Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

ROBUST NEURAL NETWORKS AGAINST ADVERSARIAL EXAMPLES

Advisor: Dario MALCHIODI

Coadvisor: Nicolò CESA-BIANCHI

Thesis by:
Giuseppe CRINÒ
Student ID: 794281

Academic year 2017-2018

Contents

Introduction

This work is about how neural networks can be improved from a resiliency standpoint, against malicious inputs. Many researchers are doing serious work (cite papers here) both by finding novel forms of defenses but also by crafting new kind of inputs that defeat the state-of-the-art defenses. This thesis tries to do a small step in the former direction, by trying to measure the performance of a defense technique.

The work that we're using as a starting point is based on is [1]. In that paper, researchers use a technique called Principal Component Analysis to derive a filter for images. It turns out that applying that filter on each image before the classification reduces the rate of success for an attacker that provides input specifically forged to make the model misclassify it. We reproduced those results and swapped Principal Components Analysis with other similar dimensionality reductions techniques. The idea is that as there's nothing intrinsically special about Principal Components Analysis hence maybe other filters are better suited for the task.

The work is organized as follows: in Chapter 1 we provide some background material about Machine Learning in general, Neural Networks and Adversarial Examples. Chapter 2 is about the technological part of this work — the tools that we used to perform our experiments. Finally, Chapter 3 is about the actual experiments and measurements, comparing the performance of a Principal Components Analysis filter to the other filters.

1 Background

Machine Learning popularity exploded in last years. If you work in tech it's hard you didn't hear about large public datasets, neural networks, deep learning, or just how people speculates a majority of jobs will be substitued by machine trained from data.
Apart from a noisy hype, many services are actually built using Machine Learning and implementing them using different techniques is nearly impossible. Think of how Spotify generates playlists that are tailored to you musical taste[1], or just how YouTube gives you suggestions specifically for you [2] — even when you should do more serious work[2].
More and more developers are getting interested in this kind of technology and as more people work on these topics, more people build better tools, empoweing larger majorities of developers to write services using Machine Learning.
This chapter is organized as follows: Section 1.1 is about what is Machine Learning and the gradient descent procedure; in Section 1.2 we'll describe Neural Networks, a classifier that's used with great success in recent times; Section 1.3 explains what are adversarial examples and how they're generated.

1.1 Machine Learning

Quoting Wikipedia[3]

> Machine learning is a field of computer science that uses statistical techniques to give computer systems the ability to "learn" (e.g., progressively improve performance on a specific task) with data, without being explicitly programmed.

A typical example to introduce people with some scientific preparation to this topic is the one of linear regression [5]. In linear regression you have data coming in pairs (x, y) and you think that y can be linearly dependent on x. That is, your hypothesis is that

[1] https://support.spotify.com/us/article/daily-mix/
[2] https://www.ft.com/content/4f82a008-0096-11e8-9650-9c0ad2d7c5b5
[3] https://en.wikipedia.org/wiki/Machine_learning

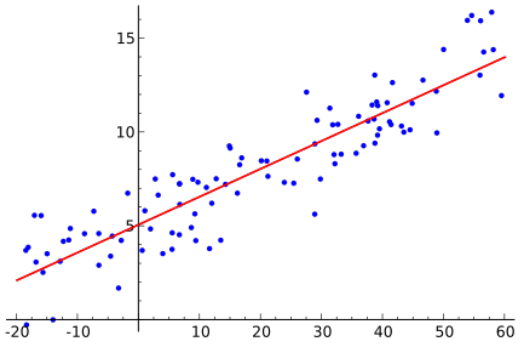


Figure 1.1: A linear model built out of a dataset. By Sewaqu [Public domain], from Wikimedia Commons.

$$y = wx + b \tag{1.1}$$

where w and b are called the weight and the bias respectively[4]. The problem is to find the best values for w^* and b^* such that y^* defined as

$$y^* = w^*x + b^*$$

is the closest possible to the associated value of x.
When both x and y are scalar, the problem of finding the best values for w and b becomes the problem of finding the best line that's *close* to all the data — see Figure 1.1.
You don't know which are the *correct* values for the weight and the bias so you either estimate them using analytical methods like the ones you probably learned in some math course or you derive the two values using a data-based procedure that minimizes the data minimizes the distance between the expected and the actual value of y.
The function that computes the distance is generally called the *loss* function and can be as simple as the L_1 distance [6] — i.e. the absolute value of the difference of the two values. Instead the algorithm that minimizes the loss can be any minimization procedure; one of the most known one is called the gradient descent [3]. The gradient descent procedure to minimize a general function $F(\boldsymbol{z}) : \mathbb{R}^n \to \mathbb{R}$ constists of iteratively computing

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n - \gamma \nabla F(\boldsymbol{z}_n) \tag{1.2}$$

[4] In other fields, w and b are more commonly known as the slope and the intercept.

