Résistance aux attaques adversaires

Maximilien de Dinechin

Juin 2018

Introduction

Les exemples adversaires sont l'une des principales vulnérabilités des réseaux de neurones, en particulier des réseaux classificateurs : une perturbation imperceptible d'une entrée peut conduire à une erreur de classification. Une telle faiblesse est préoccupante, par exemple dans le cadre de la conduite autonome pour l'identification des panneaux de circulation, ou dans tout autre système où l'on souhaite confier un rôle décisionnel à des réseaux de neurones. Une simple modification d'un pixel peut parfois suffire à induire en erreur un réseau [1], et il n'y a pas pour l'instant de consensus pour résoudre ce problème : des publications proposent régulièrement des solutions, mais sont souvent mises en échec peu de temps après par des exemples adversaires plus sophistiqués.

Ce travail, qui tente de répondre à ce problème, est motivé par l'observation d'un phénomène intéressant dans le fonctionnement d'un algorithme particulier d'attaque adversaire, mettant en évidence en premier lieu une corrélation entre la difficulté à mener cette attaque sur une image donnée et la justesse de la prédiction du réseau ; puis ensuite une corrélation plus forte encore entre cette difficulté et le fait que l'image initiale soit un exemple adversaire ou non.

1 Les attaques adversaires

1.1 Les exemples adversaires

Les réseaux de neurones sont hautement vulnérables aux *exemples adversaires* [2, 3]. Un exemple adversaire est une entrée légèrement perturbée dans le but d'induire en erreur un réseau classificateur. Le plus souvent, ces perturbations sont imperceptibles pour l'œil humain.

Plus concrètement : posons Pred la fonction qui à une image associe la catégorie prédite par réseau ; et considérons une image $img \in [0,1]^n$ (c'est-à-dire à n pixels en noir et blanc), on cherche une perturbation $r \in [-1,1]^n$, de norme (le plus souvent euclidienne) minimale, telle que :

(i) $img + r \in [0,1]^n$
(ii) $\text{Pred}(img + r) \neq \text{Pred}(img)$

Remarque : Dans toute la suite, on utilisera la norme euclidienne, notée $\|\cdot\|$. Enfin, la minimalité de $\|r\|$ n'est pas toujours nécessaire : on verra par la suite que certaines perturbations restent visuellement imperceptibles alors qu'elles ont une norme élevée.

1.2 Les attaques adversaires

Une *attaque adversaire* est un algorithme qui détermine un exemple adversaire à partir d'une image donnée.

Proposons l'attaque suivante, dite "par descente de gradient itérée". Soit une image img, prédite de catégorie c par le réseau. Introduisons Conf_c la fonction qui quantifie la probabilité selon le réseau que l'image soit de catégorie c (les réseaux classificateurs considérés retournent pour chaque entrée une distribution de probabilités sur les différentes classes possibles). On cherche alors à minimiser par descente de gradient, sur r initialisé à 0^n, la fonction :

$$\text{Loss}_1(r) = \begin{cases} \|r\| & \text{si } \text{Conf}_c(img + r) \leqslant 0.2 \\ \text{Conf}_c(img + r) + \|r\| & \text{sinon.} \end{cases}$$

Cette attaque échoue presque toujours : la perturbation r reste "bloquée" en 0. On pourrait corriger ce problème en initialisant la perturbation à une faible valeur aléatoire, mais le plus simple est d'"inciter" la perturbation à grossir en norme en ajoutant un troisième cas de figure quand $\text{Conf}_c(img + r) > 0.9$:

$$\text{Loss}_2(r) = \begin{cases} \|r\| & \text{si } \text{Conf}_c(img + r) \leqslant 0.2 \\ \text{Conf}_c(img + r) + \|r\| & \text{si } 0.2 < \text{Conf}_c(img + r) \leqslant 0.9 \\ \text{Conf}_c(img + r) - \|r\| & \text{sinon.} \end{cases}$$

Remarque : les seuils à 0.2 et 0.9 ont été choisis car efficaces en pratique.

Même si elle échoue encore souvent, cette deuxième fonction se révélera suffisante pour la suite. On appellera alors Pert_N la fonction qui à une image associe la perturbation obtenue après N étapes de descente de gradient de la fonction Loss_2 avec r initialisé à 0 (algorithme `Adam` [4], avec un taux d'apprentissage $\eta = 10^{-3}$). En **Annexe A** sont présentés quelques résultats d'attaques adversaires sur des images de `MNIST`.

1.3 Réseaux classificateurs et bases de données utilisées

On réalisera toute cette étude sur deux réseaux de type `AlexNet` (CNN avec Dropout) [5], appliqués respectivement aux problèmes de la classification des images de `MNIST` [6] et de `FashionMNIST` [7] (images de chiffres manuscrits et d'habits). Ces réseaux et leur entraînement sont décrits plus précisément dans l'**Annexe B**.

On travaillera sur les sous-bases de `test` (constituées de 10000 images) de ces bases de données, afin d'étudier des images qui n'ont pas été utilisées lors de l'apprentissage. Les deux réseaux ont alors les performances suivantes :

- 62 erreurs sur les 10000 images de `test` de `MNIST`,
- 876 erreurs sur les 10000 images de `test` de `FashionMNIST`.

Ces deux bases de données présentent ainsi l'avantage d'être très proche structurellement (même format et nombre d'images), mais de difficultés très différentes dans le problème de leur classification : étudier les deux en même temps permettra de mieux analyser les résultats obtenus.

2 Résistance à une attaque

2.1 Images "faciles" et "difficiles" à attaquer

On s'intéresse aux valeurs prises par $\|r\|$ et Conf_c au cours de l'attaque adversaire décrite en 1.2. La Figure 1 a été obtenue en attaquant deux images différentes de `MNIST`.

