# DATA SCIENCE PROJECT
## Assignment 3 - Training robust neural networks

DRISS Brahim, CHOUKARAH Ahmed, DUZENLI Mikail

December 20, 2021

# Contents

# 1 Introduction

Autonomous vehicles, video surveillance..., the rise of artificial intelligence in our daily lives is accompanied by new hacking techniques. Most of these attacks exploit the vulnerability of deep learning models to disrupt the signal to be analyzed (images, sounds), and deceive the artificial intelligences, or even guide their decisions. These techniques called adversarial attacks use imperceptible, but carefully constructed nudge in the input signal. In 2014, a group of researchers at Google and NYU found that this little perturbation is sufficient enough to fool ConvNets. This project aims to study, test and compare several adversarial attacks and defense mechanism. We will quickly introduce the theory behind these techniques before introducing our classification model and trying to make it robust against different adversarial attacks.

# 2 Description and explanation of the work

In this report, we will describe several types of attacks:

- Fast Gradient Sign Method (FGSM)
- Projected Gradient Descent (PGD)
- DeepFool attack

and as much types of defense:

- Adversarial training
- Adversarial Defense by Restricting the Hidden Space
- Noise Reduction filter with Autoencoders

For the purpose of this project, we will focus on the application of adversarial attacks on images.

## 2.1 Adversarial attacks

### 2.1.1 Fast Gradient Sign Method (FGSM)

The FGSM works by using the gradients of the neural network to create a contradictory example. For an input image, the process uses the gradients of the loss relative to the input image to create a new image that maximizes the loss. This new image is called the contradictory image. This can be summarized by the following expression:

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y)) \tag{1}$$

where :

- adv_x is the attacked image,
- $x$ and $y$ are respectively the input image and it's label,
- $\epsilon$ is the perturbation multiplier,
- $J$ is the chosen loss.

A famous image that schematize this attack is the image of a panda. The attacker adds small perturbations (distortions) to the original image, causing the model to label this image as a gibbon, with high confidence :

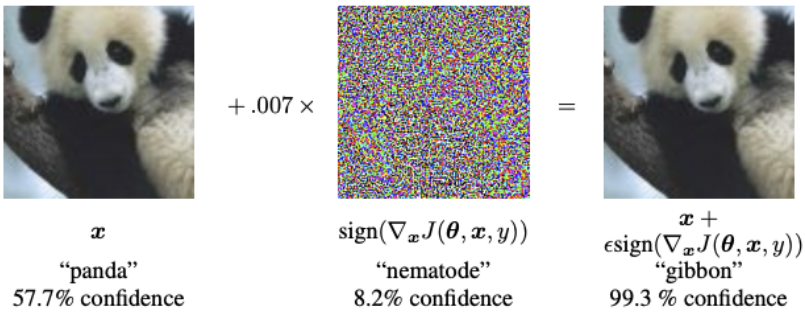



Figure 1: Image taken from Explaining and Harnessing Adversarial Examples [1]

#### 2.1.2 Projected Gradient Descent (PGD)

When FGSM is an attack designed to exploit the weakness of **linear** models, the PGD is more suited to non-linear ones. Indeed, the PGD is an iterative version of the previously introduced FGSM attack. In FGSM , the linearity of the model implies that the direction of loss is fixed. Even if we iterate it multiple times, the direction of the perturbation will not change. However, for a nonlinear model, the direction may not be completely correct after only one iteration. Thus, in PGD attack, the iterative procedure is the following :

$$adv\_x_{i+1} = \prod_{x+S} adv\_x_i + \epsilon * \text{sign}(\nabla_x J(\theta, x, y)) \tag{2}$$

where :

- $\prod$ is the operator which clips the input at the positions around the predefined perturbation range,
- $x + S$ is the perturbation set,
- $adv\_x_{i+1}$ is the $i + 1$ th iteration of the attacked image.

In both FGSM and PGD methods, one can target the misprediction to a class $\hat{y}$ by replacing $y$ by $\hat{y}$ in 1 and 2.

