Robust neural networks against adversarial examples

by
Giuseppe Crino'

Advisors: Malchiodi Dario, Co-advisor: Cesa-bianchi Nicolo'
Student badge number: 794281

Contents

1 Background

2 Tools and libraries

In this chapter I describe the tools and libraries that I used to implement the set experiments described in Chapter ??. This chapter is organized as follows: Section 2.1 is dedicated to TensorFlow [1] and how its computational graphs work; Section 2.2 to Keras, a high-level API for building Machine Learning models without working with low-level libraries like TensorFlow; Section 2.3 to CleverHans, a library to generate adversarial examples against a given model; Section 2.4 to scikit-learn, a library for building and training Machine Learning models; finally Section 2.5 is about Google Cloud Platform, a service by Google that allows you to buy computational powers to run your own code.

2.1 TensorFlow

TensorFlow is a C++ framework for Machine Learning released by Google. It uses a programming model called Data Flow that aims to allow distributed and parallel computations. While this *paradigm* makes TensorFlow suitable even for research and production environments, it can be quite daunting to use when tinkering. In fact, computational graphs are built and only later executed. This is counter-intuitive at first.
For example, in Figure 2.1 `symbol` doesn't contain a reference to the integer 42. Instead it contains a reference to a node of the computational graph (which will always output 42). In general until you don't run the computational graph it's hard to determine what's going to be the value of a tensor ("the main object you manipulate and pass around [in TensorFlow]"[2] – see below). That's slowing down exploratory analysis.

2.1.1 About computational graphs

TensorFlow 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

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

Figure 2.1

[1]https://www.tensorflow.org/
[2]https://web.archive.org/web/20180921135324/https://www.tensorflow.org/guide/tensors

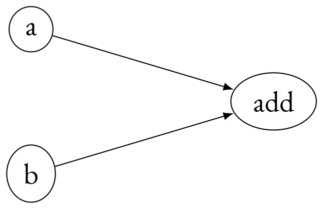

Figure 2.2: An easy computational graph

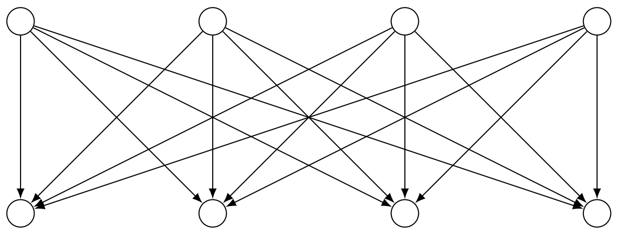

Figure 2.3: A simple neural network

is a multi-dimensional array: it can be a scalar, a matrix, a batch of RGB images (which is a 4D vector), ... Each node represents an operation on tensors, while edges represent tensors. (This distinction between nodes and edges is more important from a formal standpoint than a practical one: actually thinking of tensors as nodes makes no difference to the best of my knowledge.)
Figure 2.2 shows a very simple computational graph. It's the one associated to an addition of two `tf.float32`'s (i.e. two tensors made of 32-bit float numbers). The first two nodes `a` and `b` output a tensor each. Those are used to feed the `add` node that will output the tensor resulting from the sum of `a` and `b`.
A more interesting example would be to implement a neural network using a computational graph. For example, Figure 2.3 graphically represents a simple neural network of four input neurons and four output neurons.
Running this network requires computing

$$X * W + b$$

where $W \in \mathbb{R}^{4\times4}$ is a matrix of weights and $b \in \mathbb{R}^4$ is a vector of biases, while X is a *batch* of input vectors in $\mathbb{R}^4$. When the input is a flattened image of 2 by 2 pixels this neural network can be used to classify 4 different classes of images.
Note that the `tf.nn.softmax` function is applied to the result of the matrix multiplication. This is done for various reasons: it introduces a *non-linear factor* in the network computations allowing it to learn to compute any function – what's known as the "Universal approximation theorem"[3]; also it normalizes the results of the neural network in $[0, 1]$ allowing to interpret the output in terms of confidence levels or, as they are improperly called, *probabilities*.
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