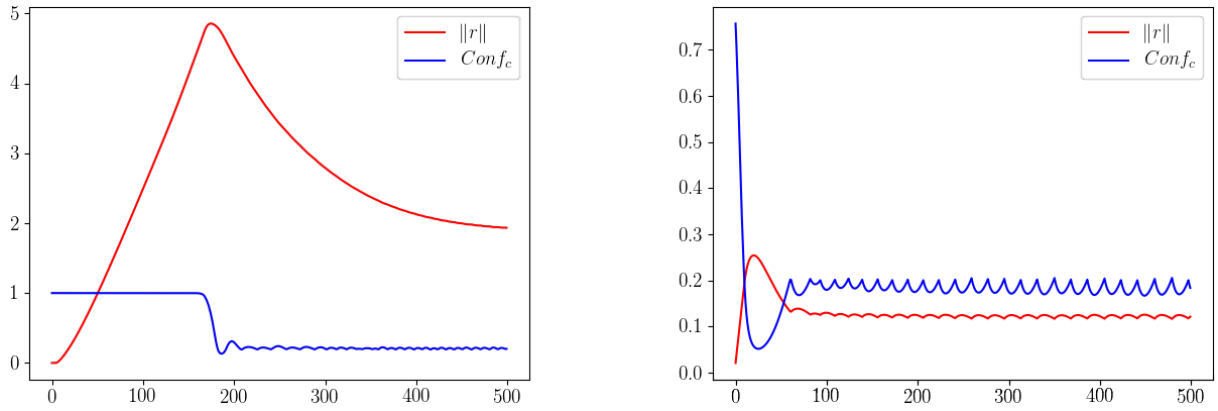


FIGURE 1 – évolution de Conf_c (en bleu) et de $\|r\|$ (en rouge) au cours d'attaques adversaires

L'image de gauche peut être qualifiée de "difficile à attaquer" : un grand nombre d'étapes ont été nécessaires pour casser la prédiction du réseau ; il a été fallu pour cela d'atteindre une norme de r très élevée ; et la norme finale de la perturbation est importante.

L'image de droite peut au contraire être qualifiée de "facile à attaquer" : il a fallu peu d'étapes pour casser la prédiction du réseau ; le pic de $\|r\|$ de très faible amplitude ; et la norme finale de r basse.

Les images de l'**Annexe C** se classifient aisément de la sorte entre "faciles" et "difficiles" à attaquer. Résumons les principales différences lors du déroulement de l'attaque entre les deux types d'images :

	Images "difficiles"	Images "faciles"
Étapes nécessaires	plus de 200	moins de 50
Pic de $\|r\|$	haut	absent ou faible
Norme de r finale	élevée	faible

Remarque : Il n'y aucune différence visuelle entre les images faciles et difficiles à attaquer.

2.2 Quantification de la résistance à une attaque

Pour quantifier plus précisément cette difficulté à attaquer une image, introduisons le concept de *résistance*. Motivés par les observations de la partie précédente, on utilisera les trois expressions suivantes : Res_N la norme de la perturbation obtenue après N étapes ; $Res_{\max}$ la plus haute norme de la perturbation au cours de l'attaque (qui correspond à la hauteur du pic) ; et $Res_{\min}$ le nombre d'étapes qu'il a fallu pour abaisser Conf_c à 0.2. On pose donc :

— $Res_N(img) = \|\text{Pert}_N(img)\|$
— $Res_{\max}(img) = \max\left\{ \|\text{Pert}_N(img)\| \text{ tel que } N \in \mathbb{N}\right\}$
— $Res_{\min}(img) = \min\left\{N \in \mathbb{N} \text{ tel que } \text{Conf}_c\big(\text{Pert}_N(img)\big) < 0.2\right\}$

Dans les cas où l'attaque échoue, on prendra systématiquement $Res = +\infty$ dans les trois expressions. Dans tout ce qui suit, on prendra $N = 500$ pour le calcul de Res_N.

Remarque : Une définition optimale de la résistance serait la norme de la perturbation minimale mettant en échec le réseau. Cette expression n'est cependant que d'un faible intérêt pratique, car incalculable.

2.3 Une corrélation avec la justesse de la prédiction

Les images attaquées dans l'**Annexe C** n'ont pas été choisies au hasard : les premières correspondent à des classifications correctes, et les suivantes à des erreurs de classification. Étudions la généralisation de ces résultats en observant la répartition des valeurs de la résistance sur des images correctement classifiées (notées `V`), et incorrectement classifiées (notées `F`) : pour `MNIST` avec 500 images dans `V` et les 62 erreurs dans `F` ; et pour `FashionMNIST` avec 500 images dans `V` et dans `F`.

`MNIST`	Res_N	$Res_{\max}$	$Res_{\min}$
90% de `V`	> 0.97	> 2.8	> 109
90% de `F`	< 0.57	< 1.3	< 58

`FashionMNIST`	Res_N	$Res_{\max}$	$Res_{\min}$
80% de `V`	> 0.28	> 0.544	> 26
80% de `F`	< 0.29	< 0.543	< 25

TABLE 1 – Répartition des résistances des images de `V` et `F`

Selon que les images sont correctement classifiées ou non, la répartition des résistances est alors très inégale : on trouve des valeurs des résistances qui discriminent de part et d'autre respectivement 90% (voire 95%) des images `V` et `F` dans le cas de `MNIST`, et tout juste 80% pour `FashionMNIST`. Une nette corrélation se dessine donc entre la résistance et la justesse de la prédiction du réseau : une résistance élevée est souvent associée à une prédiction juste, et une résistance faible à une erreur de classification.

On pourrait alors y voir une méthode pour améliorer la précision d'un réseau de neurones. Cependant, le nombre de faux positifs (images correctement classifiées mais identifiées comme des erreurs) et de faux négatifs (erreurs identifiées comme correctement classifiées) est bien supérieur à l'erreur totale commise par le réseau dans les deux cas. Une telle méthode semble ainsi peu pertinente : elle réduirait la précision du réseau. J'ai essayé d'affiner cette séparation en m'intéressant d'un seul coup à toutes les valeurs prises par Conf_c et $\|r\|$ au cours d'une attaque (c'est-à-dire les deux courbes complètes), puis en construisant un réseau de neurones "discriminateur", entraîné à faire la différence entre images correctement classifiées et incorrectement classifiées à partir de ces données, mais ici encore, le taux d'erreur est resté supérieur à celui du réseau initial.

2.4 Une méthode de détection des exemples adversaires

On observe des résultats similaires dans le cas des attaques adversaires : les exemples adversaires sont en général plus faciles à attaquer que les "vraies" images. Étudions la validité de ce résultat.