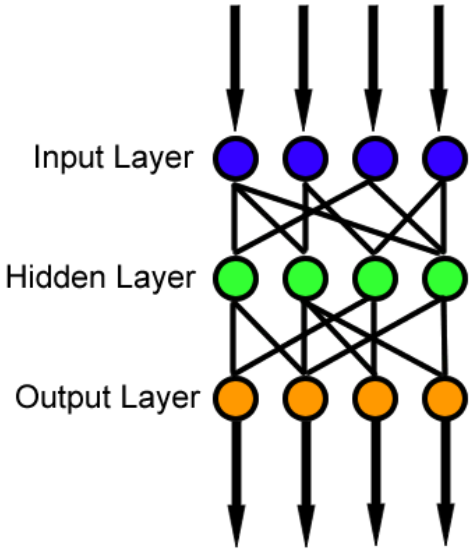


Figure 1.2: Neural network example. By User:Paskari [Public domain], via Wikimedia Commons

where $\boldsymbol{z}_n$ is the function argument at the algorithm's nth iteration, γ is the so-called *learning rate* while $\nabla F(\boldsymbol{z}_n)$ is the gradient of the loss function F evaluated at $\boldsymbol{z}_n$. The idea behind the procedure is that the gradient — that when your model is a line collapses to just being the derivative — is used to find the *direction* in which the function is growing; then it does an iterative step (whose size depends on the learning rate γ) in the *opposite* direction. That is, the procedure identifies where the loss functions is growing to perform a step towards a minimum instead. The minimum is of course local — it depends on where the procedure starts — and there's no guarantee that the algorithm will succeed in finding the best of these minima.
This algorithm has nothing to do specifically with linear regression. It's a general minimization algorithm. In fact, later in the thesis we'll use it even if we will never touch the topic of linear regression again.

1.2 Neural Networks

A neural network is one of the many models studied in Machine Learning. There are a variety of neural network types, but the ones we used throughout the thesis are called feedforward neural networks [3].

A feedforward neural networks is one of the simpler neural networks and it is the one that most people are first exposed to. Such a network is organized as a stack of layers of neurons. A layer is just a sequence of independent neurons, while a neuron is a *computational unit* that given an input and a state computes an output value.

In a feedforward neural network you have at least two layers: the input layer and the output layer. The input layer has as many neurons as the dimensions of the input — e.g. if the input space is $\mathbb{R}^4$ the input layer will have four input neurons. Neurons in the input layer do nothing but provide the other layers the data to make computation on — i.e. they all implements the identity function. In the output layers we have as many neurons as the number of classes in the classification task: each one has a value between 0 and 1, representing the confidence that the input belongs to that class. For example, if the third neuron outputs a value of 0.577, according to the model the current input belongs to the third class (the class related to the third neuron) with a confidence of 57.7%.

In between the input and output layer you typically have one or more the so-called *hidden* layers. Each hidden layer (as the output layer too) computes the composition of a linear function followed by the application of a *non*-linear function, called the activation function. That is, each neuron computes

$$\text{activation}(l\,W + b)$$

where l is the state of the previous layer, while w and b represents the state of the neuron that changes as the model is trained. The activation function can be any non-linear function; one of the simplest one is called the rectifier (see Figure ??)

$$\text{f}(z) = \max(0, z)$$

This sequence of linear function and activation function can be *repeated* multiple times increasing the hidden layers of neurons. Having several hidden layers of different width can improve the performance of the model. When the network consists of many layers, the network is referred to be a *deep* neural network. As a rule of thumb, the more the layers the better the performance of the model. Yet, that's not always the case *and* the deeper the network, the longer it takes to train that model.

In fact, deep learning is not a new idea and many consider the recent interest in neural networks as a consequence of the current available computational power that wasn't there in the 20th century.

> Advances in hardware enabled the renewed interest. In 2009, Nvidia was involved in what was called the "big bang" of deep learning [...] [Andrew] Ng determined that GPUs could increase the speed of deep-learning systems by about 100 times. In particular, GPUs are well-suited for the matrix/vector math involved in machine learning. GPUs speed up training algorithms by orders of magnitude, reducing running times from weeks to days. Specialized hardware and algorithm optimizations can be used for efficient processing.[5]

The way we used neural networks has been for classification tasks, that is given an input and a set of classes we want the network to be able to assign that input to the correct class. To understand how neural networks are used to perform classification it is useful to know the concept of one-hot encoding of data. In one-hot encoding possible values are binary numbers, that are legal only if there's only one 1 — e.g. 0001, 1000, ... but not 1001 or 1100.
One-hot encoding is used to map classes (cat, dog, mockingbird, etc.) in the training set to *states* of the output neurons in the neural network. For example, if the classification problem uses three classes, cat, dog and mockingbird, a one-hot value will be assigned to each class such to fit the *shape* of the output layer — that is, it will be assigned 100 to the class of cats, 010 to dogs and 001 to mockingbirds.
During the training phase the network is *shown* a configuration of the input neurons (the single input example) and a configuration of the output neurons (the class of the input example one-hot encoded), and the network is *confronted* with the configuration of the output it obtains using its current weights and biases in the hidden layers and *tries* to reduce this distance. Of course, it's the gradient descent procedure that does this job: the network is actually a *passive* object.

1.3 Adversarial examples

Just like gradient descent is used to change the weights and biases of the network to reduce the distance from the one-hot encoded class and the actual output of the network, we can use gradient descent on the input data to reduce the distance from the current output of the network to another output of our choice. This way we can manipulate classification.

[5] https://en.wikipedia.org/wiki/Deep_learning#History

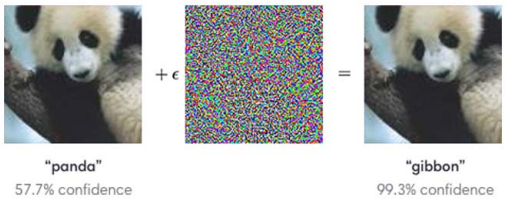


Figure 1.3: Panda misclassified to a gibbon, using carefully crafted noise.

What we get is an input data generated from the original input but modified in such a way that the input is *misclassified*. These generated inputs are called *adversarial examples*. Countrary to one might expect, for a human eye, an adversarial example are indistinguishable from the input that originated it. In Figure ?? we can see that the image classified as being the one of a gibbon is actually identical (not exactly of course) to the panda picture. That's why adversarial examples are a topic of interest for researchers: they make machine learning models behave in ways that diverge from their expected behaviour — in ways human *fail to predict*.
In a near future where machine learning models influence real-life decisions — "drive straight or stop the car?" — this becomes easily dangerous.
There are different techniques to generate these adversarial examples. One of the most notable one is called fast gradient sign and has been introduced by Goodfellow, et al. in [4]. There's no need to know much about this attack as many libraries already implements this technique — see 2.3. In fact we used the attack *as a service* without knowing too much of the details.

