process. Furthermore, in our designed architecture, we made an extensive use of

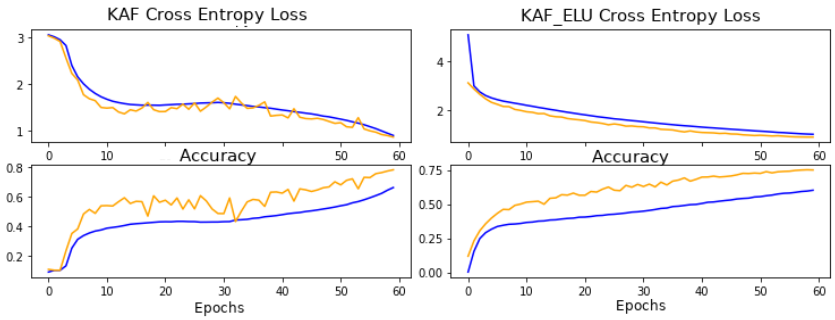


Figure 7.6. Accuracy and loss evolution during AT. The orange denotes the metric evaluated on the train set while with blue we denote the metric for the test set. In the left the plot for KAF training is shown and in the right for KAF_ELU. Notice how in both cases there is no sign of convergence at the 60-th epoch.

max-pooling layers, applying them three times during a forward pass. Even though they are very useful components in a neural network to decrease dimensionality and avoid redundant information 2.4, it is also true that such layers are non-smooth by

construction. In fact, they introduce jumps in the gradient computation, acting similarly to the previously discussed k-WTAs. Therefore, our attempt in exploiting the smoothness of novel activation functions might have failed a priori, due to the presence of non-smooth components in our architecture which canceled out our efforts. Notice how this argument could also explain why we had suprsingly poor results with ELUs and their KAF approximation.

In light of such considerations, we decided to repeat the experiment this time employing another architecture which was free from non-smooth components. A natural choice in this regard, due to its simplicity and great performances was to implement and evaluate a residual neural network (ResNet) 2.5.

7.2 Exploding Gradients with KafNets

For the implementation in TF2, we follow the architecure described in the original paper by He *et al.* (Ref. ResNet). In their study, authors design and evaluate two different ResNets, regardless of the number of layers, depending on the dataset used: one for ImageNet and the other for CIFAR10. Since we are using the latter dataset, we are going to stick to that specific ResNet. As for VGG, the basic structure of the architecure is a block of convolutions, made of two 3×3 convolutions interleaved by BN and non-linearities plus a residual connection between the input to the block and the output of the second convolution. Moreover, we divide the convolutive stage of the network in three sections articulated by their features map dimensionality which are $\{32, 16, 8\}$ respectively. Instead of using max-pooling to reduce the features map size we use strides of 2 in the first convolution of the block. The number of filters for each section are $\{16, 32, 64\}$ respectively. Before these sections there is another 3×3 convolutional layer which halves the input dimension. Finally, the network ends with a global average-pooling, a fully-connected layer with 10 units and a softmax. In total, the number of layers of such ResNet is given by:

$$6n + 2, \tag{7.1}$$

stacked weighted layers, where n denotes the number of residual blocks in each stage. Differently to the previous experiment with VGGs, this time we use the following data augmentation on the training set: 4 pixels are padded on each side of the image and a 32×32 crop is randomly sampled from the resulting image or its horizontal flip.

To check if we correclty implemented the described network, we try to replicate the results obtained by the authors in the paper for $n = \{3, 9\}$ therefore for a ResNet20 and a deep ResNet56 using the original training settings. The models are trained with a mini-batch size of 128 and for 150 epochs, with a learning rate of 0.1 and ReLU activation functions. At the end of the last epoch, our implementation for ResNet20 returns 91.8% of standard accuracy which matches the original results. Similarly, when scaling to ResNet56 we get 93.22% which, even if slightly worse than the originally reported 94%, is still sufficiently accurate. We then managed to have a working implementation for ResNet and we can start our evaluations.

