Adversarial-learning on shallow-neural-networks

Project Report

Marc Maurice Roland Moreaux/mi105f15

Aalborg University
Electronics and IT

This master thesis was written using LaTeX. The networks figures present on this report were draw with a LaTeX package called tikz. The plots, bar diagram and images were created with Matplotlib, a rendering package for Python. The simple example was fully written in Python and the broader models trained with Pylearn2. Finally most of the files are under revision control on a git repository hosted on Github.

Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Adversarial-learning on shallow-neural-networks

Theme:
Machine Intelligence

Project Period:
Spring Semester 2015

Project Group:
mi105f15

Participant(s):
Marc Maurice Roland Moreaux

Supervisor(s):
Manfred Jaeger

Copies: 2

Page Numbers: 57

Date of Completion:
June 7, 2015

Abstract:

On this thesis we propose to compare a very classic method for learning shallow-neural-networks and another one based on adversarial-learning. We propose to compare and analyze the benefits of one toward the other.

Contents

Preface

This dissertation is an original intellectual product of the author, Marc Moreaux.

I would like to enjoy the opportunity to thank Manfred Jaeger for our discutions and his support all along this semester's thesis. I would also like to thank Iann Goodfellow for the few email we exchanged that led me to do this thesis.

This thesis is a further inversigation on adversarial-learning as refered by Iann Goodfellow and is targeting any person intersested in this thechnique applied to neural-networks.

Aalborg University, June 7, 2015

Marc Maurice Roland Moreaux
<mmorea13@student.aau.dk>

Chapter 1

Introduction

This machine learning master thesis is about neural-networks (section 2). We'll investigate on a technique called adversarial-learning (section 3) to know if it can help neural-networks in performing better.

1.1 Motivation

At the very beginning of this thesis, there is me, the writer. I wanted, from the very first day, to work on neural-networks and more specifically on deep-learning. I wanted so, because neural-networks has become in the last decade the most efficient algorithm at classifying images. I was then searching for a topic. I discovered the work made by Ian J. Goodfellow, Jonathon Shlens and Christian Szegedy. In their publication named "Explaining and Harnessing Adversarial Examples"[4] they motivate that adversarial learning could improve the accuracy of some given neural-networks. After contacting one of the authors of this publication, it appeared that it would be a good contribution to validate their observations on other datasets, since most of their test are based on the MNIST dataset[8]. Therefore, the following thesis is about further understanding and validating the proposed adversarial-learning method described in [4] with different parameter and different databases.

1.2 Methodology

We are therefore going to investigate on how is adversarial-learning performing on specific neural-networks called shallow-neural-networks. For the investigation we'll evaluate the impact of different parameters on the model, we are going to compare the adversarial-model with some others and will also evaluate few adversarial-models on different databases. On the same time, we are going to investigate on the reasons underlining the adversarial performances. To do so, we

will first re-implement the proposed solution of [4] in such a way that we isolate adversarial-learning from other techniques used in the paper. At this point we will emphasize the benefits of adversarial-learning.

We will then move to another dataset similar to the first one, there will also be 10 different classes of images to classify. That dataset will be the CIFAR10[7]. Again, some knowledge will be acquired from this experience. Finally, we will test the adversarial-learning on a dataset of other nature. The one we will try is a tree location dataset: the Covertype dataset[1]. The results obtained from this dataset bring new insights on adversarial-learning.

1.3 Expected outcome

In their paper, Ian J. Goodfellow, Jonathon Shlens and Christian Szegedy, described classic shallow-neural-networks to be too linear in what they learn. by linear they meant that the classifier was too much feating the data distribution given on the train-set. With adversarial-learning, they aim at learning a classifier relying less on the inputs data as such but better relying on the idea of the class.

At the end of the report, one can expect to better understand adversarial learning. One can expect understanding the advantages of adversarial-learning on a more classic approach to learning. Also, we'll demonstrate that this learning technique generalizes to other datasets.

1.4 Adversarial-learning

Until now, we've been speaking a lot of adversarial-learning without explaining what it is about. Lets start off with mentioning what an adversarial sample to a classifier would be. Imagine you have full knowledge of a given classifier. Starting from there, you can create samples to this classifier that are going to give some predictions. Knowing how does the classifier behaves, you can creates samples leading your classifier to some mistakes. This are adversarial samples. In machine learning, adversarial-learning is about creating a classifier resisting to adversarial samples.

In the publication that interest us[4], the authors noticed that twisting each pixels of the input images by some very specific values (the adversarial values), an accurate classifier could be easily fooled. Lets detail : Consider you have a classifier able to recognize the MNIST dataset and this classifier makes 10 errors when guessing the classes of 200 samples (95% accuracy). Now, if you modify each pixels of these 200 samples by a little value, the classifier makes 180 errors (10% accuracy). You've fooled your classifier with adversarial samples.

[1] `https://archive.ics.uci.edu/ml/datasets/Covertype`

The adversarial-learning algorithm we use is going to learn from adversarial samples. Therefore, at any time the algorithm want to predict the class of a sample, he will predict the class of the adversarial sample. It's with this approach that we aim at augmenting the accuracy of the classifier.

1.5 Running example, the 'simple example'

On the next sections we will explain what are neural-networks, how do we train them and how adversarial-learning apply to them. All along these explanations we will use an example called the "simple example" to illustrate our definitions.

The **task** of our example is to predict the class of an image. There is 3 types of images, a *dot*, a *column* and a *comma*. When a sample is called to be a *dot*, it is referred as belonging to class $y = 1$. When it's a *comma*, a class $y = 2$ and a *column* a class $y = 3$.

The **inputs** to our examples are images. They are gray-scaled and have 3 pixels with values in range $[0, 1]$. The 3 pixels are aligned in a column such that there is an upper pixel (x_1), a middle pixel (x_2) and a lower pixel (x_3).

On the training dataset we have 2 samples for each classes. It is with these examples that we are going to train our classifier.

$$\boldsymbol{x}^1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} ; \boldsymbol{x}^2 = \begin{pmatrix} 0 \\ .1 \\ .9 \end{pmatrix} ; y^1 = y^2 = 1$$

$$\boldsymbol{x}^3 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} ; \boldsymbol{x}^4 = \begin{pmatrix} .1 \\ .9 \\ .9 \end{pmatrix} ; y^3 = y^4 = 2$$

$$\boldsymbol{x}^5 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} ; \boldsymbol{x}^6 = \begin{pmatrix} .9 \\ .1 \\ .9 \end{pmatrix} ; y^5 = y^6 = 3$$

Where $\boldsymbol{x}^i$ refer to the input sample i and y^i refer to the class of sample i.

The **model** we are going to use is a neural-network. Because it has few layers, it's also referred as a shallow-neural-network. Our simple neural-network will have 3 inputs, one for each pixel of the images, 3 hidden neurons and 3 output neurons. Figure 1.1 on the following page is a graphical representation of our model.

Notation:
y is the class of a sample. This y is an integer representation of the class (1, 2, 3 ...). We'll often use the 'one-hot' notation of this value. For instance, if there is 3

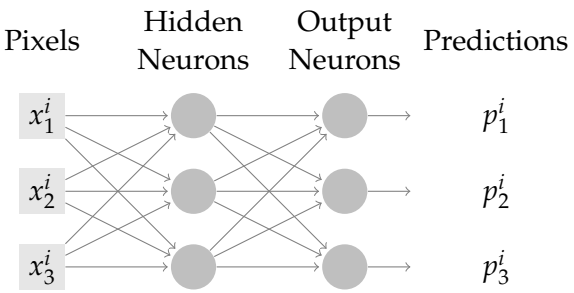

Figure 1.1: Simple example of shallow-neural-network

possible classes, the one-hot version of $y = 2$ will be $\boldsymbol{y} = [0, 1, 0]$. Where the second value of $\boldsymbol{y}$ is '1' and all the rest are '0'.

Chapter 2

Neural-Networks

In this section we will see how neural-network manage to solve classification tasks. As a reminder, a classification problem aims at identifying the sub-populations of a set belonging to a class. More specifically: if we are given a set of inputs X and outputs Y. where $\boldsymbol{x}^i$ denotes a sample from the set X and y^i its class. Then, the aim of classification is to predict the class a new sample given the knowledge based on the $\boldsymbol{x}^i$'s.

Artificial neural-networks are a family of statistical learning algorithms. They are inspired from the human brain structure: nowadays, we believe the brain has a network of neurons triggering signals over its dendrite and propagated to other neurons through a synapse interface. Network of artificial neurons mimics these properties.

In this section, we will first introduce what are the artificial neurons and how do we aggregate them to make neural-networks. After we've seen this, we will see how to train these networks: what is a cost and how do we reduce the error made by these networks.

It might be a good moment to mention that we wrote this section following the electronic book of Michael Nielsen. You'll find this book on-line at the following web page: `http://neuralnetworksanddeeplearning.com/`

2.1 Artificial neuron and Perceptron

First, the neurons. The first neuron we mention here is the **Perceptron**. It was described by Rosenblatt in 1958 [9] and was one of the first artificial neuron introduced in the literature. This neuron takes as an input multiple binary values to which it apply weights. Once multiplied, these *weighted inputs* are summed-up. If the sum is lower than zero, the neuron outputs zero. Otherwise, the neuron outputs one. A schema of an artificial neuron is given on Figure 2.1 on the next page.

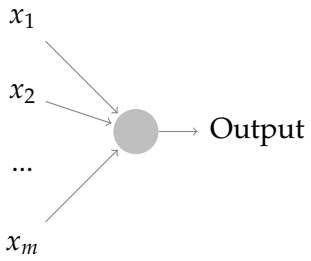

Figure 2.1: Model of an artificial neuron

Formally: given an input vector $\boldsymbol{x}$ of size m where x_i is the ith element of $\boldsymbol{x}$, the Perceptron defines a weight vector $\boldsymbol{w}$ with same shape as the input vector $\boldsymbol{x}$ and a bias term b. Then its transfer function is:

$$\text{Perceptron}(x) = \begin{cases} 1 & \text{if } \sum_{j=0}^{m}(w_j \times x_j) + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

In a vector notation this becomes

$$\text{Perceptron}(x) = \begin{cases} 1 & \text{if } \boldsymbol{w}^T \cdot \boldsymbol{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Nowadays we use other types of neurons. As the Perceptron, these neurons consider a weighted sum of their inputs plus a bias term. Now, their activation function is different and their outputs are real valued. We present two of the well known neurons:

- The **sigmoid neuron** is defined by a smooth threshold function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The **Regression Logistic Unit** (ReLU) is a neuron which activation function is equal to zero for any negative inputs and equal to its input otherwise.