La partie 1 présente une méthode efficace de génération d'exemple adversaire. On souhaite cependant se prémunir contre les meilleures attaques existantes, et c'est pourquoi j'ai confié la génération d'exemples adversaires à la bibliothèque `Foolbox` [8] (dont j'ai dû légèrement modifier le code source pour l'adapter aux réseaux de neurones utilisés ici). On utilisera les attaques suivantes :

1 `GradientAttack` qui procède par descente de gradient sur l'image,

2 `ContrastReductionAttack` qui modifie le contraste de l'image,

3 `PointwiseAttack` qui modifie fortement un faible nombre de pixels.

On considèrera qu'un exemple adversaire est satisfaisant quand l'assurance du réseau sur la mauvaise catégorie est supérieure à 95%.

Des exemples adversaires correspondant aux trois attaques utilisées sont présentés en **Annexe D** pour `MNIST` et **Annexe E** pour `FashionMNIST`. Dans le cas de `MNIST`, les attaques `GradientAttack` et `ContrastReductionAttack` sont très difficiles à réaliser : en moyenne l'attaque ne réussit que sur deux images sur 100, rendant une collecte de données difficiles ; et `PointwiseAttack`, même si elle réussit presque à chaque fois, propose des images qui s'éloignent fortement de l'image originale, au point d'en perturber la lecture par l'œil : l'image en annexe est suffisamment déformée pour que l'on puisse la confondre avec un 8. Le cas de `FashionMNIST` ne présente pas ces difficultés, et on se concentrera donc sur cette base de données pour étudier la résistance des exemples adversaires.

Étudions donc la répartition des valeurs de la résistance sur 500 images non altérées de `FashionMNIST` (notées `V`), et sur trois lots de 200 exemples adversaires correspondant aux trois méthodes ci-dessus, notées `A1`, `A2` et `A3`.

`FashionMNIST`	Res_N	$Res_{\max}$	$Res_{\min}$
94 % de `V`	> 0.16	> 0.30	> 12
94 % de `A1`	< 0.14	< 0.26	< 8
94 % de `A2`	< 0.14	< 0.25	< 7
94 % de `A3`	< 0.13	< 0.25	< 11

TABLE 2 – Répartition des résistances des images de `V`, `A1`, `A2` et `A3`

La corrélation est cette fois-ci très forte : une valeur bien choisie à laquelle on comparerait les résistances obtenues permet de discriminer avec 96% de précision les vraies images des exemples adversaires, et ce dans les trois attaques différentes envisagées.

Cette étude n'est évidemment pas exhaustive (plusieurs paramètres peuvent jouer, en particulier la consigne de devoir tromper le réseau à 95%) mais une corrélation aussi forte est un bon présage quant à la généralisation de ces résultats.

Enfin, il est envisageable que des exemples adversaires puissent être conçus spécifiquement pour tromper cette méthode de détection. Je n'ai pas eu le temps de créer de tels exemples, mais on peut conjecturer que cette condition supplémentaire les conduira à être plus visibles à l'œil humain, donc moins bons.

Conclusion

Ce travail a ainsi permis de mettre en évidence la corrélation entre la résistance d'une image donnée et la justesse de classification du réseau, toutefois de manière trop imprécise pour y voir une méthode d'amélioration des performances d'un réseau classificateur.

En revanche, la corrélation avec le caractère "vraie image"/"exemple adversaire" d'une image est plus intéressante, car plus précise. Dans un cadre où les exemples adversaires dépassent en fréquence le nombre de faux positifs ou faux négatifs (6% pour `FashionMNIST`), cette méthode peut être efficace pour se prémunir contre les attaques adversaires.

Ce dernier résultat est ainsi encourageant, mais reste à relativiser cependant : il serait utile de mener une étude plus variée (tant dans les méthodes et seuils d'attaques adversaires, dans les architectures de réseaux et dans les bases de données étudiées) de l'efficacité de cette méthode ; et d'étudier les manières de lutter contre cette méthode de détection, avant de pouvoir conclure quant à sa robustesse.

Références

[1] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. One pixel attack for fooling deep neural networks. *arXiv preprint arXiv :1710.08864*, 2017.

[2] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv :1312.6199*, 2013.

[3] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv :1412.6572*, 2014.

[4] Diederik P Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014.

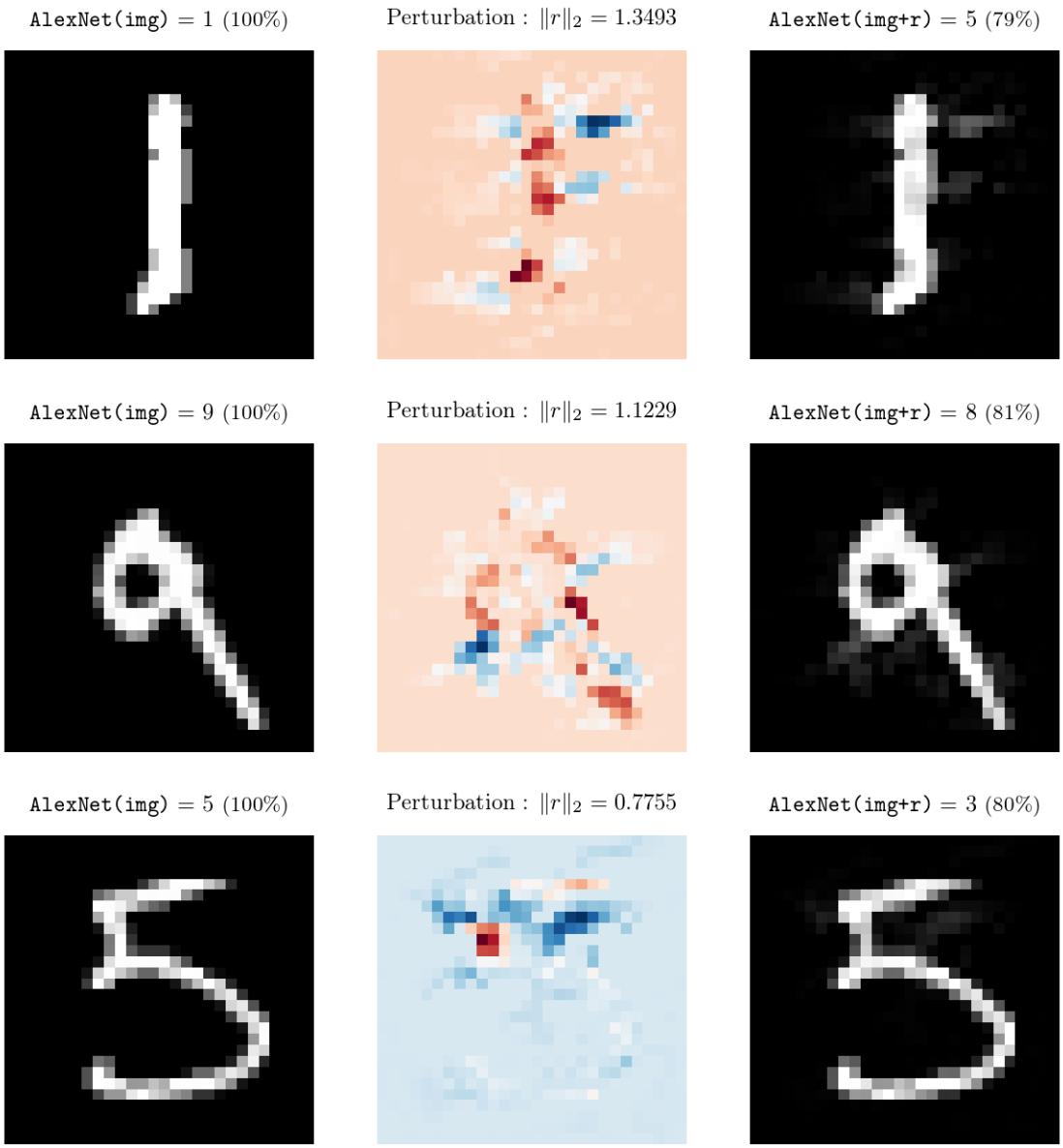
[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] Yann LeCun. The MNIST database of handwritten digits. *http ://yann. lecun. com/exdb/mnist/*, 1998.