1.4 ...

2 Tools and libraries

In this chapter we describe the tools and libraries that we used to implement the set of experiments described in Chapter ??. This chapter is organized as follows: Section 2.1 is dedicated to TensorFlow[1] and its computational graph; Section 2.2 is for Keras, a high-level API for TensorFlow; Section 2.3 describes CleverHans, a library to generate adversarial examples; Section 2.4 talks about Scikit-learn, a library for building and training Machine Learning models, while Section 2.5 briefly mentions Google Cloud Platform, a service by Google that allows you to buy computational power to run your own code.

2.1 TensorFlow

TensorFlow is a C++ framework for Machine Learning released by Google. It uses a peculiar programming model called Data Flow that allows parallel and distributed computations, although it can be quite daunting to use when tinkering. In fact, operations to perform on data are first described and executed only at a later time. We believe this can be counter-intuitive for most people. For example, during the execution of the code shown in Figure 2.1, the variable `symbol` doesn't contain a reference to the integer 42 — as opposed to what would have been if `symbol` was an `int` variable. Instead, it contains a reference to a node of a *computational graph* (a description of the computation to perform on data in terms of nodes and edges) which will always output 42. In TensorFlow,

```
>>> import tensorflow as tf
>>> symbol = tf.constant(42)
>>> symbol
<tf.Tensor 'Const:0' shape=() dtype=int32>
```

Figure 2.1: Building a *constant* tensor

[1] https://www.tensorflow.org/

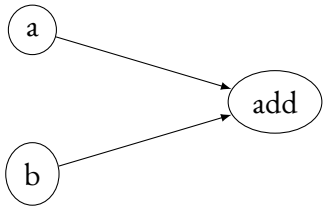


Figure 2.2: One of the simplest computational graph possible.

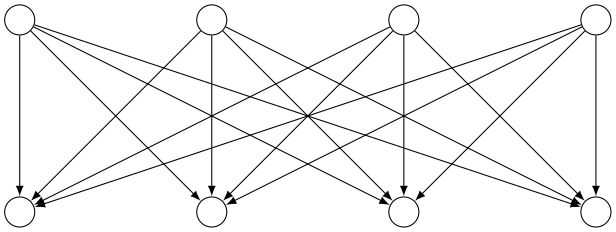


Figure 2.3: Feedforward neural network with only input and output layer

computation is done by connecting one or more of these nodes to other nodes, letting *tensors* (the object that you manipulate in TensorFlow) *flow* around this graph (in case of Figure 2.1, the tensor is the output of the only one node in the graph: the tensor is 42). Problem is that, in general, it's hard to determine beforehand what is going to be the value of a tensor, that is the *output* of the computational graph. This is slowing down exploratory analysis.

By the way, TensorFlow's computational graphs are a fundamental part of the framework. A computational graph is a directed graph representing a computation involving tensors. A tensor in TensorFlow is a generalized multi-dimensional array: it can be a scalar, a matrix, a batch of RGB images (which is a 4D vector[2]), etc. Each node represents an operation on tensors, while edges represent tensors.

For example, Figure 2.2 shows a very simple computational graph. It's the one associated to an addition of two *numerical* tensors (two tensors made of numbers: two `tf.float32`, two `tf.int64`, etc.): the first two nodes `a` and `b` output one tensor each; those are used to feed the `add` node that will output the tensor resulting from the sum of `a` and `b`.

The most commonly used way to program the computational graph is by using an API for the Python language. A perhaps interesting example of the kind of computation one can express in TensorFlow using Python would the implementation of a neural network. Figure 2.3 graphically represents the network we're going to implement: a simple feedfor-

[2] If you always thought humans can't visualize four dimensions all at once, think of this example.

```
>>> import tensorflow as tf
>>> W = tf.random_normal(shape=(4, 4))
>>> b = tf.random_normal(shape=(4,))
>>> X = tf.placeholder(shape=(None, 4), dtype=tf.float32)
>>> logits = tf.matmul(X, W) + b
>>> probabilities = tf.nn.softmax(logits)
>>> probabilities
<tf.Tensor 'Softmax:0' shape=(?, 4) dtype=float32>
```

Figure 2.4: TensorFlow commands to generate a feedforward network

ward network of four input neurons and four output neurons. As explained in Chapter 1, each layer in a feedforward neural network performs a linear operation on its input and apply an activation function on that result. That is, it computes

$$\text{activation}(X * W + b)$$

given its weight matrix and its bias vector, $W \in \mathbb{R}^{4\times4}$ and $b \in \mathbb{R}^4$ respectively, while activation is a *non*-linear function such as the rectifier or the softmax function.
To get an idea of how to do that in TensorFlow see Figure 2.4, where W and b are initialized drawing values from a standard normal distribution. Note that `X` is a *placeholder*: it is a TensorFlow way to make room in the graph for something whose value will be provided later. In Figure 2.4 that function is applied to the *logit*[3], i.e. the output of the matrix multiplication. As explained in Chapter 1 this is done for various reasons but one of the simplest ones is that it squashes the output of the network in $[0, 1]$ allowing to interpret it in terms of confidence levels — see Chapter 1.
Now, to get results out of the graph you need what's called a *session* in TensorFlow. Basically, a session *powers on* the graph, allowing you to run the computational graph with your own data and get actual tensors of actual numbers out of it. Of course building neural networks is pretty useless if you can't train them: at first the whole output is only based on the random weights and biases randomly extracted from a normal distribution — for example, probabilities in Figure 2.5 are totally non-sense: there's no training set and no training phase yet.

