Project 1 - Adversarial machine learning

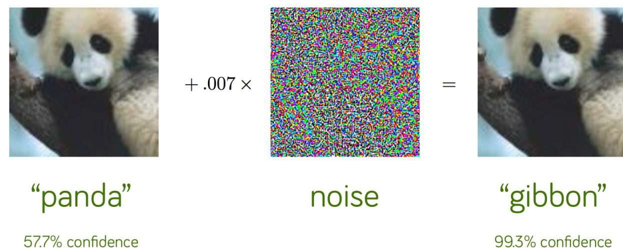
12 décembre 2019

Yoann LEMESLE, Jade GARCIA BOURREE

1 Context and aim

A classifier is a machine learning (noted ML) model that learns a mapping between inputs and a set of classes [9]. Many works have shown that those classifiers are vulnerable to adversarial integrity attacks. For example, if we have a classifier and a picture, we can create some noise (an image with same dimensions as the input) and add it to the picture in order to change the class guessed by the classifier. Furthermore, we want to target a new guessed class.

The purpose is to add a subtle noise because the attack has to be indetectable for human eye.



2 Tools

We already had a trained classifier to begin with (with 1000 classes) and a basic adversarial example that misclassified the input (a 224x224 picture).

First of all, we ran the code, played with different noises, pictures and tried to see how the classifier behaved [7].

After that we changed the attack's function to target a given class.

The code is adapted from Nicolas Papernot, and we use a few libraries :

Tensorflow is the framework we'll use to define our ML model and compute adversarial examples. It is an end-to-end open source machine learning platform that can be used by everyone.

matplolib and numpy to visualize or manipulate pictures.

3 Methods

3.1 The Fast Gradient Sign Method

This method is due to Yurii Nesterov from 1983 based on the gradient descent idea of Cauchy (1847) [3].

This is a linear perturbation of non-linear models using the gradients of the classifier to create an adversarial image. This adversarial image will be misclassified as the easier class to target with our original image (the input).

First, we used the pre-trained model to define "model_to_logits". This is the function that gives a vector of probabilities for each 1000 classes for a given input (in this example, a picture).
"model_to_logits(image_placeholer)" represents the prediction vector of our original input image. The guessed class is the index of the highest probability (we can use the dictionnary to know the name of the class).

Then, we had "labels_placeholder" wich is a one-hot vector representing the actual class of the input. A **one-hot vector** [2] is a vector with only one 1 at the lign of the guessed class and 0 for the others.

The loss function measures the probability error in discrete classification tasks (between the logits of the original image and the logits of an input) in which the classes are mutually exclusive.

The important variable is loss : we used it in "fgsm" [1].

The function "tf.gradients" is used to find a local minimum of a function. This function is "loss" and the starting point is the image_placeholder (e.g a tensor who gets the same nature as the input). The algorithm is iterative and therefore proceeds by successive improvements. At the current point, a displacement is made in the direction of the gradient, to decrease the function. The purpose is to create the noise in order to go in a class with a smaller loss logit.

After that, we took the sign of the "grad" (first return of the previous function) to know the direction to follow in order to change the image and norm it with "eps" (the maximum error accepted). We obtained the equivalent of a math vector and added it to the input to get the adversarial image. The formula of this transformation is the following one :

$$adv_x = x + \epsilon * sign(\bigtriangledown f(I_0)$$

where :
adv_x : Adversarial image.
x : Original input image.
ϵ : Multiplier to ensure the perturbations are small.
f : the Loss function.
I_0 : the starting point of the method.

We were able to run the new image in the neural network to verify the misclassification of the image and print (using matplolib) the adversarial image to verify if it still looked like the original.

If it didn't, we had to rerun the process with a smaller eps. If the classification were the same as for the input, we had to increase 'eps'.

