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 11, 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

1 Introduction **1**
1.1 Intriguing Properties of Neural Networks . . . 1
1.2 Smooth Activation Functions and Robustness . . . 1
1.3 Structure of the Thesis . . . 1

2 Fundamentals **3**
2.1 Deep Neural Networks . . . 3
2.1.1 Definition . . . 3
2.1.2 Training . . . 5
2.1.3 Activation Functions . . . 9
2.1.4 CNNs: Convolutional Neural Networks . . . 11
2.1.5 From Neural Networks to Deep Neural Networks . . . 13
2.2 Adversarial Examples Theory . . . 16
2.2.1 Another Optimization problem . . . 17
2.2.2 Fast Gradient Sign Method . . . 18
2.2.3 Projected Gradient Descent . . . 19
2.2.4 White, Grey and Black Box Attacks . . . 20
2.3 Defenses . . . 20
2.3.1 Detection Methods . . . 21
2.3.2 Robust Optimization . . . 22
2.3.3 Provable Robustness . . . 22
2.4 Kernel Based Activation Functions . . . 23
2.4.1 Adaptive Piece-Wise Linear Activation Functions . . . 24
2.4.2 Spline Activation Functions . . . 24
2.4.3 Maxout Functions . . . 25

3 Related Works **27**
3.1 K-Winners Take All . . . 27
3.2 Smooth Adversarial Training . . . 27

4 Solution Approach **29**
4.1 Lipschitz Constant Approach . . . 29
4.2 Fast is Better than Free Adversarial Training . . . 29

5 Evaluation **31**
5.1 VGG Inspired Architectures Results 31
5.2 Explofing Gradients with KafNets 31
5.3 ResNet20 Inspired Architectures Results 31

6 Future Works **33**

7 Conclusions **35**

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

Description of the remaining chapters

Chapter 2

Fundamentals

In this chapter, the basic concepts needed to understand the main arguments for the thesis are introduced. Pointers to more appropriate and detailed resources on the topics are given throughout

2.1 Deep 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.) and, more recently, Deep Neural Networks (ref.). During the last two decades Deep Neural Networks 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).

2.1.1 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:

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

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.

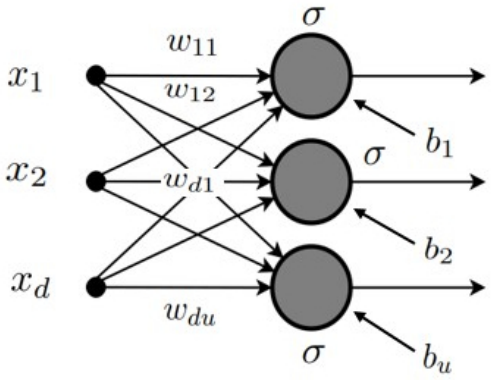

Figure 2.1. Graphic representation of a one layer NN also know as MLP(ref.)

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.

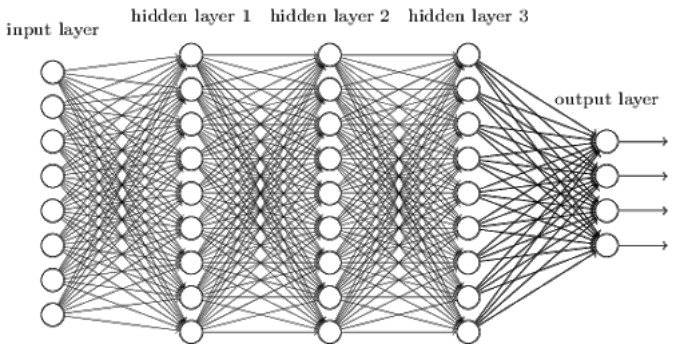

Figure 2.2. A Feedforward Neural Network with 3 hidden layers(ref. Michael A. Nielsen, Neural Networks and Deep Learning, Determination Press', 2015)

2.1.2 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?

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.*

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,

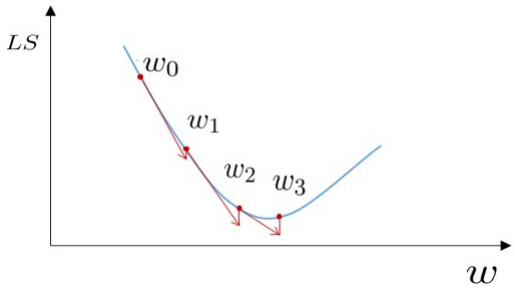

Figure 2.3. A 1-D representation of 4 gradient descent steps

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.