[3]https://datascience.stackexchange.com/a/31045/50780

```
>>> import numpy as np
>>> batch = np.random.rand(10, 4)
>>> batch
>>> array([[0.54485176, 0.33854871, 0.45185129, 0.79884188],
  [0.41204776, 0.23552753, 0.04101023, 0.47883844],
  [0.25544491, 0.7610509 , 0.49307137, 0.6098213 ],
  [0.02545156, 0.70459456, 0.22067103, 0.64743811],
  [0.92359354, 0.96497353, 0.45790538, 0.49380769],
  [0.13330072, 0.22947966, 0.02996348, 0.69954114],
  [0.38397249, 0.30473362, 0.87023559, 0.90153084],
  [0.77056319, 0.94843128, 0.39095345, 0.50572861],
  [0.90112077, 0.19240995, 0.48437166, 0.46200152],
  [0.98589042, 0.2013479 , 0.86091217, 0.55886214]])
>>> with tf.Session() as session:
...     session.run(probabilities, feed_dict={X: batch})
...
>>> array([[0.03751625, 0.8651346 , 0.05245555, 0.04489357],
  [0.11873736, 0.6301607 , 0.17646323, 0.07463871],
  [0.06252376, 0.82338244, 0.07284217, 0.04125158],
  [0.12086256, 0.6911012 , 0.14362463, 0.04441159],
  [0.03662292, 0.8575108 , 0.04834893, 0.05751737],
  [0.12984137, 0.63064647, 0.1834405 , 0.05607158],
  [0.01711418, 0.93650085, 0.02116078, 0.02522417],
  [0.04886489, 0.83023334, 0.06332602, 0.05757581],
  [0.033136 , 0.84808177, 0.05061754, 0.06816475],
  [0.01250888, 0.9262628 , 0.01797607, 0.0432523 ]], dtype=float32)
```

Figure 2.5: Running a computational graph within a session

```
>>> W = tf.Variable(W)
>>> b = tf.Variable(b)
>>> logits = tf.matmul(X, W) + b
>>> # redefine logits using variables
>>> probabilities = tf.nn.softmax(logits)
```

Figure 2.6: Making variables out of `W` and `b`

```
>>> from tf.nn import tf_cross_entropy
>>> y_true = tf.placeholder(shape=(None,), dtype=tf.float32)
>>> cross_entropy = tf_cross_entropy(logits=logits, labels=y_true)
```

Figure 2.7: Creating the cross-entropy operation

```
from tf.train import GradientDescentOptimizer as SGD
optimizer = SGD(learning_rate=0.5).minimize(cross_entropy)

with tf.Session() as session:
    session.run(tf.global_variables_initializer())

n_training_steps = 10
for i in range(n_training_steps):
    images, classes = next_batch()

    with tf.Session() as session:
        session.run(optimizer, feed_dict={X: images,
                                          y_true: classes})
```

Figure 2.8: Training a feedforward neural network built with TensorFlow

To train a network in TensorFlow we need a `tf.Variable`. A variable in TensorFlow is a tensor which can be modified by sessions, whose value persists across them. This allows you to basically add parameters to your computational graphs, allowing to iteratively *change* them. Using a learning procedure that modifies the `tf.Variables` to reduce the distance from the target function to the actual function computed by the network, makes the model *learning*. In the current example, the part of the graph that you want the learning procedure to modify are the tensors W and b. As in Figure 2.6 to build a `tf.Variable` in TensorFlow you can just wrap those tensors.

The last thing we want to do before abandoning this example is the actual training. To perform training we need a training set. As explained in Chapter 1, a training set consists of a bunch of data and a label associated to each input, representing the class of that data. As *training* basically means minimizing a loss function, we need that function: as in Figure 2.7, we're using the cross-entropy function. Cross-entropy is defined as

$$H(p, q) = -\sum_i p_i \log q_i$$

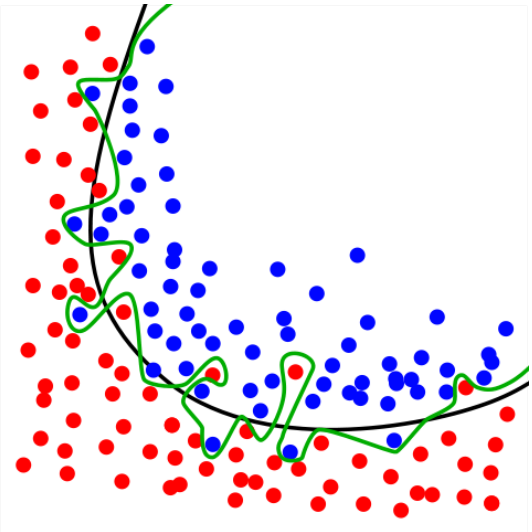


Figure 2.9: Green curve represents an overfitting classifier. By Chabacano [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)], from Wikimedia Commons

where p and q are two probability distributions. In our case, p is the *probability* distribution for an input (the classifier's confidence levels, the output of a neural network if the model is a neural network), and q is the one-hot encoding of the actual class of the input (which looks and it's in fact used here as a probability distribution).

