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

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 6, 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$_\varepsilon$ 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

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: Search Trees (ref.), Support Vector Machines (ref.), Clustering (ref.) and, more recently, Neural Networks (ref.). During the last two decades 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 $S = \{(\mathbf{x_i}, y_i)_{i=1}^n\}$ with $\mathbf{x_i} \in \mathcal{X}$ and $y_i \in \mathcal{Y}$ such that

$$f_\theta \sim f$$

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

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

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 applied at the end is the

σ 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.*

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

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

$$\sigma(W_j^\intercal x + b_j) = \begin{cases} 1 & W_j^\intercal x \geq -b_j \\ 0 & W_j^\intercal x < -b_j \end{cases} \tag{2.1}$$

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.

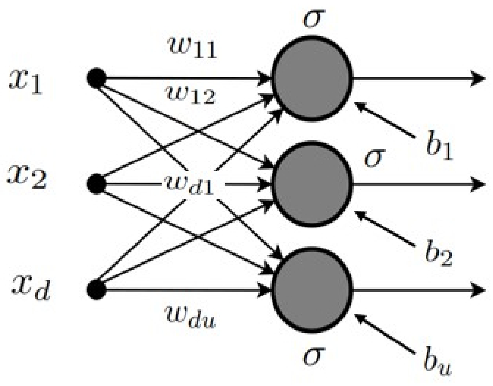


Figure 2.1. Graphic representation of a one layer NN also know as MLP(ref.)

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, $z_0 = \mathbf{x}$ and $z_l = \sigma_l(W_l z_{l-1} + b_l)$ we write a L-layered $f_{\mathbf{W},\mathbf{b}}$:

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

With $\mathbf{W} = \{W_1, \ldots, W_L\}$ and $\mathbf{b} = \{b_1, \ldots, b_L\}$. 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.

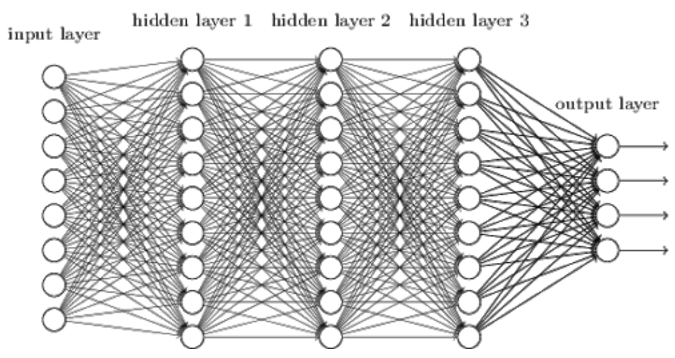


Figure 2.2. A Feedforward Neural Network with 3 hidden layers(ref. Michael A. Nielsen, Neural Networks and Deep Learning, Determination Press', 2015)

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: $-\sum_{c=1}^{C} y_c \log(f_\theta(x)_c)$ for mulicategory 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

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(f_\theta(x_i), y_i) \tag{2.2}$$

which is called *empirical risk minimization.*

Gradient Descent

So far we have described, given an input $\mathbf{x}$, 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 2.2 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^l_j to the j-th element of i^l so that the neuron will output $\sigma\left(i^l_j + \Delta i^l_j\right)$, this change will eventually propagates in the network causing the overall loss LS to change by an amount $\frac{\partial LS}{\partial i^l_j}\Delta i^l_j$. For brevity, denote the gradient of the weighted input on the j-th neuron $\delta^l_j = \frac{\partial LS}{\partial i^l_j}$, then the following holds:

$$\delta^L_j = \frac{\partial LS}{\partial i^L_j} = \frac{\partial LS}{\partial z^L_j}\frac{\partial z^L_j}{\partial i^L_j} = \frac{\partial LS}{\partial z^L_j}\sigma'_L(i^L_j) \tag{2.3}$$

and equally, taking into account the whole output layer

$$\delta^L = \nabla_{z^L} LS \odot \sigma'(i^L) \tag{2.4}$$

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:

$$\delta^l = \left((W^{l+1})^T \delta^{l+1}\right) \odot \sigma'(i^l) \tag{2.5}$$

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 2.4 with 2.5 we can compute the gradient δ^l for any layer in the network. We start by using 2.4 to compute on the last layer, then apply equation 2.5 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).

$$\frac{\partial LS}{\partial b^l_j} = \delta^l_j \tag{2.6}$$

$$\frac{\partial LS}{\partial w^l_{i,j}} = z^{l-1}_i \delta^l_j. \tag{2.7}$$

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:

$$w_l^t \to w_l^{t+1} = w_l^t - \frac{\eta}{n}\sum_j \frac{\partial LS_{\mathbf{x}_j}}{\partial w_l^t} \tag{2.8}$$

$$b_l^t \to b_l^{t+1} = b_l^t - \frac{\eta}{n}\sum_j \frac{\partial LS_{\mathbf{x}_j}}{\partial b_l^t} \tag{2.9}$$

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,

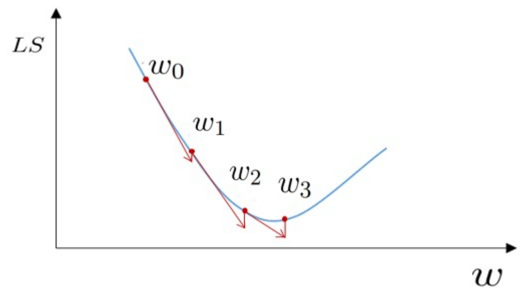


Figure 2.3. A 1-D representation of 4 gradient descent steps

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, when 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.