[3]Actually, this network is too simple to fall under the universal approximation theorem. In fact you need at least one hidden layer. See https://en.wikipedia.org/wiki/Universal_approximation_theorem

```
>>> 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: Using the TensorFlow Python API for $X * W + b$

actual tensors of actual numbers out of it. In Figure fig:use-session we use a session to get probabilities out of our neural network for a batch of 2×2 images.

Of course building neural networks is pretty useless if you can't train them. In fact the probabilities above are totally non-sense: they're only based on the initial weight and bias randomly extracted from a uniform distribution. There's no training set, no training phase yet. To train a network in TensorFlow we need a `tf.Variable`. A variable in TensorFlow is a tensor whose value persists across sessions and that can be modified. This allows you to basically add parameters to your computational graphs, allowing you to compute different functions with the same graph. That, combined with a learning procedure that modifies the `tf.Variables` allows you to train your model.

To build a `tf.Variable` in TensorFlow you can just wrap an existing tensor. In the example above, the part of the graph that you want to parameterize are the weight matrix W and the bias vector b.

To perform training we now need a training set. A training set in this example would consist of several 2×2 images and a label associated to each image representing the class of that image.

Training basically means minimizing a loss function between the provided labels of the training set and the labels guessed by the neural network. The loss function we're going to use is called *cross-entropy* and TensorFlow implements it via `tf.nn.softmax_cross_entropy_with_logits_v2`.

To minimize the cross-entropy we're using the "gradient descent algorithm" – a minimization technique that works iteratively. TensorFlow implements that algorithm via the `GradientDescentOptimizer` class. Instances of that class expose a `minimize` method that returns a tensor that each time you run it in a session will modify the graph's `tf.Variables`, trying to lower the value of the loss function.

So a typical training phase is the one in Figure 2.8.

As you increase `n_training_steps`, `W` and `b` will be modified pushing the value of the loss function towards a local minimum. That makes the model learn how to correctly classify images similar to those in the training set.

```
>>> import numpy as np
>>> ten_two_by_two_images_batch = np.random.rand(10, 4)
>>> ten_two_by_two_images_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:
...     # we're _feeding_ the X placeholder with an actual numpy array
...     session.run(probabilities, feed_dict={X: ten_two_by_two_images_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)
>>> #             ^^^^^^^^^ the last image is of the second class with a confidence of 92%
```

Figure 2.5

```
W = tf.Variable(W)
b = tf.Variable(b)
logits = tf.matmul(X, W) + b # redefine logits: using different tensors now
probabilities = tf.nn.softmax(logits)
```

Figure 2.6

```
y_true = tf.placeholder(shape=(None,), dtype=tf.float32)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=y_true)
```

Figure 2.7

```
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()) # needed.

n_training_steps = 10
for i in range(n_training_steps):
    images, classes = next_batch()
    # fictitious function returning batches from the training set

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

Figure 2.8: Manually training a TensorFlow neural network

2.2 Keras

Keras is a high level library for building Machine Learning models. It consists of a simple API and bindings for a backend of choice, most notably TensorFlow. When you use the library, you're encouraged to use *layers* instead of plain tensors. In fact the whole concept of TensorFlow tensors is hidden away by the library abstractions. To implement your model you stack layers – basically you build a list of layers. For example, to implement a simple neural network one would stack a `Dense` layer of 4 hidden neurons and an `Activation` layer for the Softmax function.

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

Under the hood, a `Layer` is simply a callable object whose `__call()__` method takes a TensorFlow tensor `X` and manipulates it. For example a simple custom one would be

```
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)
```

While the idea is beginner friendly – as it allows people to immediately concentrate on model's architecture instead of thinking about tensors – at the time of this writing the abstraction can become leaky when things go wrong, needing the programmer to know about the TensorFlow computational graphs to debug her program successfully.