The gradient descent algorithm is then used to iteratively do the job. TensorFlow implements the procedure via `GradientDescentOptimizer.minimize(loss)` that returns an operation that each time you run in a session will modify the graph's `tf.Variables`, aiming to lower the value of the cross-entropy *loss* function. We can build a loop to train the network over and over again, as in Figure 2.8. This will push the loss function toward a local minimum; hopefully a useful one, i.e. a minimum that corresponds to good results even on data outside the training set. Otherwise the model is said to *overfit*. That is, the model *fit* so well over the training set that fail generalize to the whole input space. For example, in Figure 2.9 we have plotted two classifiers: one drawn as a green curve, the other as a black curve. The *green* model has a better accuracy (that is, it has a high rate of correct classifications) on the training set, but its characteristics are clearly so much dependent on the set the models has been trained that it will almost surely fail to keep that high accuracy on data outside the training set — while the *black* model has a higher probability to be a good model.

```
from keras.engine.base_layer import Layer
import tensorflow as tf

class Add42(Layer):
    def __init__(self):
        self.fortytwo = tf.constant(42)

    def __call__(self, X):
        return tf.add(X, self.fortytwo)

model = Sequential([Dense(...), Add42(), ...])
```

Figure 2.10: A toy-example for a Keras layer

```
>>> from keras.models import Sequential
>>> model = Sequential([
... Dense(batch_input_shape=(None, 4), units=4),
... Activation('softmax')])
```

Figure 2.11: Feedforward network using Keras layers

2.2 Keras

Keras[4] is a library for building Machine Learning models using a simple API that abstracts away the interface of a backend of choice, e.g. TensorFlow. When you use the library, you're encouraged to manipulate *layers* instead of plain tensors. In fact the whole concept of *tensors* is hidden away by the library. To implement your model you're expected to stack layers one on top the other, expressing a *sequential* manipulation of the data[5]. Under the hood, a `Layer` is simply a callable object whose `__call()__` method takes a TensorFlow tensor `X` and manipulates it — see Figure 2.10.

As shown in Figure 2.11, to implement the simple feedforward neural network of Figure 1.2 one would stack a `Dense` layer of 4 hidden neurons and an `Activation` layer implementing the softmax function. Compare that to the TensorFlow implementation we did in Section 2.1 and you'll realize how much easier is to read it and to think about it.

This nicer API that allows to immediately think about model's architecture has the cost of an abstraction level that at the time of this writing in our opinion is not enough stable

[4] https://keras.io/

[5] https://keras.io/#getting-started-30-seconds-to-keras

```python
from cleverhans.attacks import FastGradientMethod
from cleverhans.utils_keras import KerasModelWrapper

# model is a Keras model
cleverhans_model = KerasModelWrapper(model)
attack = FastGradientMethod(cleverhans_model)

example_sym = attack.generate(model.input, **kwargs)
# example_sym will give the adversarial examples
# when run in a session.
```

Figure 2.12: Generating adversarial examples using CleverHans

to allow programmers to be completely oblivious of the original backend — in my case TensorFlow.

2.3 CleverHans

CleverHans[6] is a library for building adversarial examples written by Google, OpenAI and Pennsylvania State University. It implements a variety of attacks[7] including fast gradient sign (see Chapter 1) against neural networks and it's compatible with models built with Keras or just plain TensorFlow. It leverages the computational graphs of TensorFlow to generate adversarial examples, so using the library without knowing TensorFlow is not easy.

To generate adversarial examples for a given model CleverHans needs to be able to read some internals of the model — inputs are generated in a white-box approach. CleverHans already provides a couple of utilities to do that *model inspection* — e.g. for Keras there's a `KerasModelWrapper`[8] — transforming the model into an object that CleverHans is able to handle. Once a model has the interface CleverHans expects, it's possible to choose the attack technique via classes inhereting from the same `Attack` class. They are all exposing a `generate` method that will return a node of the corresponding computational graph; when you run it in a session it will return the related adversarial examples. See Figure 2.12 for a short example.

[6] https://www.cleverhans.io/
[7] https://cleverhans.readthedocs.io/en/latest/source/attacks.html
[8] https://github.com/tensorflow/cleverhans/blob/66125be/cleverhans/utils_keras.py#L101

```
import sklearn.decomposition
pca = sklearn.decomposition.PCA(n_components=100)
pca.fit(training_set)

batch = next_batch()
filtered_batch = pca.inverse_transform(pca.transform(batch))
```

Figure 2.13: Using `sklearn.decomposition.PCA` to get a *filtered* image

CPU platform	Intel Haswell
memory	60 GB
vCPUs	16
GPU	1 x NVIDIA Tesla P100

Figure 2.14: Information of the machine used via GCP.

2.4 Scikit-learn

scikit-learn is a Python library written by David Cournapeau providing a number of models, learning algorithms and data manipulation utilities for Machine Learning. It's pretty popular for fast prototyping as it uses a simple and consistent API, with the ability to handle numpy arrays or even Python lists — instead of having to learn about the computational graph.

We've used scikit-learn to reuse a couple of decomposition algorithms[9] without having to implement them. This required a bit of thinking as while Keras and TensorFlow use a lot the computational graph, scikit-learn is completely oblivious of that structure.

The way I've used scikit-learn decomposition algorithm has been by leveraging classes exposing both a `transform(X)` and a `inverse_trasform(X)`. This way I built *filters* that *reduced* the amount of information in the original data. See Figure 2.13.

2.5 Google Cloud Platform

Google Cloud Platform is a set of services that allows people to buy computational power from Google. In our case, we used GCP to get access to a GPU for few dollars a month[10].