[7] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST : a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv :1708.07747*, 2017.

[8] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox v0. 8.0 : A Python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv :1707.04131*, 2017.

Annexe A. Résultats d'attaques adversaires



(Le bleu est positif, le rouge négatif.)

Annexe B. Structure et entraînement des réseaux classificateurs

Les deux réseaux ont la même architecture (structure), inspirée de celle du réseau `AlexNet` :

- `Entrée : 28 × 28`
- `Convolution : 32 couches, noyau 5 × 5`
- `ReLU`
- `MaxPool : noyau 2 × 2`
- `Convolution : 64 couches, noyau 3 × 3`
- `ReLU`
- `MaxPool : noyau 2 × 2`
- `Dropout`
- `Couche complète : 64 × 5 × 5 → 120`
- `ReLU`
- `Dropout`
- `Couche complète : 120 → 10`
- `Softmax`

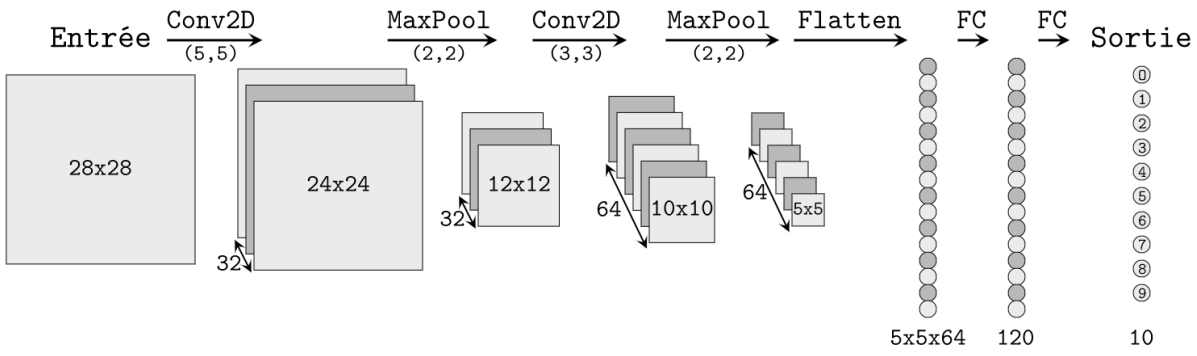


FIGURE 2 – Schéma de l'architecture utilisée

Annexe C. Évolutions de $\|r\|$ et Conf_c au cours d'attaques

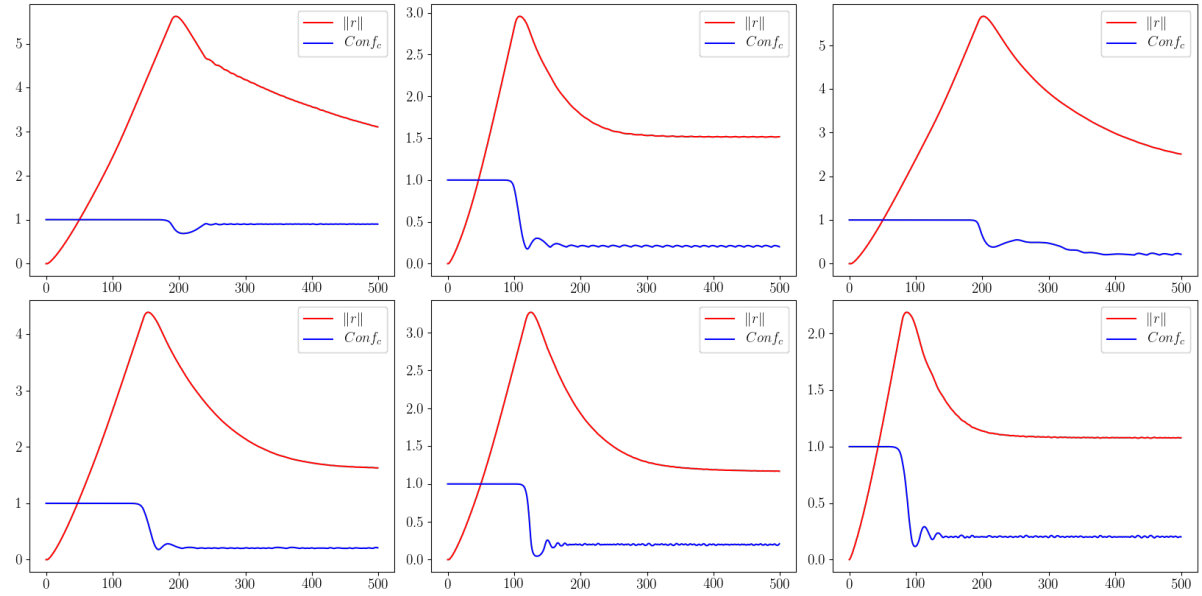


FIGURE 3 – Valeurs prises par $\|r\|$ et Conf_c au cours de l'attaque de 6 images "difficiles" à attaquer