Since we are mainly concerned in testing the model with other activation functions, we perform a sanity check with KAFs, to make sure that we dont experience drastically

changes in the outcome. Therefore we train, with equal settings, a KafResNet20 and a KafResNet56. Surprisingly, if with 20 KAF layers we get a resonable 90.07% of standard accuracy, that may be improved by tuning KAF parameters, in the second case, our 56-layered Kafnet is unable to learn anything at all from the data. As can

```
Epoch 1/150
391/391 [==============================] - 466s 1s/step - loss: 3.3430 - accuracy: 0.1000 - val_loss: 3.3450 - val_accurac
y: 0.1000
Epoch 2/150
391/391 [==============================] - 464s 1s/step - loss: 3.3431 - accuracy: 0.1000 - val_loss: 3.3432 - val_accurac
y: 0.1000
Epoch 3/150
391/391 [==============================] - 464s 1s/step - loss: 3.3430 - accuracy: 0.1000 - val_loss: 3.3432 - val_accurac
y: 0.1000
Epoch 4/150
391/391 [==============================] - 464s 1s/step - loss: 3.3431 - accuracy: 0.1000 - val_loss: 3.3431 - val_accurac
y: 0.1000
Epoch 5/150
391/391 [==============================] - 464s 1s/step - loss: 3.3430 - accuracy: 0.1000 - val_loss: 3.3431 - val_accurac
y: 0.1000
Epoch 6/150
391/391 [==============================] - 465s 1s/step - loss: 3.3431 - accuracy: 0.1000 - val_loss: 3.3430 - val_accurac
y: 0.1000
Epoch 7/150
391/391 [==============================] - 464s 1s/step - loss: 3.3430 - accuracy: 0.1000 - val_loss: 3.3431 - val_accurac
y: 0.1000
Epoch 8/150
390/391 [==============================>.] - ETA: 1s - loss: 3.3430 - accuracy: 0.1000
```

Figure 7.7

be seen from Fig. 7.7, after 10 epochs the classification is still performing randomly. Moreover, this trend does not change even if we let the network train longer or if we tune learning rates, KAF parameters or kernel weights initialization.

Since we know that the overall architecure works correctly when ReLUs are used, the problem needs to be related with the integration of KAFs. Nevertheless, we also know that KafResNet20 works flawlessly and thus, this issue only arise with the application of KAFs in deep scenarios. For this reason, we argue that the behavior we are facing is probably due to an issue with gradients updates in the network such as vanishing or exploding gradients.

To test if this is indeed the case, we log and analyze the distribution of the *Euclidean* norm for each weight, gradient pair inside the model during the first few epochs of the training. The analysis is carried out on every developed network, i.e., for both ResNet20 and ResNet56 when equipped with ReLU or KAF activation functions. To mitigate the running time, we only log the first epoch for KafResNet56 and the first 4 for the remaining architectures. After logging the inner statistics, we compare each model with their graphical visualization of the evolution of weights and gradients values. The idea is that, in the presence of vanishing or exploding gradients, we should be able to graphically spot where the problem happens, by means of macro variations in the KafResNet56 plots. All the visualization are made using the *TensorBoard* (Ref.) tool.

To showcase the effects of a potential gradient issue, we inspect three different areas of the network: the early layers, some of the middle layers, and the very last layers. In fact, wether we are dealing with exploding or vanishing gradients it would eventually become clear in the early stages of the network, however, since our goal is to understand where the phonomenon starts to emerge we need to inspect also the remaining layers.

As can be seen analyzing the first convolution and the first KAF Fig. 7.8 7.9, the norm of the gradient for the KafResNet56 is undefined (NaN) starting from the first training iteration. This is in turn reflected on the corresponding weight value which never gets updated. On the contrary, gradients and weights in the other architectures tend toward similar values 7.9.