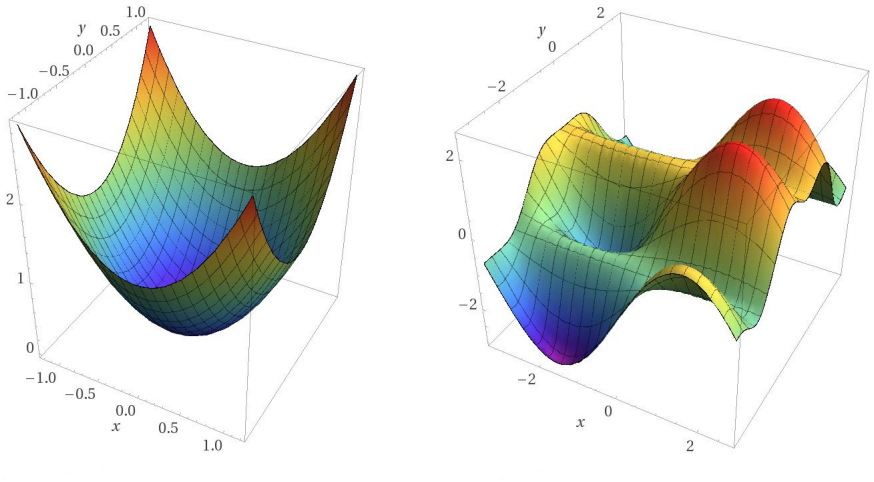

Figure 2.4. Convex and Non-convex optimization landscapes

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. (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)

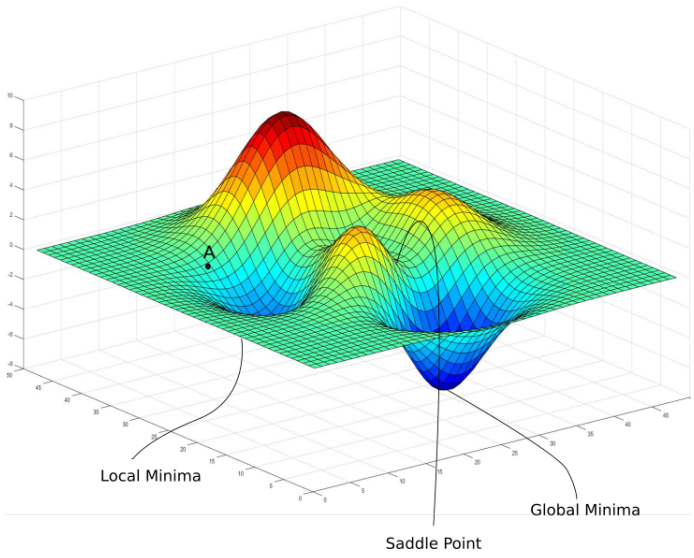

Figure 2.5. Visualization of global, local and saddle points. How can A reach the global minimum?

2.1.3 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,

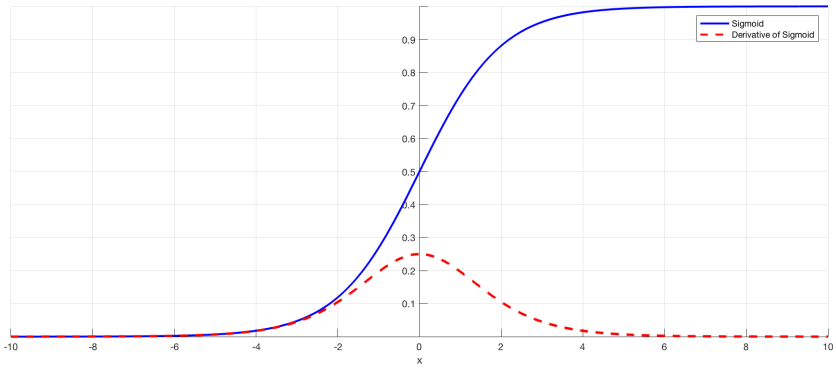

Figure 2.6. Plot of the Sigmoid function and its first derivative.

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

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

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.):

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

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:

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

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.):

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

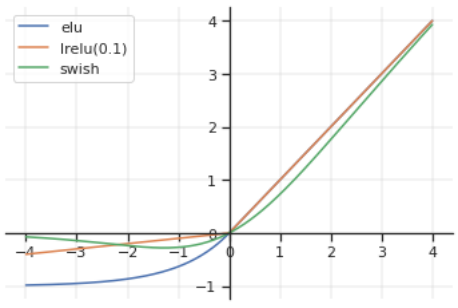

Figure 2.7. ELU, LeakyReLU($\alpha = 0.1$) and Swish functions plotted together. It can be seen that they mostly differ for negative values whereas behaving very similar to ReLU for positive arguments.