Here's the corresponding code :

```
def fgsm(loss, image, eps):
  grad, = tf.gradients(loss, image_placeholder)
  perturbation = eps * tf.sign(grad)
  return image + perturbation

def basic_attack(eps):
  # To compute our loss, we also need to know what label the input is supposed to be
  labels_placeholder = tf.placeholder(tf.float32, (1, 1000))
  # This compute the cross-entropy loss between a model prediction and label
  loss = tf.nn.softmax_cross_entropy_with_logits_v2(labels=labels_placeholder, logits=model_to_logits(image_placeholder))
  # Next, we transform the image into an adversarial example using the Fast Gradient Sign Method (FGSM)
  adversarial_example = fgsm(loss, image_placeholder, eps)
  # We run this tensor to get the actual values corresponding to the adversarial image
  adversarial_image = sess.run(adversarial_example, feed_dict={image_placeholder: image.reshape((1, 224, 224, 3)), labels_placeholder: preds})
  return adversarial_image
```

The main body of the program is a little bit different that in the given code :

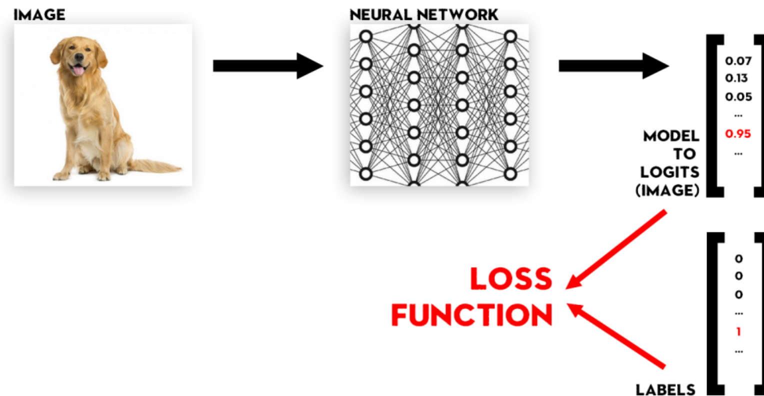
```
with tf.Session() as sess:
  # We load our pre-trained model again
  model = tf.keras.applications.MobileNetV2(weights='imagenet')
  # This time, we need to extract its logits to compute our loss (roughly speaking, how far the model predictions is from the label)
  model_to_logits = tf.keras.Model(model.input,model.layers[-1].output)
  # We run this tensor to get the actual values corresponding to the adversarial image
  adversarial_image = function(args)
  # This returns a tensor containing the symbolic model predictions
  model_preds = model(image_placeholder)
  # We run this tensor to get the actual values of the prediction on the *original* image
  preds_original_val = sess.run(model_preds, feed_dict={image_placeholder: image.reshape((1, 224, 224, 3))})
  # Finally, we pass the image through the model to get its predicctions on the adversarial example
  preds_adversarial_val = sess.run(model_preds, feed_dict={image_placeholder: adversarial_image})
```

3.2 With a targeted class

The previous method perfectly misclassified the adversarial image but we couldn't choose the new class. So the purpose was to impose a targeted class in the method. Our work is strongly inspired by this reference [4].

Instead of using the previous loss function, we created a new function adapted for the targeted class. To do that, we had to transform the digit of the class to a tensor with the function "tf.one_hot". The examples given by the TensorFlow documentation are pretty interesting to understand the operation of the function [2].

With this tensor, we were able to create the new loss function : it measures how far the class predicted of image passed as argument is to the label of the targetted class.



The other difference is that this time we wanted to substract the math vector in order to be the nearest possible to the targetted class (and maximize the loss function).

This was our picture was near the targetted class while respecting the eps (to be human impercectible).

```
def fgsm_target(image, target, eps):
  # This compute the cross-entropy loss between a target prediction and label
  loss = tf.nn.softmax_cross_entropy_with_logits_v2(labels=tf.one_hot([target],1000), logits=model_to_logits(image_placeholder))
  grad, = tf.gradients(loss, image_placeholder)
  perturbation = eps * tf.sign(grad)
  return image_placeholder - perturbation

def target_attack(target, eps):
    # We modify the image 10 times with a small step-epsilon : 0.007
    adversarial_example = fgsm_target(image_placeholder, target ,eps)
    adversarial_image = image
    for i in range(9):
      adversarial_image = sess.run(adversarial_example, feed_dict={image_placeholder: adversarial_image.reshape((1, 224, 224, 3))})
    return adversarial_image
```

4 Problems we had to face

First of all, we spent a little time to understand the given code and basic functions of Tensorflow's library.