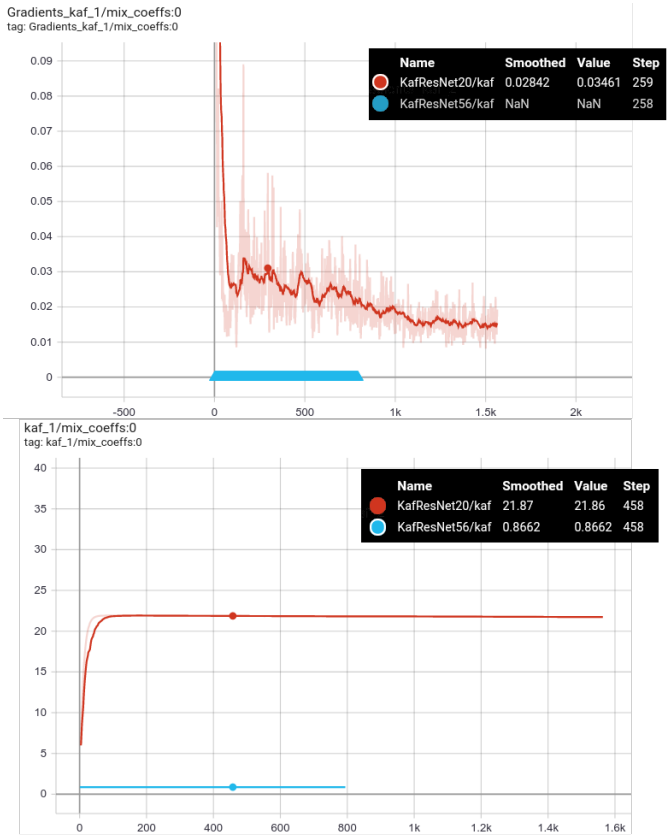


Figure 7.8

NaN values are usually the result of unrepresentable operations such as division by zero or square root of a negative number. In our specific case however, it is more likely that they are generated as a result of a vanishing or exploding gradient computation. Indeed, in either case, we would have a weight-update that underflows or overflows the numerical precision in place. Subsequently, the gradient computation for that specific weight will become undefined, producing, from that point backwards, in the backpropagation, only NaN values as the error flows through the network. We are then left to show if the observed NaNs are generating from an exploding or vanishing gradient problem.

As we continue to analyze deeper layers and their weights, we discover that, around the 15th hidden layer, gradients norm stops being undefined, becoming extremely large instead Fig. 7.10. The difference between the plotted norms is strongly noticeable. For both the kernel convolution and the mixing coefficients, KafResnet56 gradients norm is close to $2 \cdot 10^5$, thus being several orders of magnitude bigger than the same norm applied on shallow or ReLU equipped ResNets. The latter norm in fact, fluctuates between 0 and 1, even if it may appear as a straight single-valued line from the plot.

Our analysis seems to confirm our belief that what is actually happening is a gradient issue, in particular, the extreme grow of gradients which eventually lead to NaN values in the early layers is exaclty the description of exploding gradients,

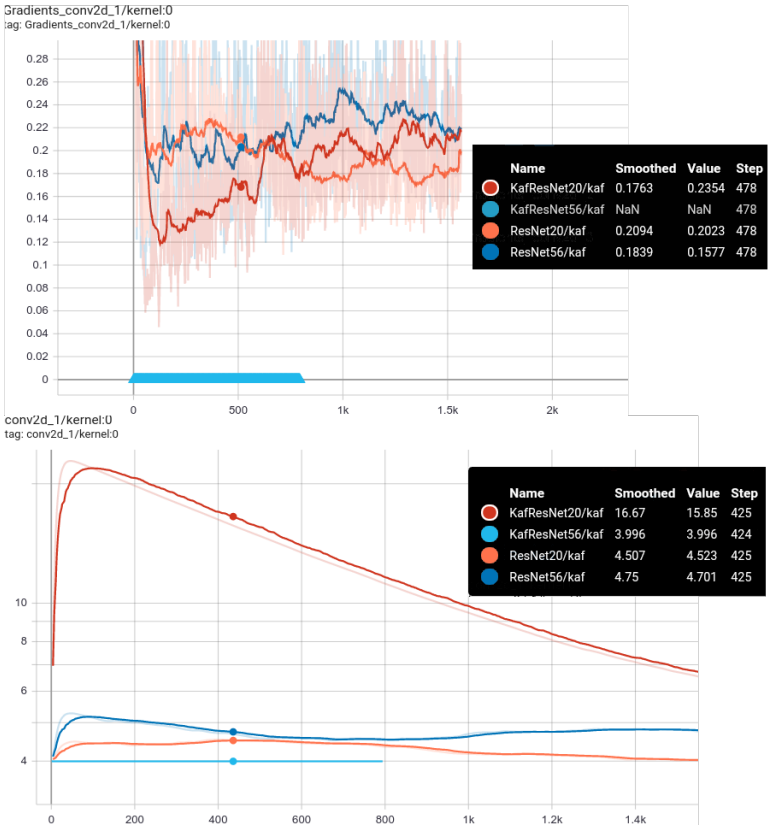


Figure 7.9

hence the inability to learn.