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

2.2 Neural-network

Now that we have neurons, we need a network of them to compose an architecture similar to the brain. The networks we are going to work with are called **feed-forward** neural-networks. They are the most common ones in the literature. Figure 2.2 on the facing page is a representation of a two-hidden-layer feed-forward

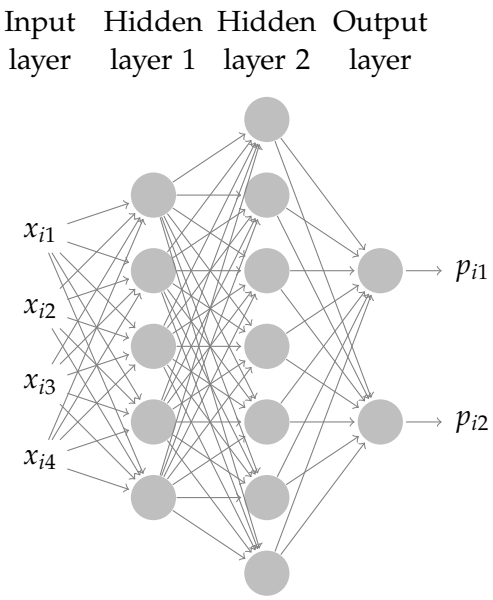

Figure 2.2: Feed-forward neural-network with two hidden layers

neural-network. As you can see, this model consist of groups of neurons. We denote as layer the group of neurons belonging to the same deepness in the model. Therefore, all the neurons visible on the left in our model, compose the first layer of neurons. It's the input layer. The following layers are called the hidden layers and the last one is the output layer. Some variations exists of this definition, for instance, some outputs of the model might be placed at the same level as some hidden layer.

Our simple example is also a feed-forward neural-network. It has an input layer fully connected to a hidden layer, which is also fully connected to the output layer. As drawn on the schema with the directions of the arrows, the signal propagates from left to right, from the inputs to the outputs. Because of these two reasons (layers fully connected and single direction signal propagation) our simple example is a feed-forward neural-network.

It's good to mention that other types of network exits such as the **recurrent networks**. In these networks, there is directed cycles on the graph which means that a neuron can depends on its own output. This model is considered to be closer to the brain structure but the challenge on training these models isn't state of the art. We won't work on these models.

Symmetrically connected networks is an other types of network, they are called the "Boltzmann machines". They are symmetrical in the sense that connections between neurons exists in the two directions and the weight on this connections is the same in both directions. Here again, we won't work on these models.

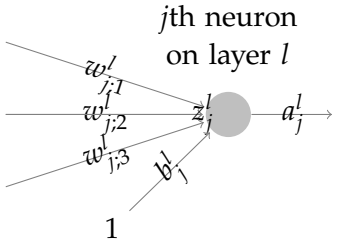

Figure 2.3: Our notation in a neuron

2.3 Forward-propagation

The logic route now is to relate our neural-network to the prediction or classification task. Lets consider we have a neural-network. This neural-network has weight vectors and biases initialized for each of its neurons. When we input a vector on the left of this neural-network each neurons on the next layer will be assigned a local output value (given the inputs on the previous layer). Following this same pattern from layer to layer, the input signal (vector $\boldsymbol{x}$) will propagate until the output layer resulting in our prediction.

Notation: To refer to the weights of the jth neuron of the lth layer we note $\boldsymbol{w}_j^l$. The superscript refer to the layer and the lower-script refer to the neuron number on this layer. Also, we note b_j^l the bias of the jth neuron of the lth layer. As a sum-up of the inputs, z_j^l is the weighted sum plus bias ($\boldsymbol{w}_j^{l^T} x + b_j^l$) of the jth neuron of the lth layer. Finally, the output of neuron j on layer l is denoted a_j^l. Figure 2.3 sums-up these notations. Instead of referring to each and every vectors, we often refer to the matrices composed by the vectors. For instance W^i is the weight matrix composed by each neuron's vectors of the lth layer.

Lets demonstrate this forward-propagation on our simple example. We will propagate $\boldsymbol{x}^1$, the first input sample of the dataset, and get the prediction made by the network. To do so, we need to initialize the weights and biases of the 6 neurons we have. We use weight initialization written on table2.1.

Given these weights, retrieving the prediction of the class of $\boldsymbol{x}^1$ consists on:

- Propagating $\boldsymbol{x}^1$ through the first neuron layer:

$$\sigma(\boldsymbol{z}^1) = \sigma(\boldsymbol{W}^{1^T}\boldsymbol{x}^1 + \boldsymbol{b}^1)$$

w_1^1	w_2^1	w_3^1	w_1^2	w_2^2	w_3^2
$\begin{pmatrix}-10\\-10\\20\end{pmatrix}$	$\begin{pmatrix}-10\\10\\10\end{pmatrix}$	$\begin{pmatrix}10\\-10\\10\end{pmatrix}$	$\begin{pmatrix}5\\-5\\-5\end{pmatrix}$	$\begin{pmatrix}-5\\5\\-5\end{pmatrix}$	$\begin{pmatrix}-5\\-5\\5\end{pmatrix}$
b_1^1	b_2^1	b_3^1	b_1^2	b_2^2	b_3^2
-13	-13	-13	0	0	0

Table 2.1: figure A.1 on page 51 shows the weights on a schema

$$\sigma\left(\begin{pmatrix}-10 & -10 & 20\\-10 & 10 & 10\\10 & -10 & 10\end{pmatrix}\cdot\begin{pmatrix}0\\0\\1\end{pmatrix}+\begin{pmatrix}-13\\-13\\-13\end{pmatrix}\right)=\sigma\left(\begin{pmatrix}7\\-3\\-3\end{pmatrix}\right)=\begin{pmatrix}.999\\.047\\.047\end{pmatrix}$$

- Propagating $\sigma(\boldsymbol{z}^1)$ through the second neuron layer (the prediction layer):

$$p(\boldsymbol{z}^1)=\sigma({\boldsymbol{W}^2}^T\sigma(\boldsymbol{z}^1)+\boldsymbol{b}^2)$$

$$\sigma\left(\begin{pmatrix}5 & -5 & -5\\-5 & 5 & -5\\-5 & -5 & 5\end{pmatrix}\cdot\begin{pmatrix}.999\\.047\\.047\end{pmatrix}+\begin{pmatrix}0\\0\\0\end{pmatrix}\right)=\sigma\left(\begin{pmatrix}9.042\\-9.991\\-9.991\end{pmatrix}\right)=\begin{pmatrix}.989\\.007\\.007\end{pmatrix}$$

We have here the prediction for $\boldsymbol{x}^1$. To have a percentage out of this prediction we normalize prediction and get that sample $\boldsymbol{x}^1$ could be of class 1 with $98,6\%$ accuracy, could be of class 2 with .7% or of class 3 with .7% accuracy. Our simple example, initialized with these weights made an accurate prediction for this sample.

2.4 Cost function

Once we have build our model, we consider a cost function, also called "loss function". This cost function define how good is the model doing, considering an evaluation criterion. In the case of classification, we want to evaluate how good the prediction is doing towards the true output. The most famous cost functions in neural-network classification are the squared error and the cross entropy (also called the "negative log-likelihood"). For a given sample i with prediction $\boldsymbol{p}^i$ and the true output $\boldsymbol{y}^i$, they are defined as follow:

- Squared error:

$$l(\boldsymbol{p}^i,\boldsymbol{y}^i)=\left\|\boldsymbol{y}^i-\boldsymbol{p}^i\right\|_2^2$$

- Cross entropy:

$$l(\widetilde{\boldsymbol{p}}^i,\boldsymbol{y}^i)=-\ln(\widetilde{\boldsymbol{p}}^i)^T\cdot\boldsymbol{y}^i$$

Where $\tilde{p}$ is the normalized value of the prediction over all the predictions. To evaluate the error of the entire dataset, we use the mean of all the errors. $l(\boldsymbol{P}, \boldsymbol{Y}) = \sum_i l(\boldsymbol{p}^i, \boldsymbol{y}^i)$

We are going to compute these two errors for the sample one but later on the report, we will only consider the cross entropy.

- Squared error:

$$l(\boldsymbol{p}^1, \boldsymbol{y}^1) = \left\| \begin{pmatrix} .989 \\ .007 \\ .007 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right\|_2^2 = .011^2 + .007^2 + .007^2 = .021$$

- Cross entropy:

$$l(\widetilde{\boldsymbol{p}^1}, \boldsymbol{y}^1) = -\ln \widetilde{\begin{pmatrix} .989 \\ .007 \\ .007 \end{pmatrix}}^T \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = -\ln \begin{pmatrix} .986 \\ .007 \\ .007 \end{pmatrix}^T \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = -\ln(.986) = .011$$

For the rest of the thesis, we'll be using the cross entropy error function. There is no special motivation under this choice. One mustn't believe that this choice is motivated by .011 being lower than .021. This result comes from the fact that there are two distinct costs. One compares a distance to the true output (the squared error) whereas cross-entropy gives a negative-log-likelihood.

2.5 Back-propagation

Until now we've seen how to build a model. We initialized our model with some specific values and propagated an input through this model. The forward-propagation of the signal gave us a prediction that could be compared to the true prediction of the sample thank to a cost function. Given an entire test set, we could use the same cost function to evaluate our model. In this subsection and the following, we investigate on how to modify the model such that it performs better.

2.5.1 Intuition

We want the model to perform better. We can't change the input samples nor their true predictions. The only parameters we can change are in the model. We could add neurons and see how new neurons improve the current model but we don't. What we are going to do is to modify the neurons such that they have a positive impact on the error. For each neurons present on the model, we are going to see

how a twist on their weights and biases impacts the error made by the model. To find the good twists we use Back-propagation and to take a step in the direction of this twist we use gradient descent (section 2.6).