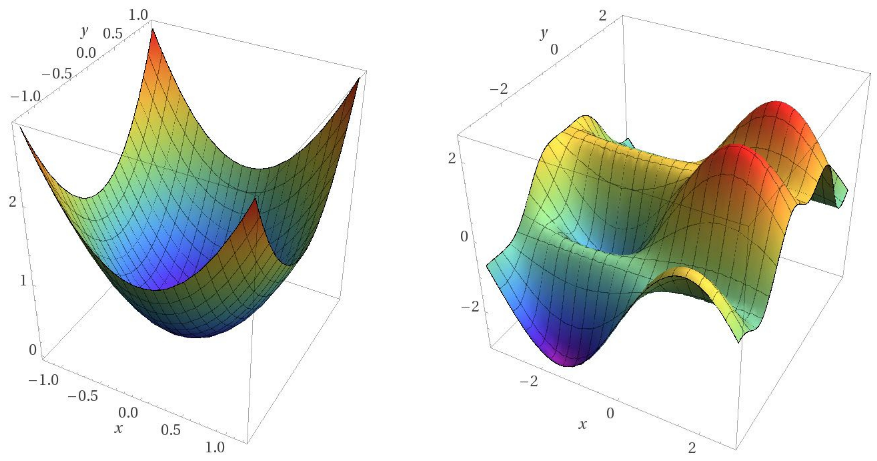


Figure 2.4. Convex and Non-convex optimization landscapes

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 2.2 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)

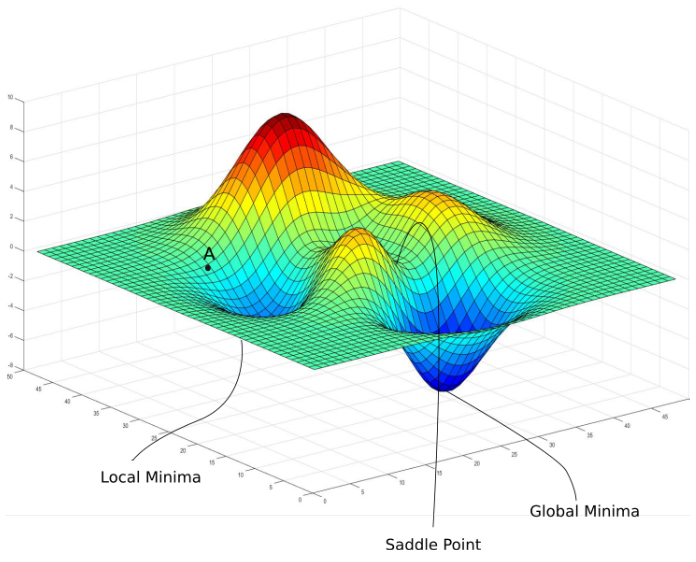


Figure 2.5. Visualization of global, local and saddle points. How can A reach the global minimum?

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 2.1 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.)

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{2.10}$$

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

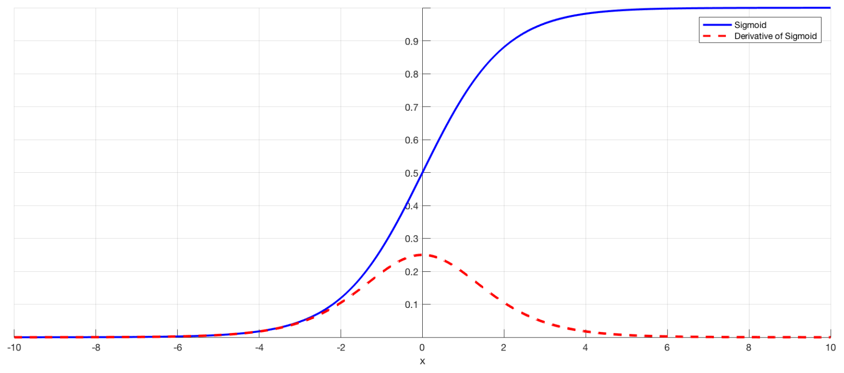


Figure 2.6. Plot of the Sigmoid function and its first derivative.