#### 2.1.3 DeepFool

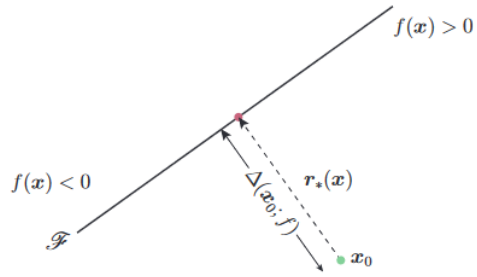
**Deepfool** is a simple and accurate method to fool deep neural networks. (Moosavi-Dezfooli et al)

If you want to find an adversarial example to an image :

- Look for the closest decision boundary
- Move the image towards a linear approximation of the decision boundary by orthogonally projecting it onto the boundary.
- Once it crosses the boundary it will be an adversarial image.

**DeepFool** is slightly different for the Multiclass Classifiers case :

- Calculate closest hyperplane from the n closest classes
- Calculate minimal projection vector
- Add perturbation and check if missclassified

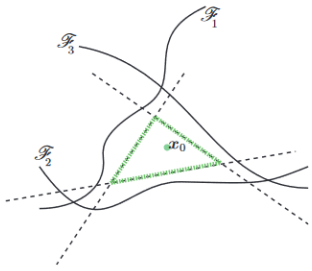



Adversarial examples for a linear binary classifier.

**Algorithm 1** DeepFool for binary classifiers

**input:** Image $\boldsymbol{x}$, classifier $f$.
**output:** Perturbation $\hat{\boldsymbol{r}}$.
Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
**while** $\text{sign}(f(\boldsymbol{x}_i)) = \text{sign}(f(\boldsymbol{x}_0))$ **do**
  $\boldsymbol{r}_i \leftarrow -\frac{f(\boldsymbol{x}_i)}{\|\nabla f(\boldsymbol{x}_i)\|_2^2} \nabla f(\boldsymbol{x}_i)$,
  $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$,
  $i \leftarrow i + 1$.
**end while**
**return** $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$.

Figure 2: DeepFool for binary classifiers




For $x_0$ belonging to class 4, let $\mathscr{F}_k = \{\boldsymbol{x} : f_k(\boldsymbol{x}) - f_4(\boldsymbol{x}) = 0\}$. The linearized zero level sets are shown in dashed lines and the boundary of the polyhedron $\tilde{P}_0$ in green.

**Algorithm 2** DeepFool: multi-class case

**input:** Image $\boldsymbol{x}$, classifier $f$.
**output:** Perturbation $\hat{\boldsymbol{r}}$.

Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
**while** $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x}_0)$ **do**
  **for** $k \neq \hat{k}(\boldsymbol{x}_0)$ **do**
    $\boldsymbol{w}'_k \leftarrow \nabla f_k(\boldsymbol{x}_i) - \nabla f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
    $f'_k \leftarrow f_k(\boldsymbol{x}_i) - f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
  **end for**
  $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x}_0)} \frac{|f'_k|}{\|\boldsymbol{w}'_k\|_2}$
  $\boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{\|\boldsymbol{w}'_{\hat{l}}\|_2^2} \boldsymbol{w}'_{\hat{l}}$
  $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
  $i \leftarrow i + 1$
**end while**
**return** $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$

Figure 3: DeepFool for Multiclass classifiers

## 2.2 Defense Mechanisms

### 2.2.1 Adversarial training

Usually, adversarial consists in training or retraining the target classification model using adversarial examples :

- The training dataset is augmented with adversarial examples produced by known types of attacks
- By adding adversarial examples $x_{adv}$ with true label y to the training set, the model will learn that $x_{adv}$ belongs to the class y

This is one of the most common adversarial defense methods currently used in practice. Another way that is also considered as adversarial training is to regularize the loss function $J(\theta, x, y)$ with the same type of attack :

$$\bar{J}(\boldsymbol{\theta}, \boldsymbol{x}, y) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, y) + (1 - \alpha) J(\boldsymbol{\theta}, \boldsymbol{x} + \delta) \tag{3}$$

where $\alpha$ is the regularization term (usually 0.5) and $\delta$ the attack (for example $\epsilon \,\text{sign}\,(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$ for FGSM).