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.

2.1.4 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

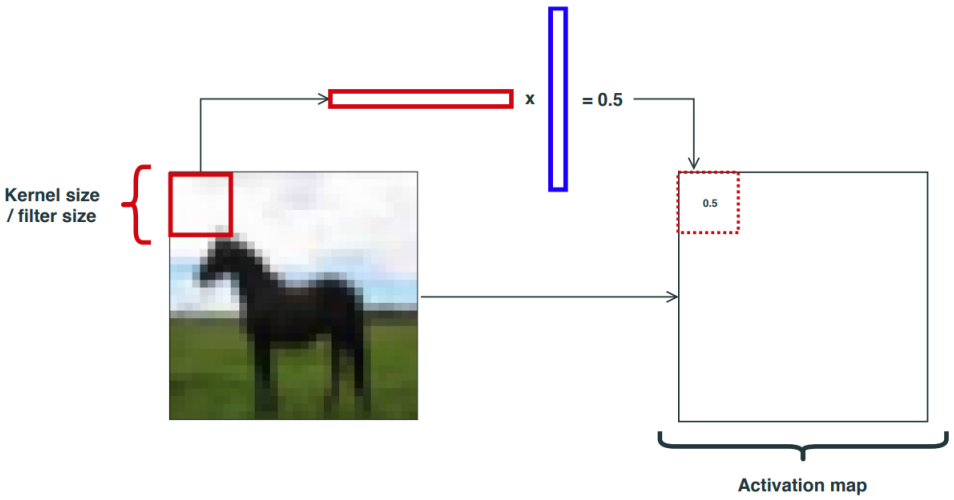

Figure 2.8. A convolution that produces the first element of the activation map for the given filter.

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

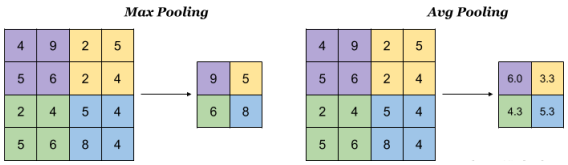

Figure 2.9. Max (left) and Average (right) pooling layer with 2x2 window size.

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:

- 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*:

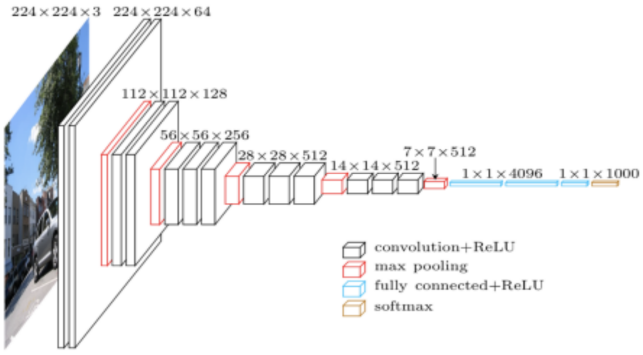

Figure 2.10. General Architecture of CNN for ImageNet (Ref.).

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.

2.1.5 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

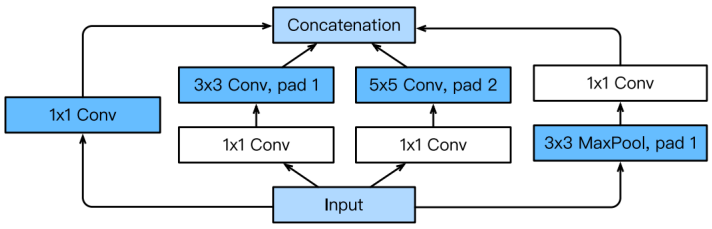

Figure 2.11. Inception Block overview, 1×1 convolutions are used to reduce the number of filters lowering complexity. Source: Dive into Deep Learning, chapt. 7.4

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

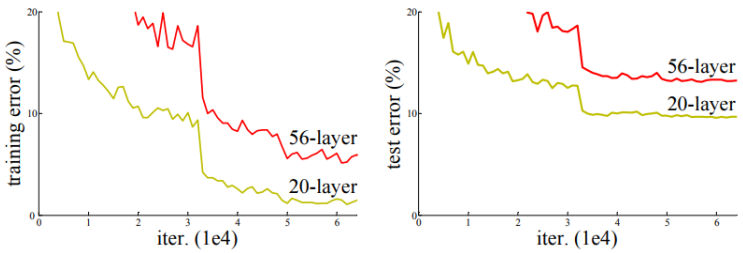

Figure 2.12. Source: Deep Residual Learning for Image Recognition

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

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.