[9] http://scikit-learn.org/0.20/modules/classes.html#module-sklearn.decomposition

[10] https://cloud.google.com/blog/products/gcp/introducing-improved-pricing-for-preemptible-gpus

```
[g@x220 ~]$ gcloud compute ssh root@bowser -- -X
No zone specified. Using zone [us-east1-b] for instance: [bowser].
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1014-gcp x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.


Last login: Thu Sep 27 15:35:52 2018 from 79.24.139.192
root@bowser:~#
```

Figure 2.15: Using `gcloud` to connect via ssh to Google remote *machine*

Since we had $300 of free credit we chose an expensive machine type as described in Figure 2.14. That made running experiments a lot faster in many cases.

3 Implementation of robust networks

In this chapter we describe the experiments that I did during my thesis work. See Chapter 1 for a rationale of what we're going to. Section 3.1 is dedicated to how the models have been trained and how the number of epochs of training has been chosen, Section ?? talks about how I *deployed* a filtering technique over an existing model and how I made that choice, and finally Section ?? is dedicated to a comparison of the different filtering techniques and which is the one that ended up to perform better as a defense against adversarial examples.

3.1 Number of epochs

As we started to test the performance of various setups we were basically generating a lot of different models that had to be trained first, before running experiments on them. Each one had different characteristics and different rates of learning so fixing a pre-defined number of epochs to train every model for might have been resulted in comparing models with very different accuracies: as explained in Chapter 1 the accuracy has a direct impact on the resiliency of the model against adversarial examples — that would have been make a filter technique win even if the real reason for that result was a low accuracy. One choice would have been to choose a predefined accuracy for all the models to reach but choosing a too low value would have been resulted in unrealistic measurements (no one wants to use models with accuracy lower than 40% for example) while choosing a high value for the accuracy had the risk of potentially being unreachable for some models.
The approach that we decided to use to somewhat solve this problem was to come up with *heuristics* to decide whether a model has stopped learning or not. It's not perfect: models will still reach different accuracies. But hopefully they will be all closer. The number of epochs is then defined dynamically as the model trains.

As we wanted to test models with the highest possible accuracy that we can make them reach we made the model reduce the learning rate (see Chapter 1) when the accuracy of the model didn't seem to improve anymore. In fact, "models often benefit from reducing the learning rate by a factor of [2 up to 10 times] once learning stagnates"[1].
Now, the heuristics we chose were:

1. stop learning after the accuracy does no longer improve over a specified number of epochs

2. stop learning after the model's weights are pretty stable over a specified number of epochs

Note that both the heuristics wait after a "specified number of epochs" to make their decision. We're calling this value the *patience*, as that's the amount of *time* the heuristic take before making its final decision.
To prove that both the heuristics work and to choose the one that worked better I decided to stick with a single model of which we assume to more or less know the number of epochs needed to reach an accuracy of 97% and see if we can reach that same accuracy even without specifying the number of epochs but relying only on the heuristics to stop training. Throughout the codebase we called this setting training the model for a number of epochs equals to -1.
The model used is a feedforward network of two layers of 100 hidden neurons each and ten output neurons. I'm going to call it `fc-100-100-10` for obvious reason throughout the rest of the document. In [1], Princeton researchers trained that model for 500 epochs achieving an accuracy of circa 97.5%. That's our baseline.

3.1.1 Early stopping heuristic

This heuristic is already implemented by Keras as a callback to the learning phase[2]. It checks if over the patience a chosen metrics has stopped improving. If that happens we deduce the model stopped training. We set a patience of 60 epochs as on our Google Cloud Platform machine 60 epochs corresponds to a whole minute.

[1] https://github.com/keras-team/keras/blob/2ad932b/keras/callbacks.py#L991-L992
[2] https://keras.io/callbacks/#earlystopping

	fixed lr	reduce lr
early stopping	97.44%	97.18%
stable weights	97.66%	97.6%

Figure 3.1: Accuracies when training with different techniques

3.1.2 Stop on stable weights heuristic

We implemented this heuristic from scratch. Data on model weights are collected. Taking the weight that has the largest standard deviation over the patience, if that standard deviation is below a certain threshold, according to the heuristic, the model has stopping learning and getting stable. Again, we set a patience of 60 epochs.

3.1.3 Data obtained

Our baseline was that of `fc-100-100` trained for 500 epochs reaching an accuracy of 97.55%. To check how stable was that accuracy we trained the same network for another 500 epochs reaching an accuracy that was only slightly higher — 97.71%. Reducing the learning rate on plateau instead decreased the accuracy to 97.47%, both for 500 epochs and 1000 epochs. That said, as we planned to let our models train until an heuristics decides to stop it we thought longer training sessions would not be our main problem for our future experiments[3].

You can see results in Figure 3.1. The combination that seemed to work better was to stop training on model's stable weights, keeping the learning rate fixed, getting a final accuracy of 97.66%. That said, as the difference between using a fixed value for the learning rate or reducing that on plateau was not that distinct and given that the reducing the learning rate intuitevely felt useful in the general case[4] we chose to reduce the learning on plateau. We understand the decision is rather arbitrary but we chose that. As for the heuristics we chose to stop training on stable weights.

In the following sections all the models considered have been trained using this setup then. That is, training for an undefined number of epochs, stopping when the model's

[3] Actually there's another problem: that the model escapes the plateau but the accuracy of the model grows so slowly that an heuristic will block its training as it thinks it's *converging*. Using the heuristic of stopping training when model's weights are stable should keep away this situation

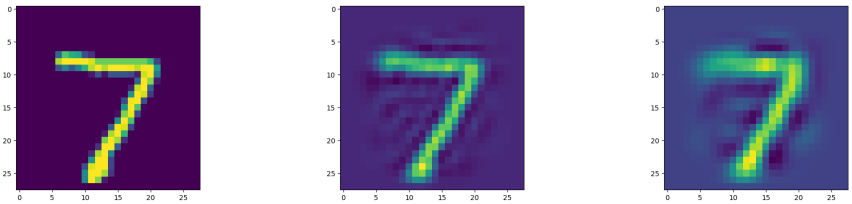
[4] https://datascience.stackexchange.com/questions/37163/is-it-a-good-practice-to-always-apply-reducelronplateau-given-that-models-b/37190

weights become stable — i.e. the related standard deviation reach a value under a threshold of 0.5 over 60 epochs — reducing the learning rate after a plateau of 30 epochs.