### 2.2.2 Noise Reduction filter with Autoencoders

As we can see in the 2.1, and since the goal of the attacks is to keep the attacked image similar to the human viewer, an approach we can use is Noise reduction to try and smooth out the attack.

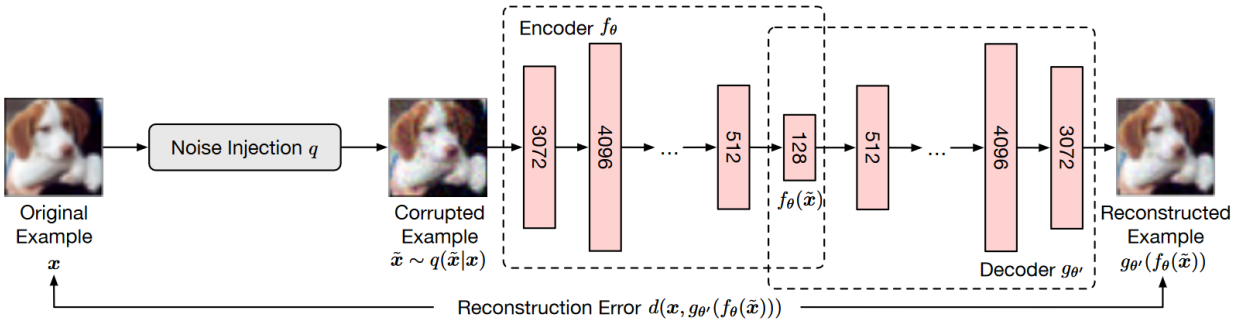



Figure 4: Noise Reduction Model

### 2.2.3 Adversarial Defense by Restricting the Hidden Space

A way to explain the efficiency of adversarial attacks is the close proximity of different class samples in the learned feature space.

Therefor, Aamir Mustafa et al.(2019) [3] suggested a custom loss that forces the features for each class to lie inside a convex polytope $\mathcal{P}_\epsilon(x;\theta)$ that is maximally separated from the polytopes of other classes :

$$\mathcal{P}_\epsilon(x;\theta) = \{\mathcal{F}_\theta(x+\delta)\ s.t.,\ \|\delta\| \leq \epsilon\} \tag{4}$$

Where $\mathcal{F}_\theta$ is a DNN with parameters $\theta$
For that purpose, we introduce a new loss :

$$\mathcal{L}_{\text{PC}}(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \left\{ \left\| \boldsymbol{f}_i - \boldsymbol{w}^c_{y_i} \right\|_2 - \frac{1}{k-1} \sum_{j \neq y_i} \left( \left\| \boldsymbol{f}_i - \boldsymbol{w}^c_j \right\|_2 + \left\| \boldsymbol{w}^c_{y_i} - \boldsymbol{w}^c_j \right\|_2 \right) \right\} \tag{5}$$

Where :

- $f_i$ are the input features,
- $w_{y_i}$ are the true class representative vectors,
- $w^C_{y_i}$ are the trainable class centroids.

This loss should minimize the distance between the hidden features $f_i$ and the centroids of there true class representative $w^C_{y_i}$ and maximize at the same time the average distance between the hidden features and the other class centroids and the distance between the true class representative centroid $w^C_{y_i}$ and the other centroids. We combine this loss to the cross-entropy loss :

$$\mathcal{L}_{\text{CE}}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{m} -\log \frac{\exp\left(\boldsymbol{w}^T_{y_i} \boldsymbol{f}_i + \boldsymbol{b}_{y_i}\right)}{\sum_{j=1}^{k} \exp\left(\boldsymbol{w}^T_j \boldsymbol{f}_i + \boldsymbol{b}_j\right)} \tag{6}$$

This combinaison enforces the intra-class compactness and an inter-class separation using learned prototypes in the output space. In order to achieve a similar effect in the intermediate feature representations, we include other auxiliary loss functions 5 along the depth of our network.