The name of back-propagation comes as the opposite of forward-propagation. We start from the cost ($C = l(\boldsymbol{P}, \boldsymbol{Y})$) of the network and see how twisting the weights of the neurons, layer after layers, affects the cost. To be more formal, we search for the direction of the derivative of the cost as a function of the weights and biases of neuron j on layer i. To reach this step, we will first consider δ_i^l, the derivative of the cost with respect to the input of neuron j of layer l.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

You might want to watch figure 2.3 on page 16 to remind the notation. Also, for notation convenience we'll be using many vector/matrix notations. As a result, we'll use the Hadamard ("$\circ$") and the Kronecker ("$\otimes$") products. The definitions of these two products are reminded in appendix B.

2.5.2 Step-by-step

The **first** step of back-propagation is to back-propagate the cost C to the last layer L.

$$\begin{aligned} \delta_i^L &= \frac{\partial C}{\partial z_i^L} \\ &= \frac{\partial a_i^L}{\partial z_i^L} \cdot \frac{\partial C}{\partial a_i^L} \end{aligned} \tag{2.1}$$

$$\begin{aligned} \boldsymbol{\delta}^L &= \frac{\partial C}{\partial \boldsymbol{z}^L} \\ &= \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \circ \frac{\partial C}{\partial \boldsymbol{a}^L} \end{aligned} \tag{2.2}$$

On this last equation, we see that deriving the cost with respect to the neurons' inputs of the last layer L consist on deriving the cost C with respect to the prediction p and on deriving the output of a neuron with respect to it's inputs.

The **second** step is to back-propagate the cost to the other layers l. We will here take advantage of the chain rule.

$$\begin{aligned} \boldsymbol{\delta}^l &= \frac{\partial C}{\partial \boldsymbol{z}^l} \\ &= \left(\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \circ \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l}\right) \cdot \left(\frac{\partial \boldsymbol{a}^{l+1}}{\partial \boldsymbol{z}^{l+1}} \circ \frac{\partial \boldsymbol{z}^{l+2}}{\partial \boldsymbol{a}^{l+1}}\right) (...) \left(\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \circ \frac{\partial C}{\partial \boldsymbol{a}^L}\right) \\ &= \left(\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \circ \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l}\right) \cdot \left(\frac{\partial \boldsymbol{a}^{l+1}}{\partial \boldsymbol{z}^{l+1}} \circ \frac{\partial \boldsymbol{z}^{l+2}}{\partial \boldsymbol{a}^{l+1}}\right) (...) \left(\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \circ \frac{\partial C}{\partial \boldsymbol{a}^L}\right) \\ &= \left(\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \circ \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l}\right) \cdot \left(\frac{\partial \boldsymbol{a}^{l+1}}{\partial \boldsymbol{z}^{l+1}} \circ \frac{\partial \boldsymbol{z}^{l+2}}{\partial \boldsymbol{a}^{l+1}}\right) (...) \boldsymbol{\delta}^L \\ &= \left(\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \circ \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l}\right) \cdot \boldsymbol{\delta}^{l+1} \end{aligned} \tag{2.3}$$

At the very end of this equation we see how deriving the cost C with respect to the inputs of layer l depends on the derivative of the next layer, and therefore how back-propagation occurs.

The **third** part of the algorithm consist on deriving the cost C with respect to the weights $\boldsymbol{w}_j^l$ and biases b_j^l.

$$\begin{aligned}\frac{\partial C}{\partial \boldsymbol{w}_j^l} &= \frac{\partial z_j^l}{\partial \boldsymbol{w}_j^l}\frac{\partial C}{\partial z_j^l} \\ &= \frac{\partial z_j^l}{\partial \boldsymbol{w}_j^l}\delta_j^l\end{aligned} \tag{2.4}$$

With $\boldsymbol{z}^l = \boldsymbol{W}^{lT}\boldsymbol{a}^{l-1} + \boldsymbol{b}^l$

$$\begin{aligned}\frac{\partial C}{\partial \boldsymbol{W}^l} &= \frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} \otimes \frac{\partial C}{\partial \boldsymbol{z}^l} \\ &= \frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} \otimes \boldsymbol{\delta}^l \\ &= \boldsymbol{a}^{(l-1)T} \otimes \boldsymbol{\delta}^l\end{aligned} \tag{2.5}$$

$$\begin{aligned}\frac{\partial C}{\partial b_j^l} &= \frac{\partial z_j^l}{\partial b_j^l}\frac{\partial C}{\partial z_j^l} \\ &= \frac{\partial z_j^l}{\partial b_j^l}\delta_j^l\end{aligned} \tag{2.6}$$

With $\boldsymbol{z}^l = \boldsymbol{W}^{lT}\boldsymbol{a}^{l-1} + \boldsymbol{b}^l$

$$\begin{aligned}\frac{\partial C}{\partial \boldsymbol{b}^l} &= \frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} \otimes \frac{\partial C}{\partial \boldsymbol{z}^l} \\ &= \boldsymbol{\delta}^l\end{aligned} \tag{2.7}$$

2.5.3 Equations applied to our simple example

Before applying the back-propagation, we are going to compute some important derivatives. We will differentiate our cross entropy cost C with respect to its inputs and differentiate the sigmoid neurons with respect to their inputs too.

- The **derivative of the cost** C with respect to the predictions p_j:

$$\begin{aligned}\frac{\partial C}{\partial a_j^L} &= \frac{\partial C}{\partial p_j} \\ &= \frac{\partial \left(-\ln(\boldsymbol{p})^T \cdot \boldsymbol{y}\right)}{\partial p_j} \\ &= -\sum_k \left(\frac{y_k \partial \ln(p_k)}{\partial p_k}\frac{\partial p_k}{\partial p_j}\right) \\ &= -\sum_k \left(\frac{y_k}{p_k}\frac{\partial p_k}{\partial p_j}\right)\end{aligned} \tag{2.8}$$

- The **derivative of a neuron**'s output a_j^l with respect to its inputs z_j^l (or $\boldsymbol{a}^l$ wrt. $\boldsymbol{z}^l$).

$$\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial \sigma(z_j^l)}{\partial z_j^l} = z_j^l(1 - z_j^l) \tag{2.9}$$

$$\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = \frac{\partial \sigma(\boldsymbol{z}^l)}{\partial \boldsymbol{z}^l} = \boldsymbol{z}^l \circ (1 - \boldsymbol{z}^l) \tag{2.10}$$

- **Derivative of the softmax neurons**: In the cost description (section 2.4), we mention a normalized input $\widetilde{\boldsymbol{p}}$ but didn't emphasize on it. This normalized input impacts the last layer of the network such that the derivatives of this layer are different than simple sigmoid neurons. That kind of layer is called a softmax layer and is often used when neural-networks when dealing with multi-class tasks. You'll find more information on this function on the "Pattern recognition and machine learning" book [2] starting on page 113. Right bellow we derive the outputs of these softmax neurons with their aggregated inputs. There is two cases to derive: the first one is when we differentiate the neuron a_k^L that correspond to the true output $y_k = 1$. The second is when we derive the other neurons a_j^L that doesn't correspond to the true output $y_j = 0$.
  - If p_k corresponds to $y_k = 1$

$$\frac{\partial a_k^L}{\partial z_k^L} = \frac{\partial p_k}{\partial z_k^L} = p_k(1 - p_k) \tag{2.11}$$

  - If p_j corresponds to $y_j = 0$

$$\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial p_j}{\partial z_j^L} = -p_j p_k \tag{2.12}$$

Back-propagate:
We've seen the result of some derivatives of some functions present in our simple example. These derivatives will help us in applying the back-propagation algorithm. As mentions on the previous section, there is three main steps to take. **First** differentiate the cost C with respect to the inputs of the last layer. In our case we differentiate the cross entropy function with respect to the inputs of normalized-output-neurons. In other words we derive the cost function with respect to the inputs of the softmax neurons. The **second** step consist on back-propagating the error layer by layer such that we get the derivative of the cost with respect to the inputs of layer l. **Third**, and last step, is to compute the derivatives at the weights

and biases level instead of just the 'input' of the neurons as we've been doing so far.

1. To compute the **derivative of the cost** with respect to the inputs of the **softmax layer** we use the results of equations 2.8, 2.11 and 2.12.

$$
\begin{aligned}
\delta_j^L &= \frac{\partial C}{\partial z_j^L} \\
&= \frac{\partial C}{\partial p_j}\frac{\partial p_j}{\partial z_j^L} \\
&= -\left(\sum_k \frac{y_k}{p_k}\cdot\frac{\partial p_k}{\partial p_j}\frac{\partial p_j}{\partial z_j^L}\right) \\
&= -\left(\sum_k \frac{y_k}{p_k}\cdot\frac{\partial p_k}{\partial z_j^L}\right) \\
&= -\left(\sum_{k=j} \frac{y_k}{p_k}\cdot p_k(1-p_k)\right) - \left(\sum_{k!=j} \frac{y_k}{p_k}\cdot -(p_j p_k)\right) \\
&= -\left(y_j(1-p_j)\right) + \left(\sum_{k!=j} y_k(p_j)\right) \\
&= \left(y_j p_j - y_j\right) + \left(\sum_{k!=j} y_k p_j\right) \\
&= \left(\sum_k y_k p_j\right) - y_j \\
&= p_j - y_j
\end{aligned}
\tag{2.13}
$$

   Which in a vector notation is $\delta^L = \boldsymbol{p} - \boldsymbol{y}$

2. Then we **back-propagate** to other layers. We only have two layers, we've just done the last layer $l = 2$. We now do layer $l = 1$

$$
\begin{aligned}
\boldsymbol{\delta}^1 &= \frac{\partial \boldsymbol{a}^1}{\partial \boldsymbol{z}^1} \circ \left(\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1}\cdot \boldsymbol{\delta}^2\right) \\
&= \frac{\partial \sigma(\boldsymbol{z}^1)}{\partial \boldsymbol{z}^1} \circ \left(\frac{\partial (\boldsymbol{W}^{2^T}\boldsymbol{a}^1 + \boldsymbol{b}^1)}{\partial \boldsymbol{a}^1}\cdot \boldsymbol{\delta}^2\right) \\
&= \sigma(\boldsymbol{z}^1) \circ (1 - \sigma(\boldsymbol{z}^1)) \circ (\boldsymbol{W}^{2^T}\cdot \boldsymbol{\delta}^2) \\
&= \boldsymbol{a}^1 \circ (1 - \boldsymbol{a}^1) \circ (\boldsymbol{W}^{2^T}\cdot \boldsymbol{\delta}^2)
\end{aligned}
\tag{2.14}
$$