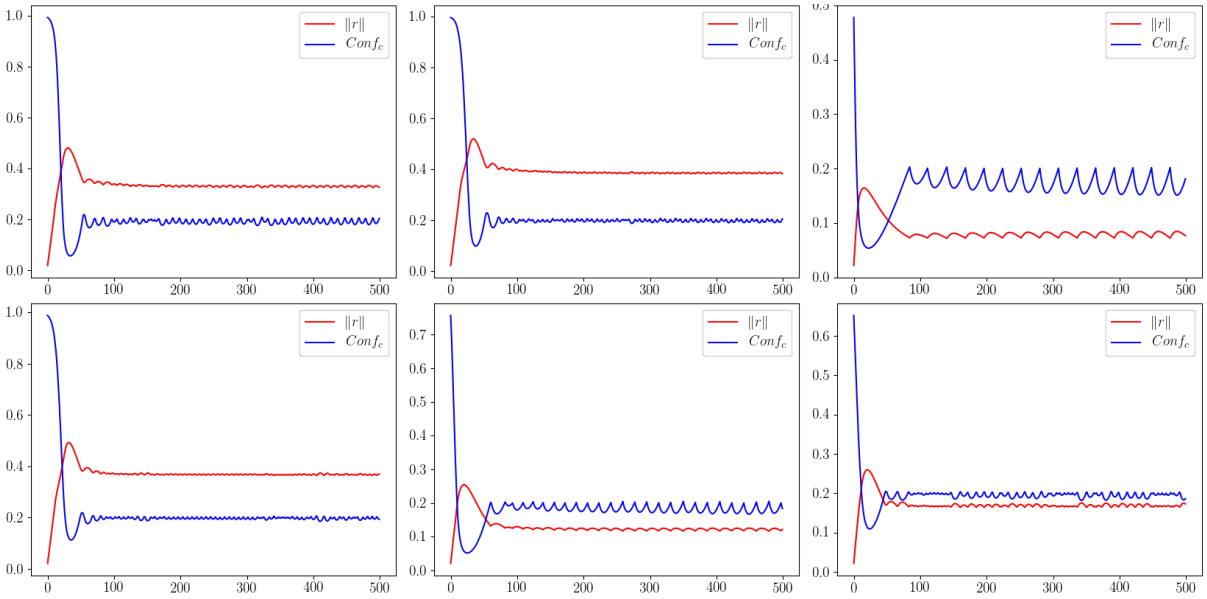


FIGURE 4 – valeurs prises par $\|r\|$ et Conf_c au cours de l'attaque de 6 images "faciles" à attaquer

Annexe D. Attaques utilisées (MNIST)