If we keep plotting towards the last layers, we observe how gradients gradually become more and more close to eachother Fig 7.11. Again, this hints to the fact that exploding gradients is actually caused by the large number of layers employed and not due to some strange, sudden gradient explosion with KAFs. More importantly, we deduce that such issue can be mitigated, if not completely avoided — as for KafResNet20 —, decreasing the number of layers.

Ultimately, it is important to mention that exploding gradients afflict many models in the field of machine learning (Ref. LSTM, ReLU) and thus, as a consequence, many techniques are available nowadays to try to tackle this obstacle. Even if we did try some of them, it is still plausible that a more careful design, such as the use of another kernel method for KAFs specifically targeted to prevent gradient issues, or the adoption of more sophisticated methods such as gradient clipping or spectral-norm regularization (Ref.), might have made possible to train a 56-layered KafNet. The code to reproduce the results discussed in this section can be found in *reskafnets_n_exploding_grads.ipynb*.

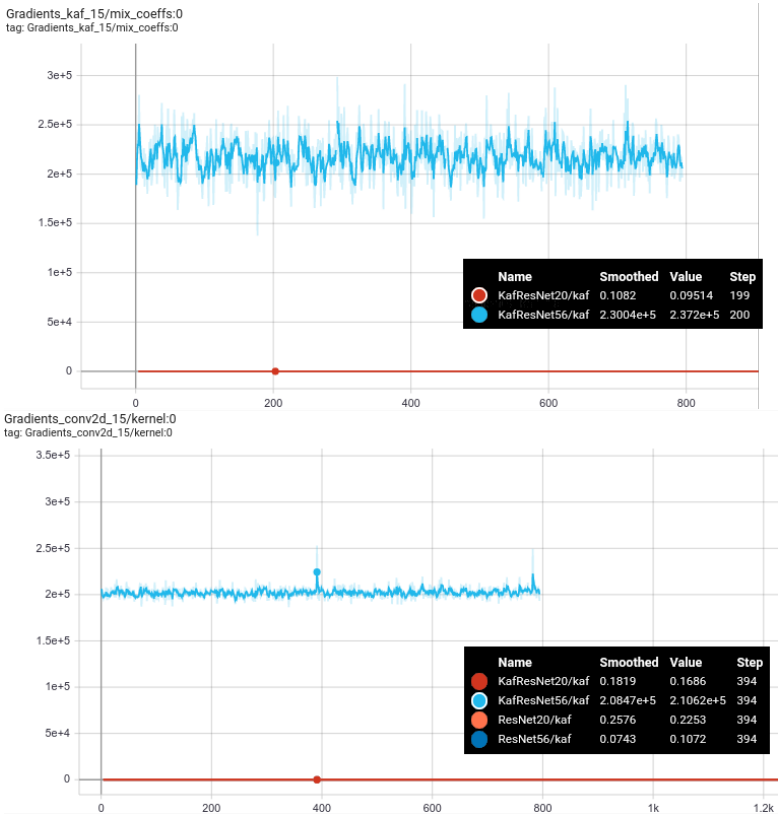


Figure 7.10

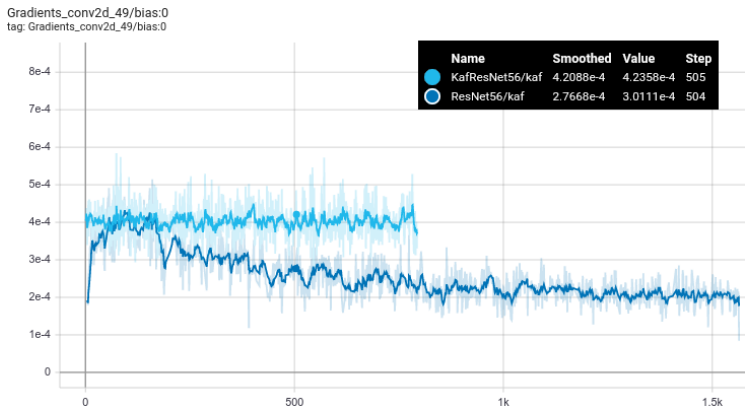


Figure 7.11

7.3 ResNet20 Inspired Architectures Results

In order to satifsy the initial requirement on the need for a smooth architecture, that would also be compatible with KAFs, we showed, in the previous section, how a relatively shallow residual neural network is ultimately a good choice. Despite the fact that deeper ResNets achieve better performances (Ref. ResNet article), using 20 layers is indeed already sufficient to classify correclty the 90% of the images in our dataset, beating any VGG we previously developed. It is then reasonable to expect

better results in terms of accuracy/robustness as well from a ResNet20. Moreover, our main intent is to assess the robustness properties that comes with KAFs and it is not to achieve state of-the-art results. Therefore, for our last experiment, we decided to evaluate the robustness of a ResNet20, as we change activation functions.