3. Finally, we compute the **derivative of the costs** with respect to the **weights**

and biases.

$$\frac{\partial C}{\partial W^2} = \frac{\partial z^2}{\partial W^2} \otimes \delta^2 = {a^1}^T \otimes (p - y) \quad (2.15)$$

$$\frac{\partial C}{\partial W^1} = \frac{\partial z^1}{\partial W^1} \otimes \delta^1 = x^T \otimes \delta^1 \quad (2.16)$$

$$\frac{\partial C}{\partial b^2} = \frac{\partial z^2}{\partial b^2} \otimes \delta^2 = (p - y) \quad (2.17)$$

$$\frac{\partial C}{\partial b^1} = \frac{\partial z^1}{\partial b^1} \otimes \delta^1 = \delta^1 \quad (2.18)$$

2.6 Gradient descent

With back-propagation, we've computed the derivatives of the cost with respect to the weights and biases of the model. In other words, we've seen how to twist the neurons' parameters such that, given an input sample, they lower the cost.

We now want to find the neurons' parameters that will lower the model's cost C. We'll search for this value with Gradient Descent. Gradient descent is an iterative algorithm aiming at finding the local minimums of a function. At each steps its improves its chances to be closer to a local minimum.

Consider a function $f : \mathbb{R}^n \to \mathbb{R}$ and its gradient ∇f. Gradient descent works by taking advantage of the gradient of the function f at a point x such that, following the negative value of the gradient, the algorithm can find a local minimum for the function.

$$x_{t+1} = x_t - \epsilon \nabla f(x)$$

On deep learning, we don't use gradient descent but **Stochastic Gradient Descent (SGD)**. This descent algorithm picks a sample at random on the dataset and evaluate the gradient from it. Still, "The derivative based on a randomly chosen single example is a random approximation to the true derivative based on all the training data" [3].

2.6.1 Example on our simple example

We want to perform a step towards a lower cost on our simple example using SGD. We'll pick 'at random' the 1st sample on the training set:

$$x^1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

That is of class $y^1 = 1$. The model we consider is initialized as before. It has the weight and biases of table 2.1 on page 17.

To perform a step toward a lower cost in the model, we use the prediction made by the current model on this sample $\boldsymbol{x}^1$ (already computed on section 2.3), and compute the back-propagation equation 2.15, 2.16, 2.17, 2.18 with the prediction $\boldsymbol{p}^1$ and output $\boldsymbol{y}^1$.

Using an epsilon $\epsilon = .5$, we get the following updates :

$$\boldsymbol{W}^1_{t+1} = \boldsymbol{W}^1_t - \epsilon \frac{\partial C}{\partial \boldsymbol{W}^1} = \begin{pmatrix} -10.0 & -10.0 & 10.0 \\ -10.0 & 10.0 & -10.0 \\ 20.0 & 10.0 & 10.0 \end{pmatrix} - \epsilon \begin{pmatrix} -0.0 & 0.0 & 0.0 \\ -0.0 & 0.0 & 0.0 \\ -0.0 & 0.003 & 0.003 \end{pmatrix}$$

$$\boldsymbol{W}^1_{t+1} = \begin{pmatrix} -10.0 & -10.0 & 10.0 \\ -10.0 & 10.0 & -10.0 \\ 20.0 & 9.998 & 9.998 \end{pmatrix}$$

$$\boldsymbol{W}^2_{t+1} = \boldsymbol{W}^2_t - \epsilon \frac{\partial C}{\partial \boldsymbol{W}^2} = \begin{pmatrix} 5.0 & -5.0 & -5.0 \\ -5.0 & 5.0 & -5.0 \\ -5.0 & -5.0 & 5.0 \end{pmatrix} - \epsilon \begin{pmatrix} -0.013 & 0.007 & 0.007 \\ -0.001 & 0.0 & 0.0 \\ -0.001 & 0.0 & 0.0 \end{pmatrix}$$

$$\boldsymbol{W}^2_{t+1} = \begin{pmatrix} 5.007 & -5.003 & -5.003 \\ -5.0 & 5.0 & -5.0 \\ -5.0 & -5.0 & 5.0 \end{pmatrix}$$

$$\boldsymbol{b}^1_{t+1} = \boldsymbol{b}^1_t - \epsilon \frac{\partial C}{\partial \boldsymbol{b}^1} = \begin{pmatrix} -13.0 \\ -13.0 \\ -13.0 \end{pmatrix} - \epsilon \begin{pmatrix} -0.0 \\ 0.003 \\ 0.003 \end{pmatrix}$$

$$\boldsymbol{b}^1_{t+1} = \begin{pmatrix} -13.0 \\ -13.002 \\ -13.002 \end{pmatrix}$$

$$\boldsymbol{b}^2_{t+1} = \boldsymbol{b}^2_t - \epsilon \frac{\partial C}{\partial \boldsymbol{b}^2} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix} - \epsilon \begin{pmatrix} -0.013 \\ 0.007 \\ 0.007 \end{pmatrix}$$

$$\boldsymbol{b}^2_{t+1} = \begin{pmatrix} 0.007 \\ -0.003 \\ -0.003 \end{pmatrix}$$

As a result, our model moved towards a better one (considering this sample). The loss of this model (toward x_1) before the updates was .0135 and now is .0134. Considering all the samples, the loss was before .014240 and decreased to 0.014233. This gradient descent algorithm helped us lowering the loss made by the model.

Chapter 3

Adversarial learning

This section is the main contribution of the thesis. We'll see how how an idea of adversarial learning can be applied on a shallow-neural-network. This Idea was first developed by Ian J. Goodfellow, Jonathon Shlens and Christian Szegedy. They proposed an adversarial training on a shallow-feed-forward-neural-network. Along the paper[4], they emphasize the be benefits of this method applied to a training on the MNIST dataset[**?**]. Thanks to their adversarial learning, they obtained a better accuracy on the test set.

3.1 Intuition

With adversarial learning, we train our classifier to resit to samples that could confuse it. On the paper mentioned above[4], the authors noticed that some adversarial modifications could be made up to fool drastically the classifier. If you were allowed to twist the pixels' values by .7%, and chose carefully these values, you could mislead your classifier from a 95% accuracy to a low .7%. The twist they proposed was adversarial in the sense that they elected a twist given the current classifier. In other words, it's by having full knowledge of the model's neurons and neurons' weights that they created the adversarial twists. These twists are computed with some math and will be detailed on section 3.2.

To take advantage of this observation, we will train our model, not on the casual dataset, but on an adversarial version of it. To be more concrete: instead of forward-propagating the input sample through the model and get the prediction out of it, we will twist the input sample and forward-propagate it. The cost will get is the one of the adversarial samples and not of the original dataset.

With this modification we hope that the classifier will learn, on the one hand, to recognize a class, like it would normally do, but on the other hand, will learn the differences between classes so that it can better discriminate them.

3.2 Creating the adversarial samples

The adversarial samples will be created from the original samples. We'll allow ourself to modify the pixels by some values. For instance, for an image, if pixels are coded with values between 0 and 255 (0 being black and 255 white) we will allow ourselves to modify the pixels values by $\epsilon_{\text{adv}} \times 255$ so that the image looks the same. The modifications will be based on our feed-forward models weights biases and cost function. Concretely, the adversarial modifications will be equal to:

$$\eta = \epsilon_{\text{adv}} \times \text{sign}(\nabla_x \text{Cost}(\boldsymbol{W}, \boldsymbol{x}, \boldsymbol{y}))$$

So that the adversarial samples $\boldsymbol{x}$ becomes:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} + \eta$$

To create an adversarial sample we compute the derivative of the model

3.3 Adversarial sample on our simple example

Lets now build an example of adversarial sample. As always, we'll use the simple example we've build up (visible on section A) and the sample $\boldsymbol{x}^1$. From those two elements, we'll build-up the adversarial sample corresponding to it.

We aim at deriving the cost of the model with respect to the inputs x. Luckily, we are able to re-use the back-propagation algorithm to compute the derivation. With the notation of section 2.5, we have:

$$\begin{aligned} \frac{\nabla_x C}{\partial \boldsymbol{x}} &= \frac{\partial \boldsymbol{z}^1}{\partial \boldsymbol{x}} \frac{\partial C}{\partial \boldsymbol{z}^1} \\ &= \frac{\partial \left({\boldsymbol{w}^1}^T \boldsymbol{x} + \boldsymbol{b}^1 \right)}{\partial \boldsymbol{x}} \cdot \boldsymbol{\delta}^1 \\ &= \boldsymbol{W}^1 \cdot \boldsymbol{\delta}^1 \end{aligned} \tag{3.1}$$

And the modified sample which is the sum of the sample plus a twist η is:

$$\begin{aligned} \tilde{\boldsymbol{x}} &= \boldsymbol{x} + \eta \\ &= \boldsymbol{x} + \epsilon_{\text{adv}} \times \text{sign}(\nabla_x C) \\ &= \boldsymbol{x} + \epsilon_{\text{adv}} \times \text{sign}(\boldsymbol{W}^1 \cdot \boldsymbol{\delta}^1) \end{aligned} \tag{3.2}$$

At this point, we know how to compute the compute an adversarial sample. From the last equation, it doesn't clearly appears that an adversarial sample depends on the variables of the model. Taking a closer look into it, we have that an adversarial sample depends on all the variables present on the model. The neurons'

weights appears twice. First on the the back-propagation and then on the forward-propagation (the cost) and the biases appears once on the forward-propagation. We also have that the sample class is present on the cost. Therefore, the adversarial sample needs an entire knowledge on the model and on the samples.

