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 2, 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

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

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.

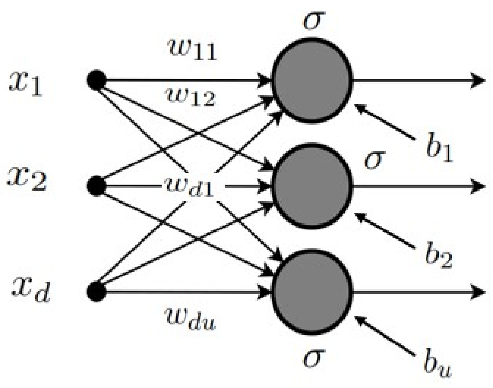

Figure 2.1. Graphic representation of a one layer NN also know as MLP(ref.)

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.

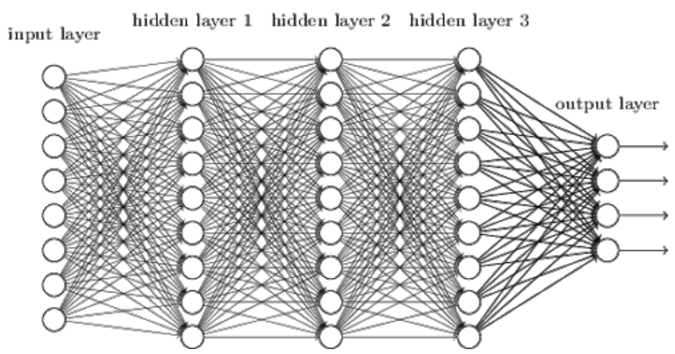

Figure 2.2. A Feedforward Neural Network with 3 hidden layers(ref. Michael A. Nielsen, Neural Networks and Deep Learning, Determination Press', 2015)

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 'real' 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

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(f_\theta(x_i), y_i) \tag{2.1}$$

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 2.1 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:

$$\delta^L_j = \frac{\partial LS}{\partial i^L_j} = \frac{\partial LS}{\partial z^L_j}\frac{\partial z^L_j}{\partial i^L_j} = \frac{\partial LS}{\partial z^L_j}\sigma'_L(i^L_j) \tag{2.2}$$

and equally, taking into account the whole output layer

$$\delta^L = \nabla_{z^L} LS \odot \sigma'(i^L) \tag{2.3}$$

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:

$$\delta^L_j = \frac{\partial LS}{\partial i^L_j} = \frac{\partial LS}{\partial z^L_j}\frac{\partial z^L_j}{\partial i^L_j} = \frac{\partial LS}{\partial z^L_j}\sigma'_L(i^L_j) \tag{2.4}$$

what does it mean to be able to compute partial derivatives of the loss ? it means be able to understand where and how the loss decreases and thus we can exploit such information to find optimal weights values, this is the idea behind the gradient descent algorithm.

2.1.3 Activation Functions

2.1.4 CNNs: Convolutional Neural Networks

2.1.5 From Neural Networks to Deep Neural Networks

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