3.2 Deploying filters

When testing different filter tecniques just *finding the right one* is not enough. In fact how do you *deploy* that filter technique in your model is something you have to choose. For example you can add your filtering technique to the data pipeline *before* training and let the model train on filtered input.
We had a couple of alternatives here and we had to measure their different performance in order to choose which one was the better fit for our work.
We identified at least two different fashions to integrate a filter technique in our models. We can either add a filter layer to a model already trained on unfiltered input, or we can add a layer to an already trained model and retrain the network for some epochs after this layer has been added. Basically the difference is that in the latter case, the model has been trained only on filtered input. Inspired by lexicon used in [1], we named the first technique as *reconstruction* and the second one *retraining*.[5]
We did some measurements to better understand which technique was better was not that clear.
We took the same model — `fc-100-100-10` — and deployed the PCA[6] filter technique using both reconstruction and retraining. Changing the number of components retained (see Chapter 1) we obtained dozens of models and trained all them leveraging the heuristics described in Section ??.
We attacked each model using the fast gradient sign technique (see Chapter 1) obtaining what was the adversarial success score given the value of η[7].
Then we compared the adversarial success score for each model trained using reconstruction to the attacker performance on the same model now built using retraining. By subtracting these two values, making an average for each models pair and finally making an

[5] There's another alternative which is to initialize the model, add the filter layer then train the network while it's still *untouched*. It basically consists in making the network never to see unfiltered input. This seemed to make things so much better for the attacker than both reconstruction and retraining that we just avoided to compare it with the other approaches.
[6] https://en.wikipedia.org/wiki/Principal_component_analysis
[7] As explained in Chapter 1, η is the the freedom given to the attacker: the more the freedom the easier is to forge an input.



(a) 784 components (b) 100 components (c) 40 components

Figure 3.2: Retaining a different number of components for a PCA filter.

average for all the pairs, we got what was the average gain (or loss) in the model's resiliency for reconstruction over retraining.

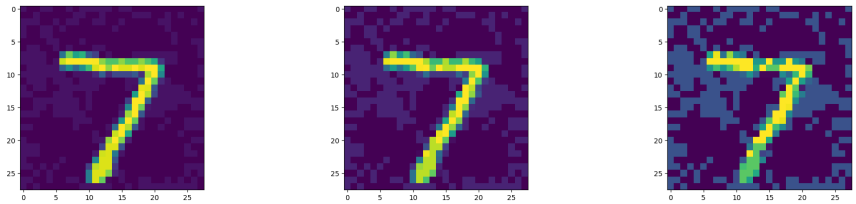
We wrote a script `retraining-versus-reconstruction.py` implementing the comparison described above and we obtained that retraining is only 1% more effective than reconstruction. This is a rather disappointing result. In fact, as implementing retraining consistently throughout the codebase was harder than to stick with reconstruction given the results we decided it was not worth the effort and we abandoned retraining.

For this reason in the following sections we will talk only about models with filter techniques deployed using reconstruction.

3.3 Filters better than PCA

In this chapter we'll test a couple of image filters against adversarial input. The intuition behind the idea of filtering the image is that to forge an adversarial input the attacker will try to put some noise distributed in the image. Trying to be stealthy the noise will be hidden to a human eye. As these filters highlight the *important* features of an image, that is more or less what an human eye sees, we hope they cut out the noise introduced by the attacker. Intuitevely, the more the attacker wants to be stealthy the more these filters are likely to succeed in deleting the attacker noise.

The idea is taken from [1] in which a *PCA filter* (that is a filter obtained applying a PCA transformation then its inverse, as explained in Section 2.4) is applied to the input of a feedforward neural network aiming to reduce the probability of success for an attacker forging adversarial examples against the model. We'll take this idea further by using a decomposition technique other than PCA: hopefully other filters will work better.



(a) $\eta = 0.05$ (b) $\eta = 0.1$ (c) $\eta = 0.25$

Figure 3.3: Adversarial examples as η increases.

Just like PCA and as already explained in Chapter 1 each of these decomposition techniques can be more or less destructive in regards of the original image. Intuitevely these filters, like PCA, *decompose* the original image of n pixels into a vector of less than n components. From this vector it is possible to reverse the decomposition operation obtaining again an image of n pixels but starting from a smaller amount of information the portion of information lost is *interpolated*. For example in Figure 3.2 a decreasing number of components is retained, that is a decreasing amount of information of the original image is kept — the rest is interpolated using an inverse function of the PCA transformation. Notice how the quality of the image degrates making the number 7 less and less recognizible but the feature of the number 7 are the ones that survive the most. That happens both for the human eye and the model.

The idea is to find a balance between this degradation that will slowly remove the features that allow to recognize the number 7 and remove the noise maliciously introduced by the attacker. Intuitively the more invasive the filter the better will be the defense. Unfortunately the better the defense the more the lost accuracy too: two different images can be confused as the same image for the model as the information that distiguished the two images is now potentially lost. That means that the *best* filter will be the one that provides the best defense given the accuracy lost by the model is not *too much*.

The attack used is called Fast Gradient Sign. As it has been already described in Chapter 1, the attacker can be more or less stealthy by choosing a higher or lower value for η respectively — see Figure 3.3. This makes Fast Gradient Sign a parameterized attack, meaning that studying which filter is more resilient to attacks would require us to use different values of η.