Lets apply this to sample $\boldsymbol{x}^1$ with $\epsilon_{\text{adv}} = .07$:

$$\begin{aligned}
\widetilde{\boldsymbol{x}^1} &= \boldsymbol{x}^1 + \epsilon_{\text{adv}} \times \text{sign}(\boldsymbol{W}^1 \cdot \boldsymbol{\delta}^1) \\
&= \boldsymbol{x}^1 + .07 \times \text{sign}\left(\begin{pmatrix} -10 & -10 & 10 \\ -10 & 10 & -10 \\ 20 & 10 & 10 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0.003 \\ 0.003 \end{pmatrix}\right) \\
&= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0.07 \\ 0.07 \\ 0.07 \end{pmatrix} \\
&= \begin{pmatrix} 0.07 \\ 0.07 \\ 1.07 \end{pmatrix}
\end{aligned} \tag{3.3}$$

From this example we can see the adversarial twist added noise in a direction to confuse the model. This augmented the error made by the model: it went from a loss of .013501 to a loss of .013580 on the adversarial sample. With this example, the benefits of adversarial learning are not clearly visible but we hope that the process underlining adversarial learning is understood.

3.4 MNIST

The first dataset we're going to evaluate is the MNIST dataset[**?**]. This dataset is composed by 70k images representing digits going from 0 to 9. Each of the images has $28 * 28$ gray scaled pixels. For each pixels is given a value in between 0 and 1 stating how bright is the pixel. The task, given this dataset, is to classify the samples into the 10 classes they belong in. For our needs, we discomposed the dataset into 3 sets: a training-set composed by 50k samples, a validation-set composed by 10k samples and a testing-set composed by 10k samples.

From our observations, this dataset is similar to a baseline in machine learning for testing algorithms in image processing. The reason is that it has been around for a long time and that it's big and small enough to quickly and accurately test new ideas related to this domain. It's for this reason that we'll make most of our tests using this dataset.

Lets now get to adversarial learning. On their paper[4], the authors used adversarial-learning in conjunction with other techniques. In order separate adversarial-

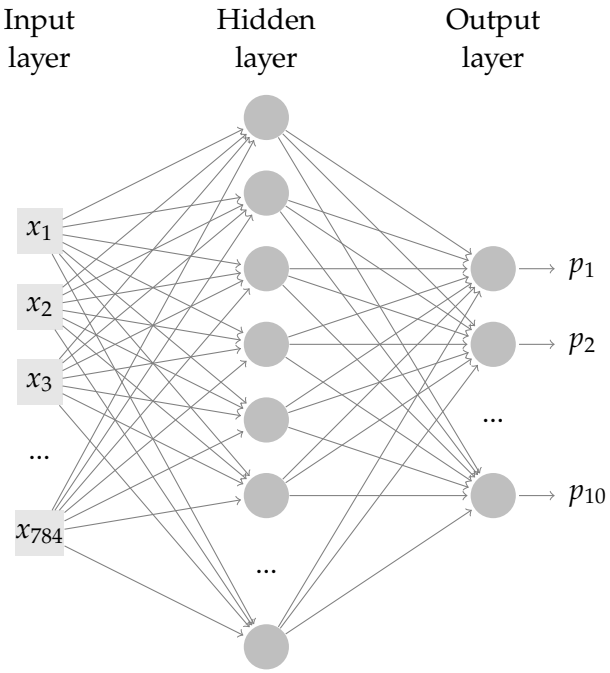

Figure 3.1: Graphical model of a MNIST net

learning from any anther technique, we've decided to re-implement the shallow-neural-network without any method that could be assimilated as a regularization. Concretely, the authors of the paper used two methods that function together called maxout and dropout. Dropout is considered to regularize the network. For this reason we removed it such that our net is only composed by the pixels inputs, the hidden ReLU layer and the softmax output layer. Figure 3.1 is a graphical representation of the MNIST net we used.

On the following subsections, we are performing some experiments to confirm the benefits of adversarial learning. We'll see the impact of the amount of neurons on the networks, the impact of modifying the adversarial epsilon and compare the adversarial-model with other ones.

3.4.1 Playing with neuron quantities

One of the first experiment we got interested in knowing which parameters should be used to train the neural network. We would expect that the size of the hidden layer has a direct impact on the predictions and on the accuracy of the model. We would therefore try different layer sizes and see which ones were over and under fitting. We also want to know if one of the adversarial or usual training would under or over-fit before the other given the same amount of neurons. If one were to under-fit longer than the other, it would imply it need to learn more functions

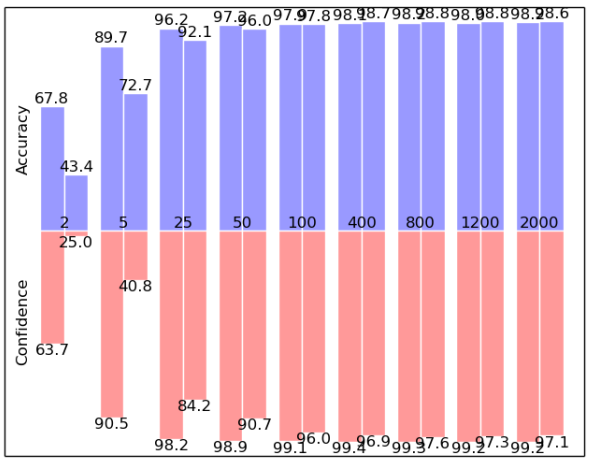

Figure 3.2: Comparing the amount of neurons' impact on non-adversarial and adversarial models. Every pair represent a non-adversarial-learning and an adversarial-learning trained with a certain amount of hidden neurons.

than the other model to be accurate.

In other words, if both of the models stops under-fitting on the same time, it would probably mean that both of the models learn the same amounts of functions, but one would be more accurate than the other. On the other hand, if a model (A) under-fits longer than the other model (B) it would probably mean that it (A) needs to learn more functions to accurate than the other model (B).

For this reason, the we've trained MNIST on models with different amount of weights. Each of the adversarial examples are trained with $\epsilon_{\text{adv}} = .25$. The graph on figure 3.2 shows the evolution of the accuracy and the confidence for the adversarial and non-adversarial models while increasing the amount of neurons. As one could expect, the accuracy is defined as :

$$\text{acc} = \frac{\#\text{correct guesses}}{\#\text{samples}}$$

And the confidence reflects how much the model was sure of which ever predictions it made

$$\text{conf} = \text{mean}(\text{argmax}(\text{predictions}))$$

Analysis:

Looking at this graph (figure 3.2), the first thing we notice is that adversarial-learning performs worse than non-adversarial-learning for models with less than

a hundred neurons. When we deal with 5 hidden neurons, adversarial model is 17 percent points less accurate than the non-adversarial model. With this observation we can hypothesize that adversarial model can't properly synthesize what it sees with so few functions (neurons). Therefore it introduces more inaccuracy in the functions it learns.

What we find more relevant is the following: the adversarial-learning plateaus at 400 neurons whereas non-adversarial-learning plateau in between 50 and 100 neurons. In other words, adversarial-learning need far more neurons (far more functions) to reach its best accuracy. From this observation, and from the insight of the universal approximator [6], we can say that adversarial-learning learns more functions than the other model does. Which is to say that the adversarial-learning method is not about having the same kind of model but one being more accurate than the other. It's about having a new model learning far more functions. Also, as the adversarial-learning models converges slower than the other model, I would say that adversarial-learning learns a whole set of new functions. If it were not the case, We would expect the adversarial models to be at least as good as the non-adversarial.

3.4.2 Playing with adversarial epsilon

The Adversarial learning we use relies on an epsilon, the one we've noted ϵ_{adv}:

$$\boldsymbol{x} + \epsilon_{\text{adv}} \times \text{sign}(\nabla_{\boldsymbol{x}} C)$$

In this subsection, we want to evaluate the impact of this value on the learning. We are going to expose the learning of a 800 hidden-neurons net trained with different ϵ_{adv}. Even though, we expect an increased accuracy with an increased epsilon, we don't have any idea on which epsilon would be better.

Analysis:
Figure 3.3 on the next page seems to reveal that a good adversarial epsilon for this dataset and given a model of 800 neurons is located between .05 and .3. On the figure we plotted, the optimum value for the adversarial epsilon seems to be .1. On this same plot, we can see that the confidence for the predictions falls down as the adversarial epsilon raises. We don't know what these two informations teaches us towards adversarial-learning but that low epsilon value seems to perform better on both accuracy and confidence.

3.4.3 Compare to noisy dataset

One of the objective mentioned for adversarial-learning was to diminish the linearity of the network: The authors of [4] said that usual networks were not able to

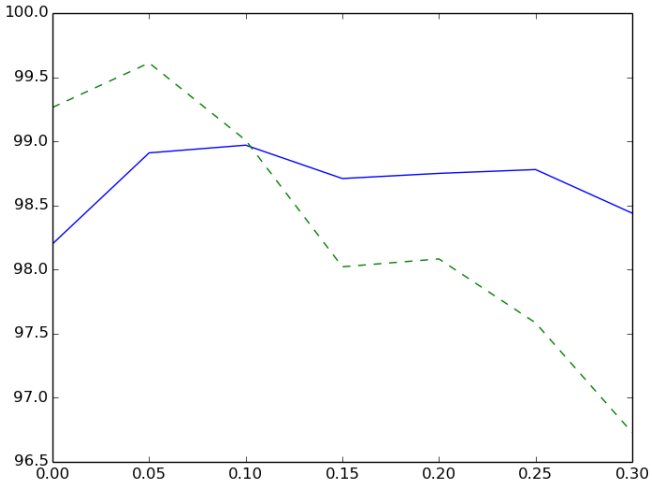

Figure 3.3: Evaluate non-adversarial and adversarial models of 800 hidden neurons.

understand the underlining nature of a class because it was too much dependent on the underlining distribution of the data present on the dataset. This dependency was described with too much linearity on the network to learn the underlining dataset. It's because of this over-linearity that adversarial samples were so good at fooling the models.