2.3 CleverHans

CleverHans is a library written by Google for building adversarial examples. It implements a variety of attacks against neural networks (FGS, T-FGS, Carlini and Wagner, ...) and it's compatible with models built with Keras or plain TensorFlow. It uses the same computational graphs of TensorFlow to generate adversarial examples; again, as you use the library more and more understanding computational graphs becomes a necessity.

2.4 Scikit-learn

Scikit-learn is a Python library written by David Cournapeau providing a number of models, learning algorithms and other utilities for Machine Learning. It's perfect for fast prototyping as it uses a simple and consistent API and handles numpy arrays or even Python lists.
I've used scikit-learn to borrow a couple of decomposition algorithms (PCA, FastICA, ...) without having to implement them. This required a bit of reasoning as reusing scikit-learn algorithms on TensorFlow graphs was not straightforward.

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 Google Cloud Platform to get access to a GPU for few dollars a month. That made running experiments a lot faster in many cases.

3 Robust networks

Before running experiments on how ML models can be *shielded* against adversarial examples we had to create and train them.

3.1 Motivation

As we're training a variety of models it's useful to find a strategy to find an appropriate number of epochs to train the model for. Fixing a number of epochs (say 500) and train all the models for that same number of epochs feels *cheesy*: some models can learn faster, other slower, and you can end up comparing models that are at different stage in their learning phase.
What we did was trying to *understand* when the model has done learning and stop training there. We used two approaches/heuristics that are described in the following sections.
Note that trying to make the model learn better we reduced the learning rate on plateau: if for a fixed number of epochs the accuracy on the validation set does no longer improve, the learning rate is reduced. This is known to help models to move on when the gradient descent algorithm is stuck somewhere.
To make our experiments here we used a fully-connected neural network with 2 layers of 100 hidden neurons each and 10 output neurons – *FC-100-100-10*. This structure is the same described in ???, which is the paper we got inspiration from. ??? trained FC-100-100-10 for 500 epochs and we assume they chose that number as it gave them the best results on the test set.
What we want to do is to empirically prove that we can achieve the same accuracy on the test set without training FC-100-100-10 for exactly 500 epochs but instead stopping training when the model stops learning according to at least one of our heuristics.

3.2 EarlyStopping

EarlyStopping is one of the two heuristics. It's already implemented by Keras as a Callback. It checks if over a specified number of epochs (the *patience*) a specified metrics stopped improving. When that happens the model is asked to stop training there. We set a *patience* of 60 epochs as on Google Cloud Platform 1 epochs runs is 1 second; a patience of 60 epochs means that the model didn't improve for a whole minute.

3.3 StopOnStableWeights

StopOnStableWeights is the second heuristic. It has to be implemented from scratch. This heuristic collects data on model's weights over a specified number of epochs called the *patience*. Then, the same weight over different epochs is checked: if it has a small standard deviation maybe it's getting stable. If the maximum of these standard deviation is below a certain threshold we conclude the model's weights are getting stable, hence the model is stopping learning.

3.4 Data obtained

We obtained the following data:

- Model trained for 1000 epochs: accuracy on the test set of 97.71%
- Model trained for 500 epochs: accuracy on the test set of 97.55%
- Model trained for 500 epochs, reducing learning rate on plateau after 30 epochs: accuracy on the test set of 97.47%
- Model trained for -1 epochs, stopping after accuracy on the validation set stopped improving for 60 epochs: accuracy on the test set of 97.44% (trained for 424 epochs)
- Model trained for -1 epochs, stopping as model weights had a maximum standard deviation of 0.5 over 60 epochs: accuracy on the test set of 97.66% (trained for 614 epochs)
- Model trained for -1 epochs, stopping after accuracy on the validation set stopped improving for 60 epochs, reducing learning rate on plateau after 30 epochs: accuracy on the test set of 97.18% (trained for 510 epochs)
- Model trained for -1 epochs, stopping as model weights had a maximum standard deviation of 0.5 over 60 epochs, reducing learning rate on plateau after 30 epochs: accuracy on the test set of 97.6% (trained for 514 epochs)

3.5 Conclusion

From the above results the best approach seems to be using the *StopOnStableWeights* heuristics with a patience of 60 epochs, reducing the learning rate on plateau after 30 epochs.
In the following chapters we'll use this setting for training models, even when they're no longer the FC-100-100-10 used here.
As we're testing different filter techniques it's important that we understand how we're implementing these techniques in our models as defenses.