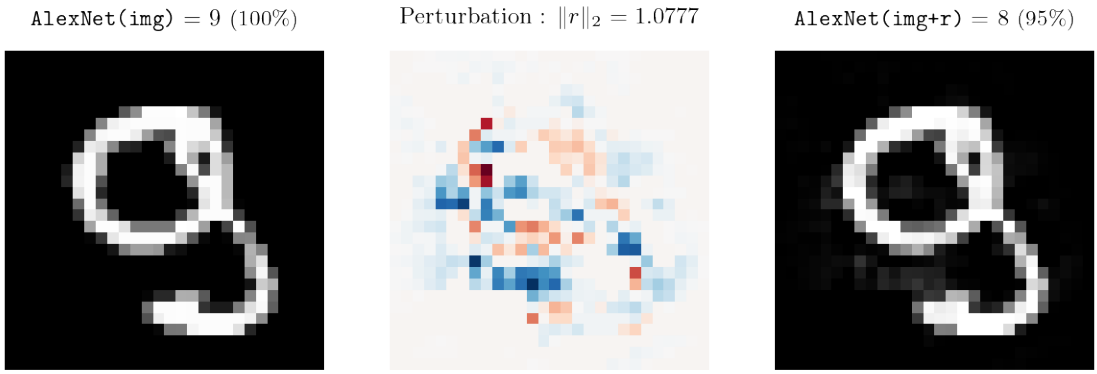


FIGURE 5 – `foolbox.attacks.GradientAttack`

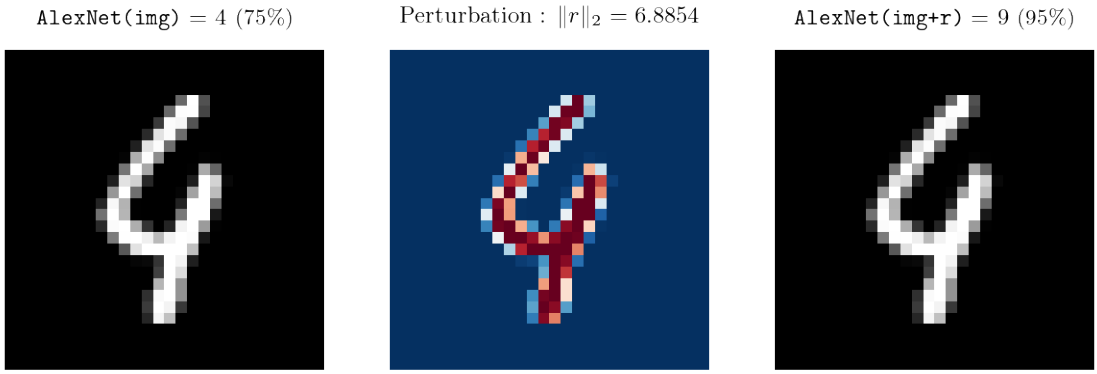


FIGURE 6 – `foolbox.attacks.ContrastReductionAttack`

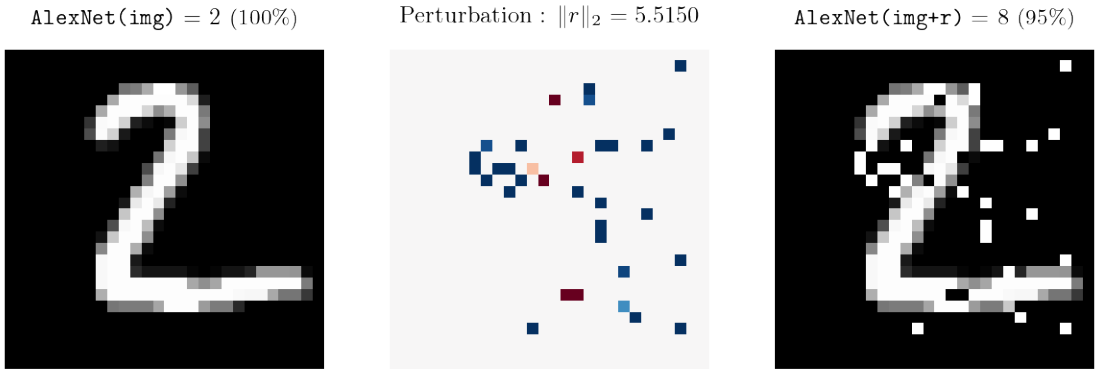


FIGURE 7 – `foolbox.attacks.PointwiseAttack`

Annexe E. Attaques utilisées (FashionMNIST)

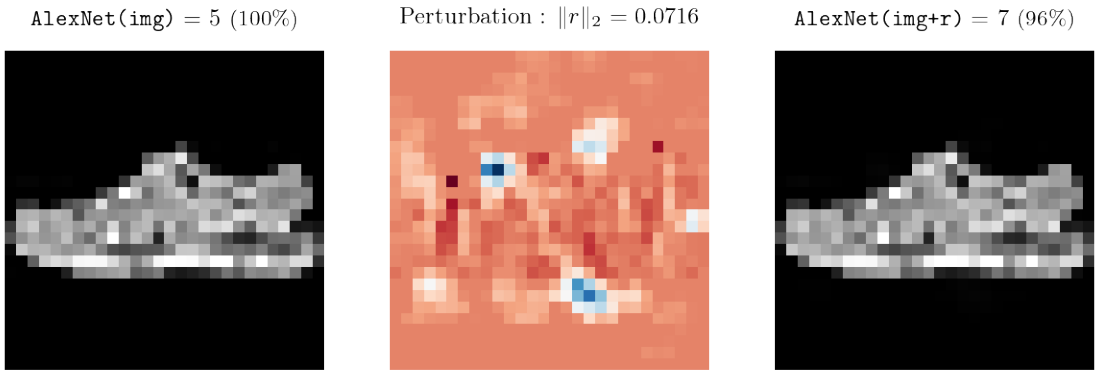


FIGURE 8 – `foolbox.attacks.GradientAttack`

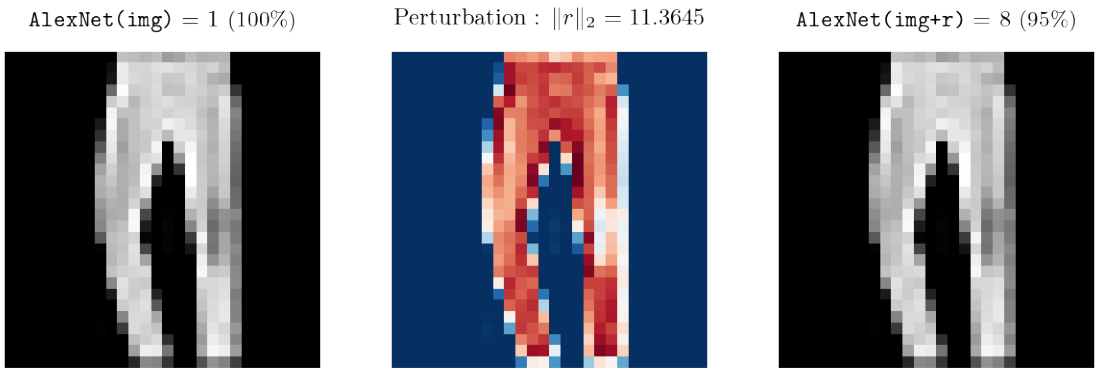


FIGURE 9 – `foolbox.attacks.ContrastReductionAttack`

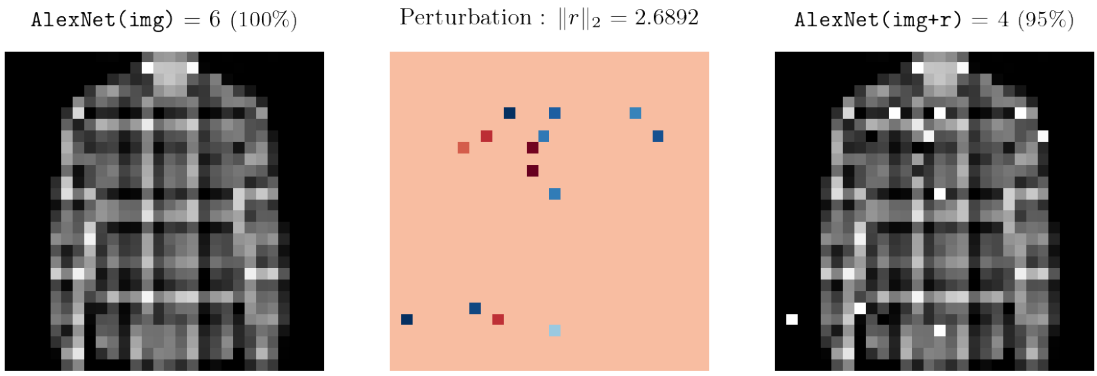


FIGURE 10 – `foolbox.attacks.PointwiseAttack`

Annexe F. Code source

Tout le code est présent à l'adresse `https://github.com/maxdinech/resistance`.

On utilise la libraire `PyTorch` (qui permet des calculs efficaces de gradients sur des tenseurs) dans sa version `0.3` (la plus récente).

Le code est divisé en six fichiers :

- `basics.py`
- `data_loader.py`
- `architectures.py`
- `train.py`
- `accuracy.py`
- `attack.py`
- `plot.py`

On utilisera les modules et bibliothèques Python suivantes :

- `torch` (PyTorch)
- `torchvision`
- `foolbox`
- `matplotlib`
- `tkinter`
- `tqdm`
- `numpy`
- `cv2`

Ainsi que `texlive` pour la trace de courbes avec des titres en LaTeX.

