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 of experiments described in Chapter 3. 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. While that allows parallel and distributed computation, it can be quite daunting to use when tinkering. In fact, operations to perform on data are first described and only later executed. I believe this can be counter-intuitive for most people. For example, in Figure 2.1 `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 the *computational graph* (a description of the computation to perform on data in terms of nodes and edges) which will always output 42. In TensorFlow computation is done by connecting one or more of these nodes to other nodes, letting *tensors* ("the main object you manipulate and pass around [in TensorFlow]"[2]) move 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 until you don't run the computational graph it's hard to determine what's going to be the value of a tensor, that is the *output* of the computational graph. This is slowing down exploratory analysis.

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

Figure 2.1: Building a *constant* tensor

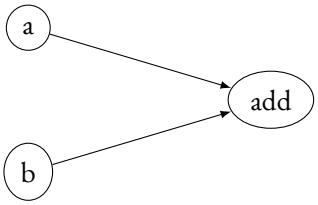


Figure 2.2: Crazy simple computational graph

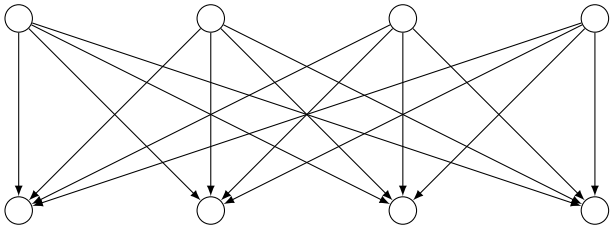


Figure 2.3: Our feedforward neural network

2.1.1 About the computational graph and TensorFlow's Python API

TensorFlow's computational graph is a fundamental part of the framework. A computational graph is a directed graph representing a computation involving tensors. A tensor in TensorFlow is a multi-dimensional array: it can be a scalar, a matrix, a batch of RGB images (which is a 4D vector[3]), etc. Each node represents an operation on tensors, while edges represent tensors.[4]
For example, 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 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 a Python API. 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 feedforward network[5] of four input neurons and four output neurons. As explained in Chapter 1 what a feedforward neural network does is a linear operation on its input and the application of an *activation function* on that result. That is, it computes

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

[1] https://www.tensorflow.org/
[2] https://web.archive.org/web/20180921135324/https://www.tensorflow.org/guide/tensors
[3] If you always thought human can't visualize four dimensions all at once, think again.
[4] This distinction between nodes and edges is important more from a formal standpoint than a practical one: actually thinking of tensors as nodes makes no difference to the best of my knowledge.
[5] https://en.wikipedia.org/wiki/Feedforward_neural_network

```
>>> 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

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 ReLU[6].
To get an idea of how to do that in TensorFlow see Figure 2.4. Note that TensorFlow already implements the activation function `tf.nn.softmax`. That function is applied to the result of the matrix multiplication. As explained in Chapter 1 this is done for various reasons: it introduces a *non-linear factor* in the network computations allowing it to learn how to compute a larger set of functions[7]; also it squashes the output of the network in $[0, 1]$ allowing to interpret it in terms of *confidence levels*.
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 uniform distribution — for example, probabilities in Figure 2.5 are totally non-sense. There's no training set and 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 session can modify. This allows you to basically add parameters to your computational graphs, allowing to iteratively change the function computed on the graph. That, combined with a learning procedure that modifies the `tf.Variables` to reduce the distance from the target function to the actual function computed by the network, make the model *learning*. In the current example, the part of the graph that you want the learning procedure to modify are W and b. As in Figure 2.6 to build a `tf.Variable` in TensorFlow you can just wrap an existing tensor.
The last thing we want to do before abandoning this example is the actual training of the network. To perform training we need a training set. As explained in Chapter 1, a training set consist of a batch of data and a label associated to each input, representing the class of that data. As *training* basically means minimizing a loss function between the provided labels of the training set and the labels guessed by the neural network, we need a loss function: as in Figure 2.7, we're using `tf.nn.softmax_cross_entropy_with_logits_v2`[8]. The gradient descent algorithm[9] is then used to iteratively minimize the cross-entropy computed against the network output. TensorFlow imple-

[6] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[7] https://en.wikipedia.org/wiki/Universal_approximation_theorem
[8] https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_error_function_and_logistic_regression
[9] https://en.wikipedia.org/wiki/Gradient_descent