We follow the same procedure used in the case of VGGs to perform evaluations. Namely, we first adversarially train the network using the enhanced FGSM AT and then a full strenght PGD to craft adversarial examples. Differently from before, we extended the number of epochs during training to 80 epochs and, importantly, we augment the train set with data augmentation as described in the previous section on exploding gradients. The learning rate policy, as well as the optimizer, PGD and KAF hyperparameters are unchanged. The details for the model architecture are again the same provided in the previous section. Finally, the code to reproduce the following results can be found in *fbf_training_kafresnets.ipynb*

In order to give a fair comparison with the VGG case, we evaluate the network with the same set of activation functions, plus a KAF approximation for ReLU (KAF_ReLU). The outcome of each evaluation is given in Tab. 7.2. Moreover, an accuracy-robustness plot Fig. 7.12 allows for a graphical visualization of each of the produced trade-offs.

Activation	**Accuracy**	**Robustness**
ReLU	77.52%	50.17%
ELU	75.42%	51.69%
KAF	78.08%	52.23%
KAF_ReLU	**79.65**%	51.44%
KAF_ELU	78.69%	52.76%
Swish	<u>78.68%</u>	**<u>53.88%</u>**
KAF_Swish	79.21%	52.10%
KAF_Free	78.88%	52.07%

Table 7.2. Summary of the robustness gained from changing activation functions in an adversarially trained ResNet20. Bold denotes best result while underline best trade-off.

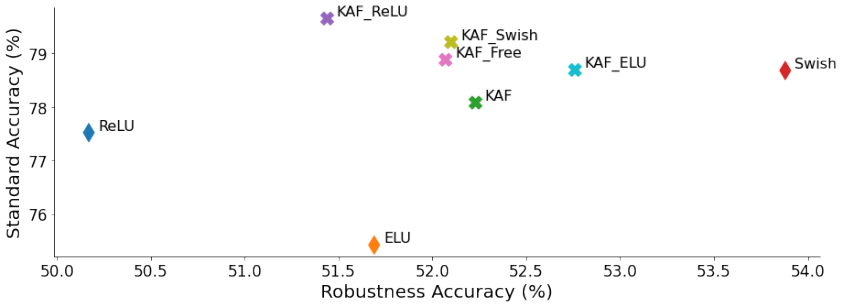


Figure 7.12. Robustness/accuracy visualization for a ResNet20 equipped with different activation functions. Fixed activation functions are shown with a thin diamond whereas non-parametric KAFs with a filled x. Swish achieves best robustness and KAFs generally tend to improve both robustness and accuracy with respect to a ReLU baseline.

Before we start to comment on each single activation function's score, it is

important to remark how, regardless of the activation function being used, the network significantly improves in robustness over the VGG case. In fact, even in the least performant case – corresponing to the ReLU baseline –, we are still outperforming the best 47.73% robustness obained with Swish equipped VGG. This highlights the importance of choosing the right architecture, which in our case corresponded to a non-smooth component free architecture, adding value to the smoothness thesis. Moreover, note how this comparison is especially meaningful since the standard accuracies and total number of weights between the two architectures are nearly the same, therefore, given equal resources, the ResNet seems to be really more robust.

A second key point to highlight concerns how, this time, ReLU is definitely acting as a base line. Indeed, any of the tested smooth activation functions, even the ELU, manages to substantially increase the robustness of the network with respect to the 50.17% achieved by ReLU. Again, this behavior is in line with (Ref. SAT) evaluations.

Furthermore, extending (Ref. SAT) observations, we discover that, in a much noticeable way than with VGG, the usage of KAFs in ResNet20 always results in a boost in standard accuray, wether we consider the baseline but, more in general, it applies to any fixed activation function since, approximating the initial shape of a KAF with a fixed activation function leads, in every instance, to greater accuracies Fig. 7.12.