Ce code est fonctionnel sous Linux, et devrait normalement être compatible avec Windows (ce que je n'ai pas néanmoins pas vérifié en pratique).

a) `basics.py`

Ce fichier contient quelques fonctions utilisées dans les autres fichiers : une fonction de chargement d'une architecture (c'est-à-dire une classe qui définit une architecture d'un réseau de neurones), et une fonction de chargement de *modèle*, c'est-à-dire un réseau de neurones déjà entraîné (donc son architecture + ses poids calculés lors de l'entraînement).

basics.py

```
"""
Basic PyTorch functions used in most of the other files.

"""


import os
import warnings

import torch

```

```
import architectures


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


def load_architecture(model_name):
    model = getattr(architectures, model_name)()
    model = model.to(device)
    return model


def load_model(dataset, model_name):
    try:
        path = os.path.join("..", "models", dataset, model_name + ".pt")
        with warnings.catch_warnings():
            # Ignores the compatibility warning between pytorch updates
            from torch.serialization import SourceChangeWarning
            warnings.simplefilter('ignore', SourceChangeWarning)
            model = torch.load(path, map_location=lambda storage, loc: storage)
            model = model.to(device)
        return model
    except FileNotFoundError:
        raise ValueError('No trained model found.')
```

b) data_loader.py

Ce fichier permet de travailler simplement avec les bases de données `MNIST` et `FashionMNIST` en les téléchargeant automatiquement et en en permettant un accès facilité.

data_loader.py

```python
"""
Automatically creates and loads the MNIST and FashionMNIST datasets.
"""


import os
import shutil

import torch
from torchvision import datasets, transforms


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


# Creates `train.pt` and `test.pt` from the specified dataset.
def create(dataset):
    root = os.path.join('..', 'data')
    if dataset == 'MNIST':
        datasets.MNIST(root=root, train=True,
                       transform=transforms.ToTensor(),
                       download=True)
    elif dataset == 'FashionMNIST':
        datasets.FashionMNIST(root=root, train=True,
                              transform=transforms.ToTensor(),
                              download=True)
    os.mkdir(os.path.join(root, dataset))
    shutil.move(os.path.join(root, 'processed', 'training.pt'),
                os.path.join(root, dataset, 'train.pt'))
    shutil.move(os.path.join(root, 'processed', 'test.pt'),
                os.path.join(root, dataset, 'test.pt'))
    shutil.rmtree(os.path.join(root, 'raw'))
    shutil.rmtree(os.path.join(root, 'processed'))


# Loads a subset from a dataset.
def load(dataset, subset, num_elements=None):
    root = os.path.join('..', 'data')
    if dataset in ['MNISTnorms', 'FashionMNISTnorms',
                   'MNISTconfs', 'FashionMNISTconfs']:
        file_name = subset + '_' + dataset[-5:] + '.pt'
        path = os.path.join(root, dataset[:-5], file_name)
        values, labels = torch.load(path)
        return values, labels.long()
    elif dataset in ['MNIST', 'FashionMNIST']:
        path = os.path.join(root, dataset, subset + '.pt')
```

```
        if not os.path.exists(path):
            create(dataset)
        images, labels = torch.load(path)
        if num_elements:
            images = images[:num_elements].clone()
            labels = labels[:num_elements].clone()
        images = images.float() / 255
        labels = labels.long()
        images = images.to(device)
        labels = labels.to(device)
        images = images.view(len(images), 1, 28, 28)  # Channels first
        return images, labels
    else:
        raise ValueError('Unknown dataset')


# Loads ((1 - val_split) * num_images) from `train` starting at position 0
def train(dataset, val_split, num_images=None):
    images, labels = load(dataset, 'train', num_images)
    num_images = num_images if num_images else len(images)
    num_train = round((1-val_split) * num_images)
    train_images = images[:num_train].clone()
    train_labels = labels[:num_train].clone()
    return train_images, train_labels


# Loads (val_split * num_images) from `train` starting at position `num_train`
def val(dataset, val_split, num_images=None):
    images, labels = load(dataset, 'train', num_images)
    num_images = num_images if num_images else len(images)
    num_train = round((1-val_split) * num_images)
    train_images = images[num_train:num_images].clone()
    train_labels = labels[num_train:num_images].clone()
    return train_images, train_labels


# Loads the test images
# The unused second argument gives the same type to the three functions.
def test(dataset, _, num_test=None):
    return load(dataset, 'test', num_test)
```

c) architectures.py

Ce fichier contient les descriptions d'architectures formelles de réseaux de neurones, auxquelles on a ajouté des paramètres d'entraînement (nombre d'epochs, taux d'apprentissage, taille de mini-batch) efficaces, afin de pouvoir passer plus simplement d'une architecture à l'autre pendant la phase d'expérimentation.

architectures.py

```python
"""
Network architectures.

Default hyperparemeters (learning rate, number of epochs, batch size,
optimizer) are included in the definition of the architectures for more
flexibility.
"""


import torch
from torch import nn


class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        # Training hyperparameters
        self.lr = 5e-4
        self.epochs = 100
        self.batch_size = 64
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 64, kernel_size=3),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2)
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(64 * 5 * 5, 120),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(120, 10),
            nn.Softmax(dim=1)
        )
        # Optimizer and loss function
        self.optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x
```

```
46
47
48 class AlexNet_bn(nn.Module):
49     def __init__(self):
50         super(AlexNet_bn, self).__init__()
51         # Training hyperparameters
52         self.lr = 5e-4
53         self.epochs = 100
54         self.batch_size = 64
55         self.features = nn.Sequential(
56             nn.Conv2d(1, 32, kernel_size=5),
57             nn.ReLU(inplace=True),
58             nn.MaxPool2d(kernel_size=2),
59             nn.BatchNorm2d(32),
60             nn.Conv2d(32, 64, kernel_size=3),
61             nn.ReLU(inplace=True),
62             nn.MaxPool2d(kernel_size=2),
63             nn.BatchNorm2d(64)
64         )
65         self.classifier = nn.Sequential(
66             nn.Linear(64 * 5 * 5, 120),
67             nn.ReLU(inplace=True),
68             nn.BatchNorm1d(120),
69             nn.Linear(120, 10),
70             nn.Softmax(dim=1)
71         )
72         # Optimizer and loss function
73         self.optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
74         self.loss_fn = nn.CrossEntropyLoss()
75
76     def forward(self, x):
77         x = self.features(x)
78         x = x.view(x.size(0), -1)  # Flatten
79         x = self.classifier(x)
80         return x
81
82
83 class VGG(nn.Module):
84     def __init__(self):
85         super(VGG, self).__init__()
86         # Training hyperparameters
87         self.lr = 1e-4
88         self.epochs = 40
89         self.batch_size = 32
90         self.features = nn.Sequential(
91             nn.Conv2d(1, 64, kernel_size=3),
92             nn.ReLU(inplace=True),
93             nn.Conv2d(64, 64, kernel_size=3),
94             nn.ReLU(inplace=True),
95             nn.MaxPool2d(kernel_size=2, stride=2),
96             nn.Conv2d(64, 128, kernel_size=3),
97             nn.ReLU(inplace=True),
98             nn.Conv2d(128, 128, kernel_size=3),
```

```
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(128 * 4 * 4, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 10),
            nn.Softmax(dim=1)
        )
        # Optimizer and loss function
        self.optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x


class VGG_bn(nn.Module):
    def __init__(self):
        super(VGG_bn, self).__init__()
        # Training hyperparameters
        self.lr = 1e-4
        self.epochs = 40
        self.batch_size = 32
        self.features = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(128 * 4 * 4, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
```

```
            nn.Dropout(),
            nn.Linear(4096, 10),
            nn.Softmax(dim=1)
        )
        # Optimizer and loss function
        self.optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x
```