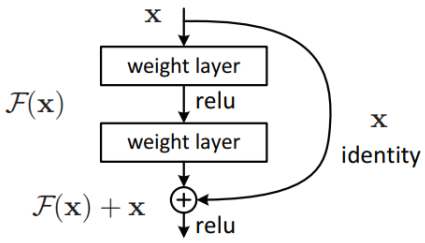

Figure 2.13. Source: Deep Residual Learning for Image Recognition

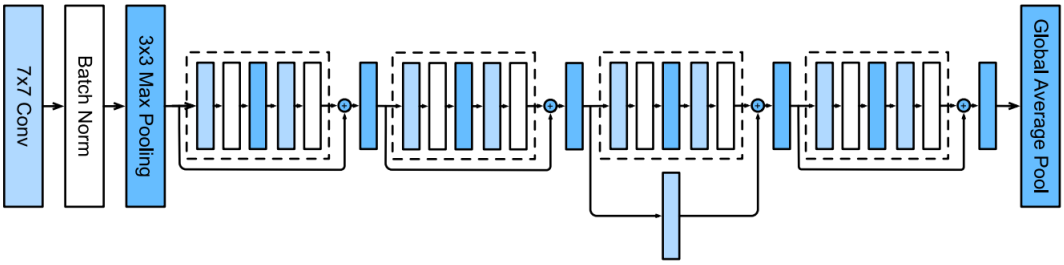

Figure 2.14. Source: Dive Deep into Deep Learning

2.2 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.

2.2.1 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:

$$\max_{\hat{x}} \mathrm{LS}\left(f_\theta\left(\hat{x}\right), y\right), \tag{2.24}$$

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:

$$\max_{\delta \in \Delta} \mathrm{LS}\left(f_\theta\left(x+\delta\right), y\right), \tag{2.25}$$

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:

$$\|z\|_\infty = \max_i |z_i| . \tag{2.26}$$

Thus the space of allowed perturbations becomes:

$$\Delta = \{\delta : \|\delta\|_\infty \leq \epsilon\}, \tag{2.27}$$

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:

$$\max_{\delta, \|\delta\|_\infty \leq \epsilon} (\mathrm{LS}(f_\theta(x+\delta), y) - \mathrm{LS}(f_\theta(x+\delta), y_{target})). \tag{2.28}$$

2.2.2 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:

$$\delta^{t+1} = \delta^t + \alpha \cdot \nabla_{\delta^t} \mathrm{LS}\left(f_\theta(x+\delta^t), y\right), \tag{2.29}$$

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

$$\delta^{t+1} = \mathrm{clip}(L_\infty, \delta^{t+1}, [-\epsilon, \epsilon]), \tag{2.30}$$

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:

$$\delta = \epsilon \cdot sign(\nabla_x \operatorname{LS}(f_\theta(x), y)), \tag{2.31}$$

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.

2.2.3 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 2.29 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**
    $\delta \leftarrow \delta + \alpha \cdot \nabla_\delta \operatorname{LS}(f_\theta(x + \delta), y)$
    $\delta \leftarrow \mathcal{P}(\delta)$
  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 2.25. 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).

2.2.4 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 Crypthography, 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 trainig set that was used to train the classifier and another surrogate classifier is trained using surrogate data (Ref. Survey on current defenses).

2.3 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.

2.3.1 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. 2.15. 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.

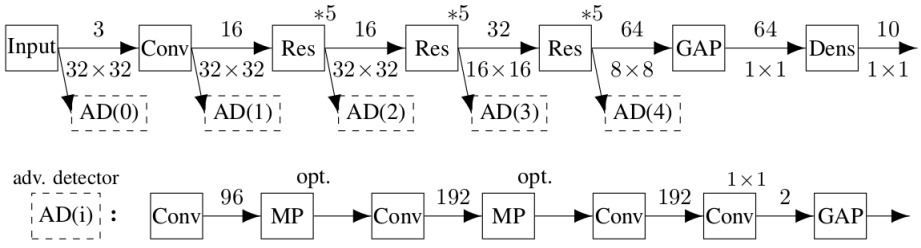

Figure 2.15. Metzen *et al.* (Ref.) used a ResNet as original classifier, plus dections is made thanks to many interleaving detectors between each residual block. Each detector is implemented as a DNN that learns how to spot the presence of an attack by looking at the activations layer's distributions during training.