On this subsection, we want to see if, effectively, the model is better at generalizing. To do this, we are going to compare a non-adversarial model (A) with an adversarial model (B) on modified test-sets. The first one will compare A and B on the adversarial test-set and will most probably confirm the hypothesis seen above.The second one is going to compare A and B on an other modified version of the test-set. Here we'll by twisting each pixels by a random value. This random value comes from a standard normal distribution (with a mean of 0 and variance of 1). To produce this set we compute :

$$\tilde{x}^i = x^i + \epsilon_{\text{ads}}\text{noise}$$

Where noise is randomly generated following a standard normal distribution.

Analysis:
With the left plot of figure 3.4 on the following page, we confirm that adversarial samples easily fools a usual feed-forward model. Twisting the pixels by 10% in the direction of the worst case (our adversarial case),drastically confuses the original model. Now, trying to fool the adversarial-model with this same technique doesn't work as good. The adversarial-model can't be fooled as easily.

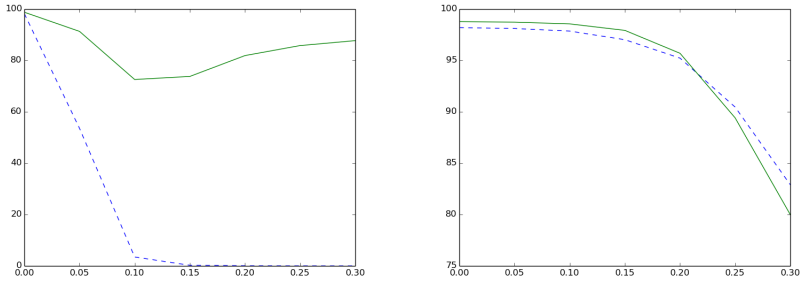

Figure 3.4: Evaluate non-adversarial (dashed line) and adversarial models (plain line) of 800 hidden neurons on modified test-sets. On the left, we try the two models on an adversarial test-dataset. The horizontal axis represents the epsilon values used to modified the test-set. On the right, we try the two models on an normal-distribution-modified test-dataset with different standard deviations.

Figure 3.5: Extract of the test-set with the same image with different noises. On the top, you see the adversarial noised sample with $\epsilon_{\text{ads}} = [.0, .1, .2, .3]$. On the bottom you see the uniform noised sample with, from left to right the standard deviation taking values: $\epsilon_{\text{noise}} = [.0, .1, .2, .3]$

With the plot on the right, we compare the accuracy of both the models on another twisted dataset. This time, the adversarial-model keeps an advantage until the dataset becomes too noisy $\epsilon_{\text{noise}} = .22$. This result is a bit contradictory with our expectations. In fact, we would have expected that adding noise to this test-set would have fooled more the non-adversarial model than the adversarial one. The reason being that adversarial-learning was supposed to better generalize the underlining distribution of the test-set and therefore be less dependent on noise.

Despite this, the adversarial-model still outperforms the other one on this modified test-set before the test-set gets really noisy ($\epsilon_{\text{noise}} > .22$)

3.4.4 Train with noisy dataset

So far we've been comparing an usual model to the non-adversarial model. Here, we introduce a new model originating from the previous experience. As a matter of fact, we stated that model A (the usual model) was too much linear. We introduced model B (adversarial-model) as being less dependent on the underlining distribution. Then we tested the models on a noisy test-sets. Now we want to compare B with a new model that is also less dependent on the underlining distribution. Therefore we introduce a model C (also having 800 hidden neurons) that is trained on noisy dataset. This dataset is also computed live. At every iterations of the back-propagation a new test-set is generated (as for the adversarial-model (B)). This live modification of the test set permits the classifier not to over-fit an original noise given to the network. It must learn that noise can appear. The input sample modification is the same a before:

$$\tilde{\boldsymbol{x}}^i = \boldsymbol{x}^i + \epsilon_{\text{ads}}\text{noise}$$

Where noise is randomly generated following a standard normal distribution.

As a recall, adversarial-learning modified the input samples towards a worst case prediction given the model. Now, instead of modifying the input sample towards this worst case sample, we twist the sample by adding a random noise. At this point, we don't necessary expect this learning to be better at classifying but we hope that it can better resist to adversarial samples. If it does, it could support that usual training is too much linear.

Analysis:
With the left plot of figure 3.6 on the next page we can make two observations. The first one was unexpected and is that model C (new one) performs as good as model B (adversarial model) on the original dataset. They both score close to 98.7% accuracy on the original test-set. The second observation, more interesting to us, reveals that model C is more resistant to adversarial samples than model A.

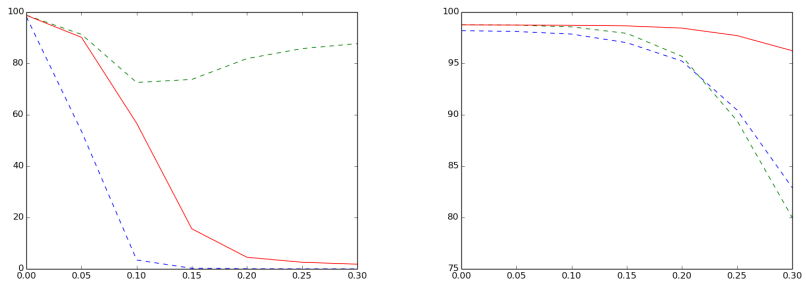

Figure 3.6: Evaluate normal-distribution-based model versus the casual and adversarial models of 800 hidden neurons on noisy test-sets and adversarial datasets.

This observation come in favor of A being too linear and relying too much on the original distribution of the samples. Why so ? The objective in training the dataset with model C, was to enforce the classifier to learn that each of the pixels of a class followed a normal distribution. We enforce the model to consider an alternative data distribution than the one present on the original training-set. If C would have performed as bad as A on the adversarial test-set, it could have meant that model A, actually, generalizes and is not as linear as we ought to say. With the plot on the right, we have that model C outperforms models A and B. This observation is legitimate with the experiment we've build up. We trained model C on believing that the pixels were under a normal distribution and then tested the model on a test-set enforcing this normal distribution modification. Once again, we are a bit disappointed here that adversarial-learning can't best predict this noisy dataset.

3.4.5 Visualizing weights

We thought it could be nice to visualize what had been learned by the network. We've therefore tried to visualize the weights to find out the patterns it relies on. The **first** idea that came to our mind was to "back-propagate" the output-neurons' weights connected to the inputs. As a result, we would have visualized a pattern of what was understood as a 1, a 2 or any class of the task. Mathematically we just needed to compute:

$$W_{\text{visualize}} = \boldsymbol{W^1} \cdot \boldsymbol{W}^2$$

And each of the columns of $W_{\text{visualize}}$ would have represented the patterns learned from a class. On figure 3.7 on the facing page, you see these weights for classes 0 to 4 of MNIST. The upper-row are casual-learning and the lower-row is adversarial-learning. On both neither of these images we can distinguish the patterns of a one, a two or any other digit.

The **second** way we tried to visualize the patterns was at the first (and only) hidden-layer we have. We've plotted some of the neuron's weight but none gave

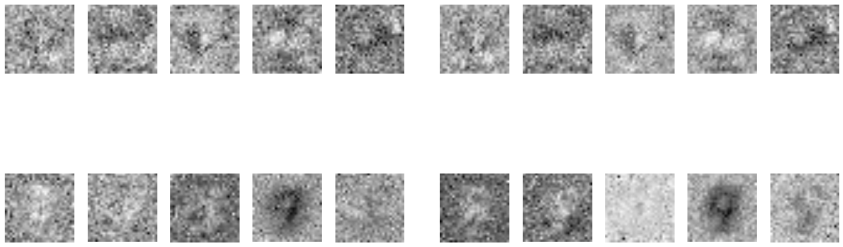

Figure 3.7: Attempt at visualizing classes pattern on casual-model (top) and adversarial-model (bottom) of 800 neurons (left) and 2000 neurons (right).

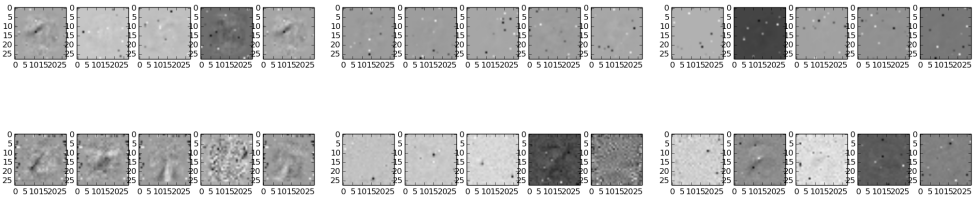

Figure 3.8: Attempt at visualizing neuron patterns on casual-model (top) and adversarial-model (bottom) of 100 neurons (left), 800 neurons (center) and 2000 neurons (right).

gave us more insight on what was seeing model A and model B. After watching figure 3.8, the only comparison we could make was that the adversarial-model seems to have more grain on its neurons compared to the casual-model.

After these two failures in visualizing the patterns we expected to have, we've browsed the find and found that this types of networks were not supposed to contain some neurons with the patterns we were looking for. Instead, another model called Restricted Boltzmann Machines (RBM) would have these properties (where you can see patterns learned by neurons).

3.4.6 MNIST summary

With the MNIST dataset we've seen many things.

- Playing with the hidden neurons quantities we've seen that adversarial-model (B) learned more functions than the usual-model (A) did.
- Playing with the adversarial epsilon we've seen which epsilons seemed to perform better.
- Comparing models A and B with noisy datasets confirmed that A was sensible to adversarial samples when B was not.
- Training a new model (C) on a noisy dataset probably confirmed that A was too linear.

Figure 3.9: Extract of Cifar10 dataset.

- Visualizing weights didn't worked as expected but played a role in knowing what future work could be made of.

3.5 CIFAR-10

Another dataset we've used is the CIFAR-10 database. CIFAR-10 is a dataset composed by 60k images. An image is about one of the following 10 classes: *an airplane, an automobile, a bird, a cat, a deer, a dog, a frog, a horse, a ship or a truck*. Each images is composed by $32 * 32$ RGB pixels which makes an input vector composed by 3072 values. As for the MNIST dataset, we discomposed the dataset into 3 sets: a training-set composed by 40k samples, a validation-set composed by 10k samples and a testing-set composed by 10k samples. Figure 3.9 is an extract of the dataset.