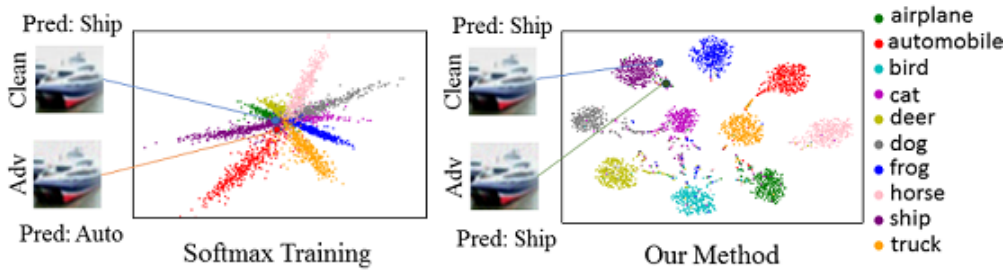



Figure 5: Visual representation of the method taken from Adversarial Defense by Restricting the Hidden Space of Deep Neural Networks [3]

# 3 Our work and results

All the work is coded in Python using Pytorch [4] library. The dataset we worked with is the CIFAR-10 collection, commonly used to train machine learning and computer vision algorithms.

## 3.1 Our model

### 3.1.1 Structure of the model

For our experiments, we used the mobilenet-v3 network [2] for multiple reasons :

- It is a widely used network with relatively good benchmarks compared to the state of the art
- It is a small network that is faster to train than complex networks such as VGG/Resnet

  We used the pytorch implementation provided here (The small version) .

### 3.1.2 Training and results on CIFAR-10

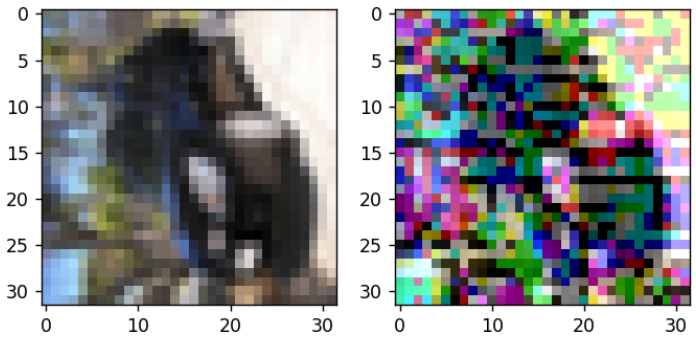
For a randomly initialized network, 200 Epochs, and a learning rate of 0.001, we achieve an accuracy of 77.73% However, (and as expected) it does do do very well versus attacks.

- Accuracy after FGSM Attack : 9.57%
- Accuracy after PGD Attack : 3.24%

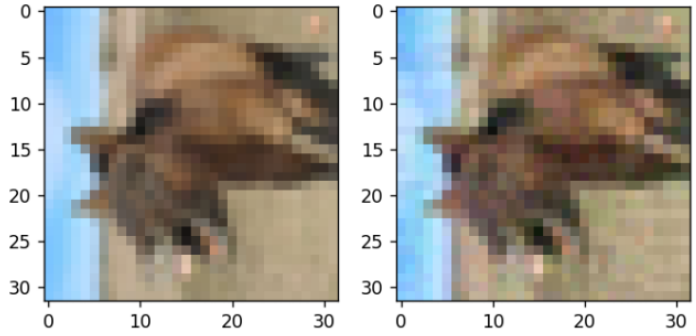
## 3.2 Implementation of the attacks

### 3.2.1 Fast Gradient Sign Method (FGSM)

The FGSM is quite easy to implement. We tried big and small $\epsilon$ to see the result of the attack on the CIFAR-10 images :



(a) Image attacked with $\epsilon = 0.25$



(b) Image attacked with $\epsilon = 0.025$

With no target, we have an average natural accuracy of 77.73 % vs 9.57 % after attack.