After that we tried to do the work for a targeted class by ourselves but we had a bad comprehension of the library. For example we did not quickly understand the line "Sess.run(var, feed_dict)" and the object "placeholder". Turns out Sess.run(var, feed_dict) runs "var" (it can be a function) by replacing all the "placeholder" values with the values given in feed_dict.
It took some time for us to understand that the targeted label should be the new value of "labels" in the loss function. When we understand the changement of loss function, we spent time to try different options (and understand them) of adding or substracting the perturbation. The problem was the mental representation of the loss function.

By the way, it finally worked and we completed our goal!

5 Results

5.1 Comparison of the attacks on two examples

We compared the two attacks (basic and targetted) for two images. In reality we did it with more examples but only two are presented.

First, we compared a picture of a giant_panda with both methods.
The basic attack doesn't misclassify the picture. It is just decreasing the confidence of the classifier on it.
For the targetted attack, we can change the class as you wish with a big confidence.
For the original image, the prediction was a giant_panda with confidence 0.882.
For the basic attack, the prediction was a giant_panda with confidence 0.022.
For the targetted attack, the prediction could be a black_swan with confidence 0.999.
To wrap things up, the basic attack didn't work on all the picture. We easily found one of those. But the targetted attack was working on it.

Secondly, we compared a picture of a stop sign, because the basic attack misclassified it. For the original image, the prediction was a street_sign with confidence 0.226.
For the basic attack, the prediction was an apron with confidence 0.059.
For the targetted attack, the prediction could be an apron with confidence 0.999,
or a Tibetan_terrier with confidence 0.582.
We targeted the class given by the first method to apply the second one. In this case, it was very easy to see that the targetted method is more effective than the basic one even if we chose the same class.

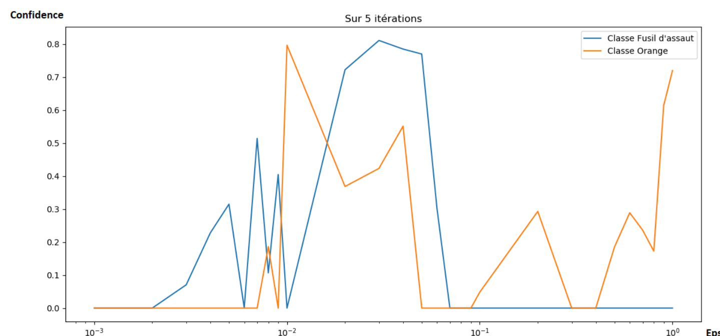
We thought that the basic attack didn't work on too confident prediction of the original image. To verify it, we did several tests and found that the misclassification doesn't work if the confidence is bigger than 0.6.