```
>>> 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`

```
>>> 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: 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()) # needed.

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

```
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 2.9: A toy-example for a Keras layer

ments that algorithm via `GradientDescentOptimizer.minimize(loss)` that returns an operation that each time you run in a session will modify the graph's `tf.Variables`, in the quest of lowering the value of the loss function. We can build a loop to train the network over and over, 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.

2.2 Keras

Keras[10] 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,

[10] https://keras.io/

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

Figure 2.10: Feedforward network using Keras layers

```
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 2.11: Generating adversarial examples using CleverHans

expressing a *sequential* manipulation of the data[11]. Under the hood, a `Layer` is simply a callable object whose `__call()__` method takes a TensorFlow tensor `X` and manipulates it — see Figure 2.9.
As shown in Figure 2.10, to implement the simple feedforward neural network described in 2.1 one would stack a `Dense` layer of 4 hidden neurons and an `Activation` layer implementing the softmax function. Compare that to the TensorFlow implementation and you'll see it's much more easier to read and to think about. This nicer API that allows to immediately think about model's architecture is paid an abstraction level that at the time of this writing in my opinion is not enough stable to allow programmers to be completely oblivious of the original backend — in my case TensorFlow.

2.3 CleverHans

CleverHans[12] is a library for building adversarial examples written by Google, OpenAI and Pennsylvania State University. It implements a variety of attacks[13] against neural networks and it's compatible with models built with Keras or just plain TensorFlow. It uses the same computational graphs of TensorFlow to generate adversarial examples, so again using the library knowing something about computational graphs helps.
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

[11] https://keras.io/#getting-started-30-seconds-to-keras
[12] https://www.cleverhans.io/
[13] https://cleverhans.readthedocs.io/en/latest/source/attacks.html

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

couple of utilities to do that *model inspection* for some commonly used libraries — e.g. for Keras there's a `KerasModelWrapper`[14] — 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 2.11 for a short example.

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.
I've used scikit-learn to reuse a couple of decomposition algorithms[15] 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. Making use of all the three libraries together required a bit of thinking.
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 2.12.

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[16]. Since we had $300 of free credit we chose an expensive machine type "n1-standard-16 (16 vCPUs, 60 GB memory)", CPU platform "IntelHaswell" and as for GPUs "1 x NVIDIA Tesla P100". That made running experiments a lot faster in many cases.

[14] https://github.com/tensorflow/cleverhans/blob/66125be4e0e98686c5e005677263e52d3f3cea47/cleverhans/utils_keras.py#L101
[15] http://scikit-learn.org/0.20/modules/classes.html#module-sklearn.decomposition
[16] 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 2.13: Using `gcloud` to connect via ssh to Google remote *machine*

3 Robust networks

In this chapter I 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 3.2 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 Dynamically choosing the number of epochs

As I started to test the performance of various setups I was 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 blatlantly make a filtering 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 60% for example) while choosing a high value had the risk of been unreachable for some models.
The approach that I decided to use to somewhat solve this problem has been 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 use 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.

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

	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

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 neuron. 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* — i.e. a specified number of epochs — 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.

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 then of `fc-100-100` trained for 500 epochs reaching an accuracy of 97.55%. To check that accuracy was relatively *stable* — that is, it was a local maximum — 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. In fact, the problem with reducing the learning rate is that when you escape the plateau your model will learn slower. 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].