## 3.3 Implementation of the defense mechanisms

### 3.3.1 Adversarial training

For our adversarial trainings, we implemented the data augmentation with adversial examples. We chose to do so because it is the most common way. The following table summurazizes our results :

| **Training** | | | | | |
|---|---|---|---|---|---|
| **Natural** | | | **Adverserial** | | |
| Nat_acc | PGD_acc | FGSM_acc | Nat_acc | PGD_acc | FGSM_acc |
| 77.73 | 3.24 | 9.57 | 62.53 | 34.28 | 10.6% |

As we can see, the adversarial training worked well for the PGD attack, but not for FGSM. The reason is still to be found but the most likely explanation is an hidden bug in our code. However, after training for 100 epochs on PGD adversarial training, our model shows an average natural accuracy of 62.53% against 34.28% on adversarial examples (almose 31% rise on adversarial accuracy for a decrease of 15% of natural accuracy).

### 3.3.2 Noise Reduction filter

First of all, we train a convolutional autoencoder separately on the CIFAR10 Dataset with added gaussian noise (with a 0.1 mean). After that, we add a first layer of this autoencoder to our network.

After a few tests, we can see that the autoencoder does not get rid of the attack in the images unfortunately, therefore, we change slightly our protocol to first add a gaussian noise to the image before passing it to the autoencoder.

| Results with multiple denoising layers | | |
|---|---|---|
| Number of Noise reduction Layers | Base Accuracy | FGSM Accuracy |
| Base Model - No Layer | 77.73% | 9.57% |
| One Layer - No Noise | 70.10% | 9.66% |
| One Layer with Noise | 71.38% | 35.44% |
| Two Layers with Noise | 68.26% | 44.9% |
| Three Layers with Noise | 67.28% | 49.0% |
| Four Layers with Noise | 65.72% | 53.32% |

Table 1: Results for the Denoising Method

Not only does this method give relatively good results, but we can re-iterate it multiple times to improve the accuracy. It has also the advantage of leaving a good base accuracy on the network.

Note that as we had a few issues with loading other models in the platform, we did not activate this method for the testing (however, it is implemented).

While this method looks promising, we need to fix a multitude of hyperparameters in the autoencoder as well as the noise factor. Other interesting approaches would be training the autoencoder on more subtle noise such as the ones injected by the attacks.

#### 3.3.3 Adversarial Defense by Restricting the Hidden Space

The paper Adversarial Defense by Restricting the Hidden Space of Deep Neural Networks [3] provides a PyTorch code of the method. We had to adapt it to our model because the original code used a ResNet as a model and therefor added the auxiliary loss functions 5 to different location along the depth of the network. For our MobileNetV3, we added 2 auxiliary loss functions : one after the average pooling and one after the last dense layer of the model before the output layer.

After training for T=50 epochs on cross-entropy loss then T'=100 epochs on custom-loss combined to cross-entropy we get average natural accuracy of 62.11% vs 41.13% after attack. These numbers are not as high as expected from [3] but can be the result of too low number of epochs. The main reason is that each custom loss 5 requires two Stochastic Gradient Descent to optimize both the distances we described in section 2.2.3, which is really computationally demanding.

## 4 Conclusion

In this project, we have tried different approaches to tackle the adversarial attacks problem, and while some are more effective than others, most require a very specific learning to avoid the attacks and weakens the network as a whole.

## References


[1] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2015. arXiv: 1412.6572 [stat.ML].

[2] Andrew Howard et al. *Searching for MobileNetV3*. 2019. arXiv: 1905.02244 [cs.CV].

[3] Aamir Mustafa et al. "Adversarial Defense by Restricting the Hidden Space of Deep Neural Networks". In: Sept. 2019. DOI: 10.1109/ICCV.2019.00348.

[4] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.