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

$$\lim_{x\to-\infty} \sigma(x) = c, \quad \lim_{x\to\infty} \sigma(x) = 1. \tag{2.11}$$

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

$$\tanh(x) = \frac{\exp\{x\} - \exp\{-x\}}{\exp\{x\} + \exp\{-x\}}, \tag{2.12}$$

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

$$\mathrm{ReLU}(x) = \max(0, x), \tag{2.13}$$

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.):

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}, \tag{2.14}$$

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:

$$\mathrm{ELU}(x) \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp\{x\} - 1) & \text{otherwise} \end{cases}, \tag{2.15}$$

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.):

$$\mathrm{swish}(x) = \mathrm{sigmoid}(x) * x, \tag{2.16}$$

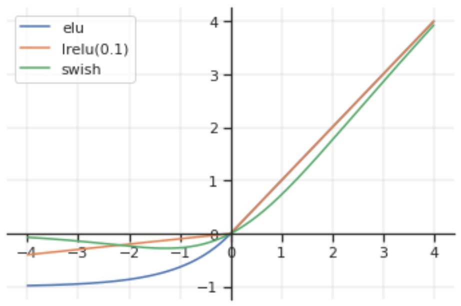


Figure 2.7. ELU, LeakyReLU($\alpha = 0.1$) and Swish functions plotted together. It can be seen that they mostly differ for negative values whereas behaving very similar to ReLU for positive arguments.

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

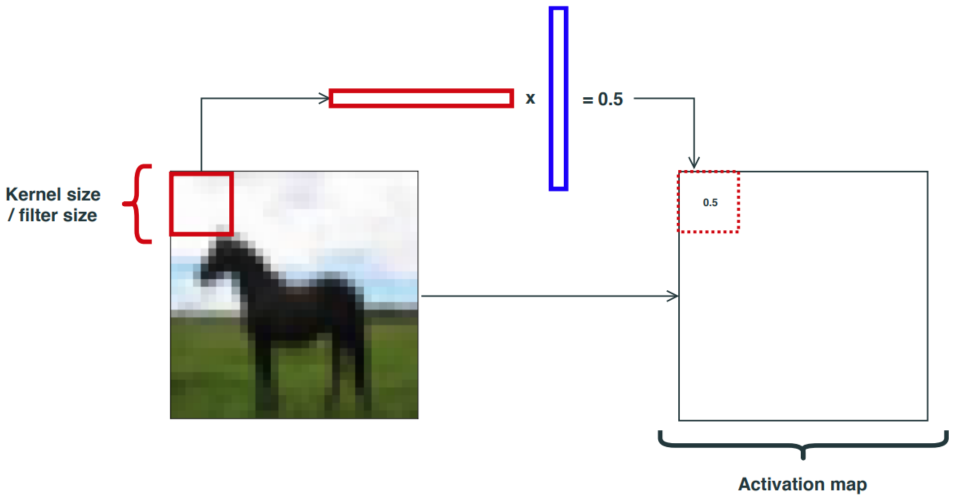
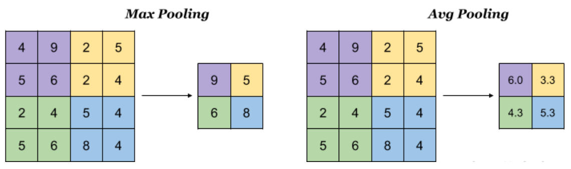


Figure 2.8. A convolution that produces the first element of the activation map for the given filter.

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



4	9	2	5
5	6	2	4
2	4	5	4
5	6	8	4

9	5
6	8

6.0	3.3
4.3	5.3

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*:

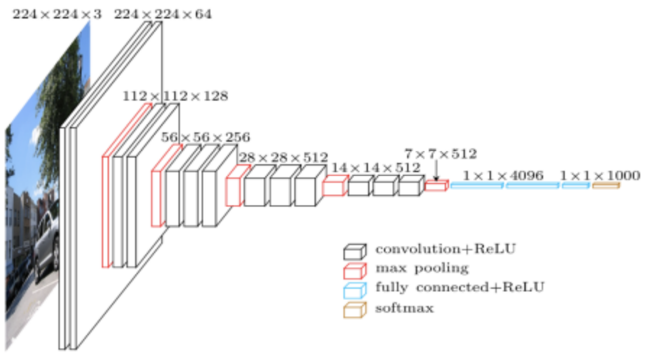


Figure 2.10. General Architecture of CNN for ImageNet (Ref.).

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. 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 image randomly sampled transformations 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:

$$\hat{z^l} = z^l \odot m, \tag{2.17}$$

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* value during training:

$$\mathbb{E}[\hat{z^l}] = p \cdot z^l. \tag{2.18}$$

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

2.2 Adversarial Examples Theory

Formal definition of what is an adversarial attacks plus currently well known attacks

2.3 Defenses

Review of the literature on defenses to improve robustness: provable robustness, adversarial training

2.4 Kernel Based Activation Functions

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