Ultimately, the most remarkable result for our study concerns the robustness obtained with Swish functions. Achieving a total robustness of 53.88% and thus improving by a margin of almost 4 points from the baseline and also considerably from the second best performing functions (KAF_ELU), Swish is confirmed once again as the preferable function to pick for maximizing the robustness of a deep neural network. Moreover, Swish performs resonably well in terms of standard accuracy too, being less than one point away from the maximum reached with KAF_ReLU. Therefore, after having evaluated several activation functions and strucutrally different architectures, we argue that is tipically convenient to choose Swish activation functions when we want to build a robust neural network.

Chapter 8

Conclusions and Future Works

8.1 Conclusions

In this thesis, we addressed the problem of the importance of activation functions for the robustness of deep neural networks, with particular emphasis on the class of non-parametric activation functions known as kernel-based activation functions (KAFs). Our initial hyphotesis on the potential of KAFs in improving adversarial robustness was only marginally validated by the experiments we made. Even though it is still plausible that a fine-grained optimization of hyperparameters such as the dictionary size or the kernel-bandwidth, or even the regularization of the mixing coefficients, might have led to stronger results, we conclude that the main reason behind the observed lack in effectiveness, is to be found in the properties of the KAF itself. That is, as much as an user can exploit smoothness and flexibility to improve accuracy and, in particular, to strenght adversarial training, we should not forget that this also holds true from the adversary side. Enhanced flexibility means more resiliency against attacks, but it also means more space for the attacker to search for even stronger adversarial examples. The addiction of non-linearity in the output space of f_θ that comes with KAFs, can in fact be laveraged by any gradient-based attack, including PGD. On the contrary, fixed activation functions do not introduce new parameters or flexibility in the network. Therefore, only through smoothness, we can train a model against strong attacks, that cannot, loosely speaking, be made stronger by any adversary. It is this lack of unbalanceness towards the defense side that is missing with KAFs, preventing them to overcome, for instance, the performance of Swish activation functions.

As a side effect, the results reported in this thesis can also serve as a contribution to the bag of evidences for some of the related works that we mentioned and used, especially for those recent ones, which may benefit from additional empirical validations. Specifically, from the VGG and ResNet robustness evaluations, it clearly emerged how KAFs are likely to improve the generalization properties over fixed activation functions, as first described in (Ref. Scardapane KAF). Similarly, in these experiments we also find that the Swish activation function consistently outperforms other activation functions in terms of robustness, giving credits to (Ref. SAT). Lastly, we were able to implement the fast adversarial training method described in (Ref. FBF) in another framework and still matching the reported running-times.

8.2 Future Works

Many different ideas, extensions, and experiments have been left for the future due to lack of time and resources. In particular, the following is a list of tracks that deserve to be tested:

- Despite KAFs did not stand out as the optimal activation function to be used in an adversarial training scenario, it is worth to recall that adversarial training is just one out of different known methods to tackle the problem of Robust Optimization. Another approach that is known to be effective and in which KAFs might find a successful application, is the area of certified defenses. In such particular setting, the flexibility of KAFs might be exploited to devise defenses that are provable to be secure against norm-bounded perturbations. In this regard, authors in (Ref. Lipschitz-Margin Training) propose a training procedure to enlarge the provably guarded area around data points by means of a minimization of the Lipschitz constant of the network. In (Ref. MICAI2002: Advances in Artificial Intelligence) an explicit upper bound for the Lipschitz constant of a KAF is given.

- Previously known to be a computationally prohibitive task to accomplish, the adversarial training of a deep neural network for ImageNet is now becoming ever more accessible due to the progress of the hardware and the development of efficients procedures such as the FSGM adversarial training used (Ref. FBF). Therefore, scaling the experiments discussed in this thesis to ImageNet to check if the results are coherent with those of CIFAR10 is advisable.

- Other than PGD, perform other famous attacks such as *C&W* (Ref.), *DeepFool* (Ref.) or, more importantly, *adaptive attacks* to assess the robustness of adversarially trained Kafnets. Adaptive attacks are attacks that were specifically designed to target a given defense and it is nowadays become the standard to perform true robustness evaluations (Ref. Carlini et al. Towards evaluating). Indeed, just as was showed for k-WTA (Carlini https://arxiv.org/pdf/2002.08347.pdf), there might a subtle gradient-based attacks which could target the presence of trainable activation functions in the network to produce adversarial examples capable of making adversarial training worse than when using, for instance, ReLU activation functions. Several best practices have been proposed to help spotting possible flaws in a defense and design adapative attacks (Ref. https://arxiv.org/pdf/2002.08347.pdf).