Right below, we are going to see how adversarial-learning did with CIFAR-10.

3.5.1 Training CIFAR-10

We've build a model to learn CIFAR-10. At the beginning, we trained a model with 3072 inputs, 2500 hidden neurons and 10 outputs. Training this dataset took a huge amount of time (we didn't reached the end). Therefore, we decided to shrink the input vector. To do so, we gray-scaled the input sample. In other words, we changed the RGB pixel to gray pixels. This modification shrank our input by 3 times less inputs values (1024 instead of 3072). Even shrinking the input vector, learning the dataset took 3 days on our computer. Because of this reason, we were not able to perform all the test we performed on MNIST.

As we did for MNIST, we are going to compare basic model (Ac) and adversarial-model (Bc) to noisy datasets. As before, we'll have an adversarial dataset and a dataset with random noise. Before rnedering the graph, we expect to have the same kind of results. We expect the adversarial samples to fool Ac but not Bc. We expect Bc to be more accurate that Ac is and, maybe we'll have this time Bc performing better than Ac on the random noise test-set. Figure 3.10 on the next page are the resulting plots applied for CIFAR-10.

Analysis:

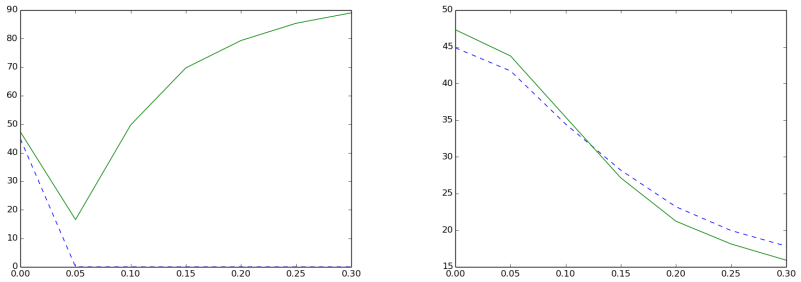

Figure 3.10: Evaluate non-adversarial (dashed line) and adversarial models (plain line) of 2500 hidden neurons on modified test-sets. On the left, we try the two models on an adversarial test-dataset. The horizontal axis represents the epsilon values used to modified the test-set. On the right, we try the two models on an normal-distribution-modified test-dataset with different standard deviations.

Looking at the results, we have that Bc outperforms Ac on the original test-set. This observation comes in favor of adversarial-learning being less linear that Ac is. This result is very similar to the one we had with MNIST on section 3.4.3. As before, we see that Ac is easily fooled by adversarial learning and that Bc is not. It appear weird that Bc does an amazing job at classifying the adversarial-test-set, but remember that the noise given to the adversarial-test-set depends on the output. Therefore, building this adversarial-test-set, we gave informations about which classes the samples belonged in. As before, also, we have that Bc is not doing an amazing job on the noisy test-set. The adversarial-classifier can't predict a sample with random noise much better than Ac does.

3.5.2 CIFAR summary

With this new dataset, we confirm some of the observations we've made earlier with the MNIST dataset. Sadly, the higher dimensionality needed on the models made us unable to recompute all the tests performed earlier.

3.6 Covertype dataset

This is the third and last dataset we performed experiments on. This one is called the covertype dataset[1]. We elected this dataset because its nature was far from the two image dataset seen previously and because it consists on a multi-class classification task. With this dataset we "classify the predominant kind of tree cover from strictly cartographic variables". The samples belongs in 7 classes and the input vectors are composed by 54 values. Here the 54 values are not necessarily in a given range. It might happen that a value is a numeric encoding.

[1] `https://archive.ics.uci.edu/ml/datasets/Covertype`

3.6.1 Training Covertype

Learning this dataset was not as successful as learning the two other datasets. We've trained many models hopping to see an adversarial one better than its non-adversarial equivalence but it never really happened. the results we got are the followings:

hidden neurons	60	80	100	120	150
non-adversarial	.5103	.5107	.5116	.5108	.5108
adversarial	.5106	.5111	.5103	.5106	.5116

Table 3.1: Accuracy of adversarial and non-adversarial models with different amount of hidden neurons on the Covertype dataset.

3.6.2 Covertype summary

From this experiment we learn that adversarial-learning won't necessarily perform better non-adversarial-learning on every dataset. The fact that adversarial-learning doesn't perform better here is relevant with the nature of the dataset.

3.7 Adversarial learning conclusion

Here ends our experiments on the different datasets. we'll come back on the results obtained here in the conclusion. We are now going to see how we implemented these experiments.

Chapter 4

Implementation

In this section we present the software we chose to implement our solution. The elected software comes from the Lisa-lab[1] in Montreal and is called Pylearn2 [5]. The Lisa lab is specialized in neural-network and more specifically, deep learning. Yan Goodfellow, one of the writers of the paper[4], was part of this institution, he is one of the main contributor of Pylearn2 and write most of his paper with this tool. His implementation of Adversarial-learning was made with this tool but isn't detailed. In the following section, we are going to present the software's needs we had with respect to this thesis and on the next one we'll explore the way Pylearn2 works.

4.1 software needs

At the very beginning of this thesis we searched for a good software with respect to many needs. Of course, as we are doing neural-nets, our first need was to have a software able to implement a neural-network. Then there was some other needs. The learning curve for this software had to be steep enough so that we could implement our solution in some reasonable time. After this, the software had to be flexible towards our needs of adversarial-learning. Indeed, the cost of our shallow-feed-forward network had to be modified in order to accept adversarial-learning. Also, as we have large databases like the Cifar-10 (composed by $60k$ sample of $3k$ values each), the software's algorithms had to perform fast operations. To show the importance of this criterion, with the software we finally choose, it took us more that 48 hours to compute the adversarial model of Cifar-10.

In neural networks there was 4 main softwares answering partially our needs.

- Torch: "Torch is a scientific computing framework with wide support for machine learning algorithms". Their framework is very easy to use. In one hour

[1] urlhttps://github.com/lisa-lab

you can build a neural-network classifying MNIST. Now, the implementation of new functions is complicated. Implementing a new cost function is hard, it's for this reason that we didn't chose this soft.

- Lush: "Lush is an programming language designed for researchers, experimenters, and engineers interested in large-scale numerical and graphic applications". Even though the description seems appealing, the software isn't maintained and doesn't embed modern gradient descent algorithms.
- Pylearn2: "Pylearn2 is a machine learning library". It was designed for deep-learning, is written in python and can differentiate function. Out of the other alternatives, it's the hardest tool to use. Still, because it also fulfills all the software needs, this alternative was chosen
- Matlab: The very famous Matlab could have been used for our project. Now the libraries provided weren't as specific as the other softwares.
- R: "R is a language and environment for statistical computing and graphics". This language integrates deep-learning algorithms but was not mentioned as a reference in the deep leaning domain.

Advantage and drawbacks of Pylearn2:
On the one hand, Pylearn2 is written in **Python**: a very common and easy to understand language. That makes a big advantage to Pylearn2 toward a soft written in C or Lua. On the other hand, Pylearn2 uses a file format called "**Yaml**" to build and train the networks. These files are made so that you can easily call classes with many parameters. Though it's practical, it's hard to understand this approach if you've never seen something similar before. Pylearn2 is being **under development** which makes it a software up to date. Now, this advantage is also a drawback as the python documentation isn't as furnished as one could expect nor are the error messages. It was often needed during the implementation to browse specific forums and investigate on which could be a good solution for us. Thought the python documentation lacks of details, a blog is maintained by the Lisa-lab to introduce to their tools.

At this point, Pylearn2 is far from being the easiest tool to use in order to build and trained a neural-network, but what really came in the balance to support it use is a useful library on which it relies: **Theano** [1]. Theano, also developed in the Lisa lab. It's "a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently". This library also include **automatic differentiation**, so that you can create a new cost function without having to differentiate it by hand.

I'll finish these motivations mentioning that Pylearn2 is hard to get into. One need to spend some time on the code, on the blog and on the forums to builds its own models but, at the end, Pylearn2 is very flexible and offer a useful tool of differentiation to rapidly test new cost functions.

4.2 Pylearn2

Lets now present the tool. Pylearn2 is a python library. It mainly relies on Numpy, for matrix manipulation and on Theano, for optimized operations on matrices. There is three main files on Pylearn2: the model, the cost and the routine files. Following comes the description of them. You may watch before figure 4.1 on the following page for a graphical representation of the system.

- **The model**: The model is represented by a class in Pylearn2. It has to follow rhe sucture of an abstract class (*pylearn2.models.model.Model*) in order to work with the system. In supervised learning (what we are doing), this model will be called by the cost to compute it. The model, in our case, is a shallow-feed-forward-neural-network. An flexible implementation of this model is already available on the Pylearn2 library.

- **The cost**: In Pylearn2 philosophy, one should be able to modify easily the cost to test it. This is the strength of Pylearn2, where you would give the cost and the library will differentiate it. This ease implies some difficulties as the user has to input a cost in a 'Theano-symbolic' format. In the system, this cost is also represented by an abstract class (*pylearn2.costs.cost.Cost*) where the user must implement the 'expr' function which outputs the 'Theano-symbolic' function.

- **The routine**: Finally, when one desire to build a model with a specific cost and a specific learning algorithm, one will create a 'yaml' file. This file is only composed by classes calls. For the training, the library expects first the *Train* class containing both the *Model* one and the *Learning algorithm* class which itself contains the *Cost* class. With these classes and the dataset given as parameters of them, Pylearn2 will initiate a learning process.

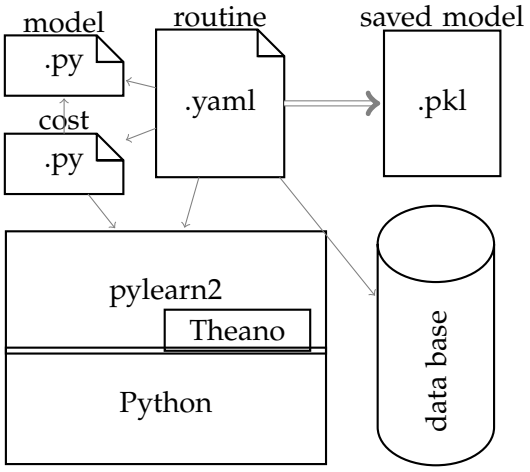

Figure 4.1: Graphical representation of Pylearn2's routine

4.3 Our implementation

We decided to re-implement the adversarial-learning for two reasons. The first one was to get more insights on the way Pylearn2 works and the second reason was that there were no real available code doing adversarial-learning. When we found Ian's work, it was to difficult for any Pylearn2 beginner. At the end, the code produced is very easy to understand. None the less, this bit of code doesn't reflect the difficulties of writing it, as there is a huge lack of documentation and proper error logs on Pylearn2.