[2] https://keras.io/callbacks/#earlystopping

[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

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 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 a filtering technique

When testing different filtering tecniques just *finding the right one* is not enough. In fact how do you *deploy* that filtering 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 performances in order to choose which one was a better fit for our work.
We identified at least two different fashions to integrate a filtering technique in our models. We can either add a filtering layer to a model already trained on unfiltered input, or we can add a layer to a an already trained model and retrain the network for some epochs after this layer has been added. Basically in the latter case, the model has been trained only on filtered input, while in the former it does not. Inspired by lexicon used in [1], we named the first technique as *reconstruction* and the second one *retraining*.[5]
Which technique was better was not that clear, so we did some measurements to better understand it.

3.2.1 How did we perform the comparison

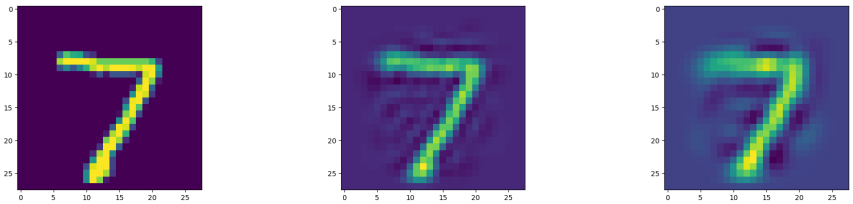
We took the same model — `fc-100-100-10` — and deployed the PCA[6] filtering technique using both reconstruction and retraining. Changing the number of components retained (see Chapter 1) we obtained dozens of models and trained all them leveraging the heuristics described in Section 3.1.
We attacked each model using the Fast Gradient Sign technique (see Chapter 1) obtaining what was the adversarial success score given a specific value of η[7].

[4] https://web.archive.org/web/20180930101331/https://datascience.stackexchange.com/questions/37163/is-it-a-good-practice-to-always-apply-reducelronplateau-given-that-models-b/37190

[5] 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.

[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.



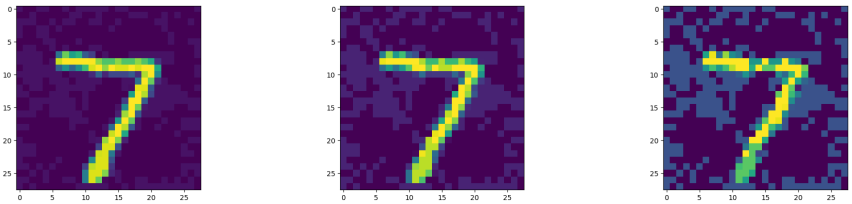
(a) 784 components (b) 100 components (c) 40 components

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

Then we compared the adversarial success score for each model trained using reconstruction to the attacker performance on the same model now trained using retraining. By subtracting these two values, making an average for each models pair and finally making an average for all the pairs, we got what was the average gain (or loss) of the model's resiliency for reconstruction over retraining. We wrote a script `bin/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 about models with filtering techniques deployed using only reconstruction.

3.3 Which filter performs 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.
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 inverse 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



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

Figure 3.3: Adversarial examples as η increases.

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 keep the feature that allow to recognize the number 7 while removing the noise malevolently 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*.

3.3.1 Attack used against models

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 kind of attack with no parameters. The choice of 0.1 has been made since it's a median value between a completely inefficial attack of η equals to 0 and an attack that's recognizible from 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 these quantities 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.

3.3.2 Choosing and fitting sklearn transformers

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

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

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

reduction advertised by the name of the class — and its inverse function. `sklearn.decomposition` implements the first for all of its classes but the related inverse function is not always available.
As shown in Figure 3.4 of 16 decomposition techniques implemented in `sklearn.decomposition` only half of them have the inverse function implemented, or available — I guess that for some of them the inverse function does not even exist, but didn't make any research 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 computing the required function: 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 3.2.

3.4 Finding something that worked 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 deployed with different filtering 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.

∼accuracy	decomposition	accuracy diff from closest PCA	adversarial score diff from closest PCA
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 other filters to PCA.

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. When using a PCA filter the adversarial score is of

4 Conclusions