3.6 Motivation

There are at least two different fashions to integrate a filter/decomposition technique in our models. We can either add a layer to a pre-trained model and use it only *after* training, or we can add a layer to a pre-trained model and re-train the network after this layer has been added. In the latter case, the model has been trained only on filtered input, while in the former it does not. Following lexicon used in ???, we named the first technique as *reconstruction* and the second one *retraining*.
There's another alternative which is to initialize the model, add the filter layer then train the network while it's still freshly initialized. 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. Instead when it comes to reconstruction and retraining who is better is not that clear. We did some measurements to better understand it.

3.7 How did we perform the comparison

We built and train models using both reconstruction and retraining. We had now two families of models. For example, for FC-100-100-10 with a filter layer implementing PCA retaining 80 components we had one model trained using reconstruction and another trained using retraining. We attacked each model using Fast Gradient Sign obtaining a curve saying what was the adversarial success score as η increases – where η is the freedom given to the attacker: the more the freedom the easier is to forge an input. Then, we paired each model trained using reconstruction with the same model trained using retraining. We computed the average value of the curve obtained subtracting the curve for retraining from the curve for reconstruction. At last we computed the average for the set of pairs, getting what is the average difference between reconstruction and retraining.

3.8 Data obtained

We obtained that retraining is only 1% more effective than reconstruction when it comes to protect models from adversarial input.

3.9 Conclusion

While implementing retraining defense is not hard we concluded the difference is not significant enough and decided to stick with reconstruction which is a clearer solution – it allows us to think about filter techiniques without worrying about how that impacts on training.
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 little noise distributed in the image. As these filters highlight the *important* features of an image we hope they cut out the noise introduced by the attacker.
For each of the filters tested in this chapter we can set its parameters such that it can be more or less destructive in regards of the original information. 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*.
«SHOULDNT WE WRITE AN ATTACK CHAPTER?» As a baseline we performed an attack of Fast Gradient Sign with its parameter η set to 0.1. We chose that value for η as it's a median value between a non-effective attack – when η equals 0 – and an attack that's mostly unstoppable – when η is greater than 0.25.
«INSERT 3 IMAGES OF THE SAME INPUT AS ETA INCREASES»

In our *gray-box* setting we measured an accuracy on the MNIST test set of 97.39% and a probability of success for the attacker of 81%. Again, we expected these quantities to go down as we increase the *power* of the filter technique: while we try to stop the attacker we have to reduce the accuracy of the model too.

3.10 Principal Component Analysis

«WRITE GENERAL STUFF ON PCA»
We tried the following setting for PCA: retain all the components of an image (784 pixels), 331 of them, 100 of them, 20 and 10 components.

- Retaining all the 784 components: accuracy 97.38%, adversarial success score 82%.
- Retaining 331 principal components: accuracy 97.52%, adversarial success score 71%.
- Retaining 100 principal components: accuracy 97.48%, adversarial success score 44%.
- Retaining 20 principal components: accuracy 94.18%, adversarial success score 23%.
- Retaining 10 principal components: accuracy 83%, adversarial success score 22%.

Of all these setting the best result is achieved when we retained 100 components. The accuracy on the MNIST test set is pretty the same (in fact, it's even increased with the help of the filter) and the adversarial success score is halved: from the original probability of 81% to 44%. When we're comparing PCA with other image filters we'll consider this setting of 100 components.

4 Conclusions