We won't present here the **model** for two reasons. At first we used an implementation that is provided in the library, and, secondly, because the code is hundreds of lines. How-ever, the code beneath the model does what we expected it to do. It creates a layer of Regression Logistic Units and a softmax layer.

Then comes the implementation of the **cost**. On the constructor we define a learning epsilon value that will be able to modify when calling the routine. Then, the 'expr' function computes the adversarial-cost in a Theano-symbolic way.

```
class AdversarialCost(DefaultDataSpecsMixin, Cost):
    # The default Cost to use with an MLP.
    supervised = True

    def __init__(self, learning_eps):
        self.learning_eps = learning_eps


    def expr(self, model, data, **kwargs):
        # check input space
```

```
        space, sources = self.get_data_specs(model)
        space.validate(data)

        # Compute adversarial cost
        X, y = data
        alpha = .5
        adv_X = X + self.learning_eps
                * T.sgn(T.grad(model.cost_from_X(data),X))
        adv_data = (adv_X, y)

        return alpha*model.cost_from_X(data)
                + (1-alpha)*model.cost_from_X(adv_data)

    @wraps(Cost.is_stochastic)
    def is_stochastic(self):
        return False
.
```

Finally, the **routine** file (the .yaml one) covers all the function calls. At first you call Pylearn2's *Training* class. Then you define the model you want to train, here a ReLU shallow-neural-net. On the example below, the network is composed by 784 input units (for the MNIST database), 1200 ReLU hidden units on a single layer and a softmax output layer composed by 10 classes. Here, each of the classes will refer to a digit class. After the model comes the *learning algorithm* class. We elected the Standard Gradient Descent (SDG) with some improvements: we use a batch version of it so the results comes quicker and add a momentum to the learning rate so that learning takes into consideration its previous moves when following the gradient. Still in the SDG, we elected common early-stopping stopping technique based on a validation set: we train on a testing set and, when ever the validation set show that training is over-fitting we stop the training. Furthermore, we use a rollback so that we don't consider over-fitting to early, in other words, the learning wont stop as soon as the validation states it performs worst than the train set, but we wait a hundred iteration before claiming the over-fitting (still based on the train set outperforming the validation set = over-fitting). Finally comes the cost related to the learning algorithm. Here we use the one we've shown previously: the adversarial cost. In this class comes a learning epsilon, it is the one of equation 3.2. When given 0, it's an usual cross-entropy learning and when given a value between $]0, 1[$ it's an adversarial learning.

```
!obj:pylearn2.train.Train {
  dataset: &train !obj:pylearn2.datasets.
        dense_design_matrix.DenseDesignMatrix {
    X: !pkl: '/home/marc/data/mnist_train_y_train_X.pkl',
    y: !pkl: '/home/marc/data/mnist_train_y.pkl',
    y_labels: 10,
```

```
},
model: !obj:pylearn2.models.mlp.MLP {
  layers:[!obj:pylearn2.models.mlp.RectifiedLinear{
       layer_name: 'h0',
       dim: 1200,
       sparse_init: 15
     }, !obj:pylearn2.models.mlp.Softmax {
       layer_name: 'y',
       n_classes: 10,
       irange: 0.
     }
  ],
  nvis: 784,
},
algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
  batch_size: 200,
  learning_rate: .01,
  learning_rule: !obj:pylearn2.training_algorithms
      .learning_rule.Momentum {
    init_momentum: .5 },
  monitoring_dataset: {
    'train' : *train,
    'valid' : !obj:pylearn2.(...).DenseDesignMatrix {
      X: !pkl: '/home/marc/data/mnist_valid_X.pkl',
      y: !pkl: '/home/marc/data/mnist_valid_y.pkl',
      y_labels: 10,
    },
    'test' : !obj:pylearn2.(...).DenseDesignMatrix {
      X: !pkl: '/home/marc/data/mnist_test_X.pkl',
      y: !pkl: '/home/marc/data/mnist_test_y.pkl',
      y_labels: 10,
    }, },
  termination_criterion:!obj:pylearn2.
      termination_criteria.And { criteria: [
      !obj:pylearn2.termination_criteria.MonitorBased {
          channel_name: "valid_y_misclass",
          prop_decrease: 0.,
          N: 100
      },
      !obj:pylearn2.termination_criteria.EpochCounter {
          max_epochs: 1500
      }
    ]
  },
  cost: !obj:costAdv.AdversarialCost {
    learning_eps: %(learning_eps)f }
},
```

```
  extensions: [
    !obj:pylearn2.(...).MonitorBasedSaveBest {
        channel_name: 'valid_y_misclass',
        save_path: "%(path)s/mlp_1_%(learning_eps)s.pkl",
    }, !obj:pylearn2.(...).MomentumAdjustor {
      start: 1,
      saturate: 10,
      final_momentum: .99
    }
  ]
}
```

As a note, one may wonder why is the cost inside the learning algorithm. The reason is that some neural-networks can be unsupervised, meaning that their cost doesn't depends on their inputs. As a result, the learning algorithm just doesn't ask for a cost.

Chapter 5

Conclusion

Time has almost come to conclude on the work that have been done. But, before that, lets summarize what we did. At the very beginning of this thesis, we mentioned our motivations on writing this thesis. We wanted to work on adversarial-training to understand the underlining advantage of it over a more basic approach. We wanted to expose the linearity of the basic approach and how adversarial-learning generalizes. Also we wanted to figure out if this technique could work on another dataset.

After all the experiment we've been through, we can most probably conclude that basic-learning (A) on shallow-neural-networks **doesn't generalize** the data distribution it is trained on and is therefore too linear. We came to this conclusion by predicting on a this model an adversarial test-sets and observing how quickly it could be misled. Also, comparing another model (C) trained on random modifications, to our model A confirmed this observation, that a model trained to generalize a data distribution was less linear. This last point might sound obvious but is key as A could have been as good as C if it had generalized. Comparing the amount of neurons on the adversarial model (B) lead us to believe that B learns more functions than A does and might generalize with these new functions. Finally, comparing this technique on other datasets let us realize that this technique was specific to a certain type of inputs features and that non linear features most probably aren't concerned with adversarial-learning.

We didn't exactly found what adversarial-learning was specifically doing. My hypothesis would be that it better separate the classes by learning what makes the class different from the others. Visualizing the weights and comparing A and B's weights would have helped us on validating this hypothesis but we can't visualize patterns with this type of networks.

This last observation leads us to a desired **future work**: We would like to

confirm the hypothesis stated right above by training another type of network where we can visualize the class patterns and where the adversarial-learning is also better than the basic one.

Appendices

Appendix A

Weight initialisation on simple example

The figure A.1 shows our simple example initalized with some specific hand-engineered features.

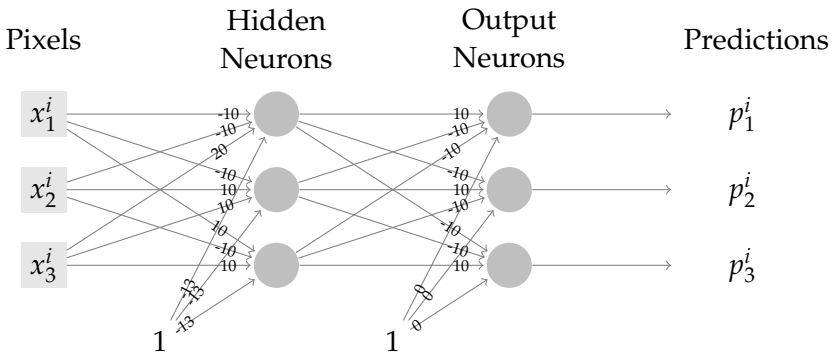

Figure A.1: Simple example with weight initialization

Appendix B

Hadamard and Kronecker products

The **Hadamar** product "$\circ$" is a mathematical operation used in between two matrices of same shapes (or two vectors). This operator computes an element-wise multiplication of "ij" elements of the matrices. Here is an example:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \circ \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{pmatrix}$$

The **Kronecker** product "$\otimes$" is a mathematical operation used in between two matrices. This operator augments each matrix elements (a) by another matrix (B) multiplying B by a.

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{pmatrix}$$

Appendix C

Vectors and matrices

On this appendix, we summarize the notation used for the neural networks and give the shape of the vectors in the case of our simple example (Se) and in the case of the 800 hidden neurons adversarial model trained on MNIST (model B). These shapes will be written in parenthesis at the end of each lines

- $\boldsymbol{x}^i$ is the inputs vector. In the case of an image, it has as many values as there is pixels. Se(3,1), B(784,1)
- $\boldsymbol{W}^l$ is a weight matrix where each columns represent a feature detector. w^l has as many columns as there is neurons on the layer l and as many lines as there is inputs in layer $l-1$. On the figure A.1 on page 51 the weights correspond to the numbers initialized on the edges in between each layers. Se(3,3), B(784,800)
- $\boldsymbol{b}^l$ is a bias vector. It has as many values as there is neurons on the layer l. Se(3,1), B(784,1)
- $\boldsymbol{a}^l$ is the output vector of a layer. It has as many values as there is neurons on layer l. Se(3,1), B(800,1)
- $\boldsymbol{z}^l$ is the input vector of a layer, what comes right before the transfer function (like the sigmoid). It has as many values as there is neurons on layer l. Se(3,1), B(800,1)

Bibliography

[1] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[2] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 4. springer New York, 2006.

[3] Charles Elkan. Maximum likelihood, logistic regression, and stochastic gradient training, 2013.

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

[5] Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

[6] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[7] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

[8] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 1998.

[9] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.