As that would make the number of cases to study combinatorically explode we decided to fix a value of η to 0.1 and consider fast gradient sign as a single type of attack with no

Decomposition technique	Inverse function
`DictionaryLearning`	no
`FactorAnalysis`	no
`MiniBatchDictionaryLearning`	no
`MiniBatchSparsePCA`	no
`SparsePCA`	no
`SparseCoder`	no
`FastICA`	yes
`IncrementalPCA`	yes
`KernelPCA`	yes
`NMF`	yes
`PCA`	yes
`TruncatedSVD`	yes

Figure 3.4: Decomposition techniques implemented by sklearn 0.20

parameters. 0.1 has been chosen as it's a median value between a completely ineffective attack of η equals to 0 and an attack that's detectable by a human eye — that is, η equals to 0.25.
In our gray-box setting (see Chapter 1) we measured an accuracy on the MNIST test set of 97.39% and an adversarial success score — that is, the probability of success for the attacker — of 81%. We expect the latter to go down as we increase the degree to which the filter technique is destructive. Yet as we try to stop the attacker we have to reduce the accuracy of the model as well.

3.4 Decompositions

When it comes to testing various filtering techniques we had two choices: implementing them from scratch using TensorFlow — as we did at first with PCA — or we could leverage the `decomposition` module of sklearn[8]. As explained in Chapter 2 to derive a filter from a decomposition technique we need to have both the transformation function — that performs the dimensionality reduction that names the class — and its inverse function. `sklearn.decomposition` implements the former for all of its classes but the related inverse function is not always available.

[8]http://scikit-learn.org/0.20/modules/classes.html#module-sklearn.decomposition

As shown in Figure 3.4 of 16 decomposition techniques implemented in `sklearn.decomposition` only half of them have the inverse function implemented, or defined — I guess that for some of them the inverse function does not even exist, but didn't make any research on that.
Of course each of these transformers had been *fitted*, as the function they compute is dependent on the dataset you're going to use. For example, to do PCA filtering on the MNIST dataset you need to fit the transformer on the MNIST dataset. This required time and some transformers — namely the KernelPCA — didn't succeeed in fitting, failing to *converge* : I had to discard them even if they had an inverse function.
Once transformers has been fitted we can add them as a preprocessing layer in the models in a *reconstruction* fashion — as explained in Section ??.

3.5 Better than PCA

Now we're ready to answer the question that motivated the thesis — see Chapter 1. That is, we started from the work in [1] where a PCA filtering technique has been proposed as a defense against adversarial examples. We want to find out if we can find some filter technique that works better than PCA.
Answering this question required a bit of reasoning. In fact, defining what's *better* than PCA wasn't straightforward, especially since models using with different filter technique had different accuracy and comparing all the models together would result in priviledging the models with a low accuracy — as the lower the accuracy the harder is for the attacker to forge adversarial examples.
So instead of defining an index that would compute a score of the defense provided by the filter somehow weightening the accuracy of the model — which we did at first —, we decided to partition the models in intervals of accuracy and tried to find the best filter restricted to that interval.
What we did find is that it's hard to improve over PCA. The performance of the other filters is very similar to the performance of PCA, as shown in Figure 3.5. The only interesting improvement is for using TruncatedSVD on a model whose accuracy is of circa 97.39%; in fact, we registered an improvement of 11% over the adversarial score for the same model defended by a PCA filter.

≈accuracy	filter	accuracy	adversarial score
97.39%	TruncatedSVD	+0.02%	-11%
97.36%	PCA	N/A	N/A
97.32%	IncrementalPCA	+0.02%	-0.05%
97.17%	NMF	-0.17%	-1.6%
93.8%	PCA	N/A	N/A

Figure 3.5: Comparing filter performances in respect to PCA.

Conclusion

This work took more or less three months, basically a student's summer. This has been my first experience with Machine Learning so I had to spend some time learning about fundamentals of Machine Learning: what's the training set? A validation set? The learning rate of the gradient descent algorithm? Or simply what's a neural network?
The research that I proposed to my advisors was an attempt to merge the curiosity of what's a hot topic right now — Machine Learning and neural networks — with my passion of watching computers break. I find that the topic of adversarial examples can be as fascinating to security interested people just like a memory corruption can be. Tools and culture is a lot different from the traditional hacker one but I think that if Machine Learning will become more ubiquitous more security researchers will join the field[9].
As to me, this work was the shortcut to learn about Neural Networks. I had some book titles that I wanted to read by the summer but when presented the opportunity to learn about that same topic hands-on I couldn't resist. A lot of time has been used to glue different libraries together, reading documentations and trying to not depress when confronted with crazy long researchers' papers.

[9] "Weaponizing Machine Learning", talk at DEF CON 25

1. A. N. Bhagoji, D. Cullina, C. Sitawarin, and P. Mittal. “Enhancing robustness of machine learning systems via data transformations”. in: *Information Sciences and Systems (CISS), 2018 52nd Annual Conference on*. IEEE. 2018, pages 1–15.
2. P. Covington, J. Adams, and E. Sargin. “Deep Neural Networks for YouTube Recommendations”. in: *Proceedings of the 10th ACM Conference on Recommender Systems*. New York, NY, USA, 2016.
3. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
4. I. J. Goodfellow, J. Shlens, and C. Szegedy. “Explaining and harnessing adversarial examples (2014)”. *arXiv preprint arXiv:1412.6572*.
5. S. Hartshorn. *Linear Regression And Correlation: A Beginner’s Guide*. Kindle Edition. 2017.
6. E. F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, 2012.