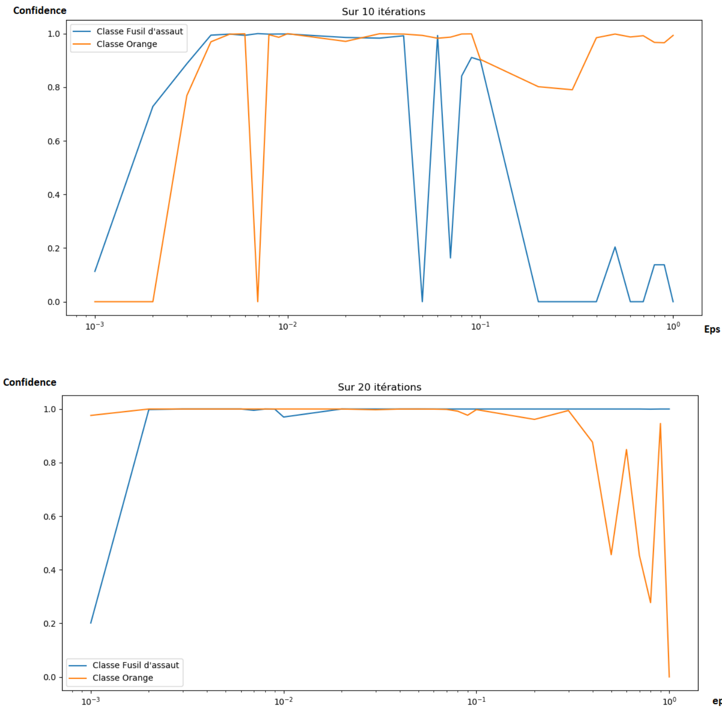
5.2 Class accessibility

At this point, we knew that for some examples the misclassification of a picture was better with the targetted attack. But we had to verify that all the classes could be targetted with the method for a given picture.
We tested it for the giant_panda picture. We "just" needed a loop in the main function that checked if the targetted class was the adversarial class. In reality it took some hours to run.
Actually, with our picture (and epsilon = 0.07) the misclassification didn't work for the classes 142 ("dowitcher", a bird) ; 178 ("Weimararer", a dog) and 278 ("kitfox"). But we knew that the original confidence was high (0.88) so it is not a surprise. The misclassification didn't work everywhere but it had just a 0.3% rate of failure. It can be considered negligible.

5.3 Choosing epsilon / iteration number

We then found interesting to study the influence of the choice of epsilon and the number of iterations for some attacks. To do that, we chose a picture of a great white shark, two targetted class (an assault rifle and an orange) and tried differents epsilon between 0 and 1 (staggered with log scale). The results are given by those graph :





We can see how a number of iterations that is too small gives bad results. From 10 iterations things get better and we can see how the ideal epsilon seems to be between 0.01 and 0.1. Above that, confidence on the adversarial image drops and the attack often fails.

6 Extensions

Because we finished the project in time, we chose to do one of the optional ideas.
But after reflexion, we prefered to spend time on doing a statistical analysis instead of doing an other work.

7 Conclusion

We're glad we have been able to make the targeted attack work. We saw trough our analysis that it is far from being perfect, but we were still able to obtain incredible results with confidence level extremely close to 100% !

This shows the necessity of creating more robust machine learning models, otherwise those kind of attack could have dramatic consequences, the most obvious example being how we don't autonomous cars to see an empty road when there is humans !

8 References

The following references are those cited in the document to which were added the two oral summary documents [8] [6] and the conference cited [5].

Références

[1] The documentation about 'aversarial_fgsm' in tensorflow. https://www.tensorflow.org/tutorials/generative/adversarial_fgsm. Accessed : 2019 - 10 -25.

[2] The documentation about 'one_hot' in tensorflow. https://www.tensorflow.org/api_docs/python/tf/one_hot. Accessed : 2019 - 10 -25.

[3] Gradient descent. https://en.wikipedia.org/wiki/Gradient_descent. Accessed : 2019 - 10 - 25.

[4] Richeer Awasthi. Breaking deep learning with adversarial examples using tensorflow. https://cv-tricks.com/how-to/breaking-deep-learning-with-adversarial-examples-using-tensorflow/.

[5] Dominique Cheveau. Intelligence artificielle et risques associés. 2019.

[6] Jonathon Shlens Christian Szegedy Ian J. Goodfellow. Explaining and harnessing adversarial examples. *ICLR*, 2015.

[7] Clémentine Maurice. Project 1 : Adversarial machine learning. https://cmaurice.fr/teaching/ENS/project1.html, 2019.

[8] D. Wagner N. Carlini. Audio adversarial examples : Targeted attacks on speech-to-text. in deep learning and security workshop. 2018.

[9] Ian Goodfellow Somesh Jha Z. Berkay Celik Ananthram Swami Nicolas Papernot, Patrick McDaniel. Practical black-box attacks against machine learning. *Asia CCS*, 2017.