2.3.2 Robust Optimization

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:

$$\min_{\theta} \frac{1}{|S|} \sum_{x,y \in S} \max_{\delta \in \Delta} \mathrm{LS}\left(f_\theta(x+\delta), y\right), \tag{2.32}$$

that goes under the name of *Robust Optimization*.

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 Learning* (AL) 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 AL 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 AL, 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 AL, proposing approximations and variations of it (Ref. Free AL)(Ref. Fast is better than free).

2.3.3 Provable Robustness

Provable defenses try to theoretically find certificates in distances or probabilities to certify the robustness of DNNs. Can 2.32 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:

$$\mathcal{L}_{adv} = \max_{\sigma \in \Delta} \left\{ \max_{i \neq y} f_\theta(x) \left(x+\delta\right)_i - f_\theta(x) \left(x+\delta\right)_y \right\}, \tag{2.33}$$

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

$$C(x, f_\theta) < 0 \tag{2.34}$$

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 $x, r \in \mathcal{X}$:

$$\|f_{\theta_i}(x) - f_{\theta_i}(x+r)\|_p \leq L_i \|r\|_p. \tag{2.35}$$

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:

$$\|f_\theta(x) - f_\theta(x+r)\|_p \leq L \|r\|_p, \tag{2.36}$$

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 usally based on generally intractable methods or ad-hoc methods.

2.4 Kernel Based 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 be trained along with the others weights of the network allowing wach 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, we deal 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.):

$$\mathrm{SReLU}(x) = \begin{cases} t^r + a^r\left(x - t^r\right) & \text{if } x \geq t^r \\ x & \text{if } t^r > x > t^l \\ t^l + a^l\left(x - t^l\right) & \text{otherwise} \end{cases}, \tag{2.37}$$

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 grater modelling capabilities, what if we allow them to model any continuous segment? This question is 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 properies and experimental results.

2.4.1 Adaptive Piece-Wise Linear Activation Functions

Introduced in (Ref. Agostinelli et al 2014), APL generalize the SReLU 2.37 activation function summing up S parametrized linear segments that are learned under the constraint that the resulting function is continuous:

$$\mathrm{APL}(x) = max\{0, x\} + \sum_{i=1}^{S} a_i \max\left\{0, -x + b_i\right\}, \tag{2.38}$$

where S is a user-defined value and a_i are the parameter to learn. APL add $2S$ 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 functions, but only saturating ones. Moreover, APLs introduce S non-differentiable points which may harm backpropagation.

2.4.2 Spline Activation Functions

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

$$\sigma(x) = \sum_{i=0}^{S} a_i x^i, \tag{2.39}$$

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 very 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 step size Δx and call *knots* the corresponding y-valules $\{\text{SAF}(x_i)\}_0^S$. 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 a SAF at t:

$$\text{SAF}(t) = \mathbf{u}^T \mathbf{B} \mathbf{q}_k, \tag{2.40}$$

where $\mathbf{u} = \left[u^P, u^{P-1}, \ldots, u^1, 1\right]^T$, P 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 iwith SAFs is that each knot has only local effect on the overall shape. Therefore, their training allows for faster and better convergence to the optimal with respect to the initialized knots. As for polynomial interpolation, SAFs can in principle approximate any smooth function. A drawback, however, comparing to ALF, is that regularization cannot be explicitly implemented for knots.

2.4.3 Maxout Functions

ASNSANS

Chapter 3

Related Works

3.1 K-Winners Take All

3.2 Smooth Adversarial Training

Chapter 4

Solution Approach

Comparing the activations's distributions for different activation functions (ReLU, KWTA, Kafs) seem to suggest Kafs might be good candidates to improve model robustness

4.1 Lipschitz Constant Approach

On the limitations of current Lipischitz-Constant based approaches especially when involving Kafs

4.2 Fast is Better than Free Adversarial Training

Adversarial training (Madry et al.) and current methods to improve the efficiency (Fast is better than free)

Chapter 5

Evaluation

5.1 VGG Inspired Architectures Results

5.2 Explofing Gradients with KafNets

The exploding gradients problem with KafResNet, why is it happening? (still to clarify)

5.3 ResNet20 Inspired Architectures Results

Chapter 6

Future Works

Different Kernels, resolve the exploding gradient problem and scale to ImageNet
Perform more adaptive attacks to assess the robustness of kafresnets as is the current standard (Carlini et al.)

Chapter 7

Conclusions

This thesis tries to add to the bag of evidences in literature that smoother architectures might benefit improvements in adversarial resiliency