d) `train.py`

Ce fichier permet l'entraînement d'un réseau de neurones en gardant la plus grande souplesse possible : on passe en argument l'architecture utilisée, la base de donnée sur laquelle on travaille, le taux de validation (c'est-à-dire le pourcentage d'images utilisées pour l'entraînement, les autres servant à valider la généralisation des résultats), la taille du mini-batch, le nombre d'epochs...

train.py

```
"""
Trains a specified architucture to classify a specified dataset.
The networks are defined in architectures.py

---

usage: python3 train.py [-num NUM] [-split SPLIT] [-lr LR] [-e E] [-bs BS]
                        [-t T] [-S] model dataset

positional arguments:
  model       Network architecture (defined in architectures.py)
  dataset     Dataset used for training

optional arguments:
  -num NUM      Number of images used (default: all)
  -split SPLIT  Images proportion in val (default: 1/6)
  -lr LR        Learning rate (default: value in the model class)
  -e E          Num. of epochs (default: value in the model class)
  -bs BS        batch size (default: value in the model class)
  -k K          Top-k error metric (default: 1)
  -S, --save    Saves the trained model (default: not saved)

Note: the -split s argument is used to divide the training images into `train`
and `val`, with proportions respectively `s` and `(1-s)`.

"""


import os
import sys
import argparse

import torch
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
from tqdm import tqdm

from basics import load_architecture
import data_loader
import plot


# Parameters parsing
parser = argparse.ArgumentParser()
```

```
parser.add_argument("model", type=str,
                    help="Network architecture (defined in architectures.py)")
parser.add_argument("dataset", type=str,
                    help="Dataset used for training")
parser.add_argument("-num", type=int,
                    help="Number of images used (default: all)")
parser.add_argument("-split", type=str, default="1/6",
                    help="Images proportion in val (default: 1/6)")
parser.add_argument("-lr", type=float,
                    help="Learning rate (default: value in the model class)")
parser.add_argument("-e", type=int,
                    help="Num. of epochs (default: value in the model class)")
parser.add_argument("-bs", type=int,
                    help="batch size (default: value in the model class)")
parser.add_argument("-k", type=int, default=1,
                    help="Top-k error metric (default: 1)")
parser.add_argument("-S", "--save", action="store_true",
                    help="Saves the trained model (default: not saved)")
args = parser.parse_args()

model_name = args.model
dset_name = args.dataset
num_img = args.num
val_split = eval(args.split)  # Allows to pass fractions in parameters
k = args.k
save_model = args.save


# Model instanciation
model = load_architecture(model_name)


# Loads model hyperparameters (if not specified in args)
batch_size = args.bs if args.bs else model.batch_size
lr = args.lr if args.lr else model.lr
epochs = args.e if args.e else model.epochs


# Loads model functions
loss_fn = model.loss_fn
optimizer = model.optimizer


# Loads the training database, and splits it in into `train` and `val`.
train_images, train_labels = data_loader.train(dset_name, val_split, num_img)
val_images, val_labels = data_loader.val(dset_name, val_split, num_img)
num_train = len(train_images)
num_val = len(val_images)


# DataLoader of the `train` images
train_loader = DataLoader(TensorDataset(train_images, train_labels),
                          batch_size=batch_size,
```

```
                          shuffle=True)

num_batches = len(train_loader)


# Computes the Top-k acccuracy of the model.
# (computing the accuracy mini-batch after mini-batch avoids memory overload)
def accuracy(images, labels, k=1):
    data = TensorDataset(images, labels)
    loader = DataLoader(data, batch_size=10, shuffle=False)
    count = 0
    for (x, y) in loader:
        y_pred = model.eval()(x)
        y_pred_k = y_pred.topk(k, 1, True, True)[1]
        count += sum(sum((y_pred_k.t() == y).double()))
        # .double(): ByteTensor sums are limited at 255.
    return 100 * count / len(images)


# Computes the loss of the model.
# (computing the loss mini-batch after mini-batch avoids memory overload)
def big_loss(images, labels):
    data = TensorDataset(images, labels)
    loader = DataLoader(data, batch_size=100, shuffle=False)
    count = 0
    for (x, y) in loader:
        y_pred = model.eval()(x)
        count += len(x) * loss_fn(y_pred, y).item()
    return count / len(images)


# NETWORK TRAINING
# ----------------

# Prints the hyperparameters before the training.
print(f"Train on {num_train} samples, val on {num_val} samples.")
print(f"Epochs: {epochs}, batch size: {batch_size}")
optimizer_name = type(optimizer).__name__
print(f"Optimizer: {optimizer_name}, learning rate: {lr}")
num_parameters = sum(param.numel() for param in model.parameters())
print(f"Parameters: {num_parameters}")
print(f"Save model : {save_model}\n")


# Custom progress bar.
def bar(data, e):
    epoch = f"Epoch {e+1}/{epochs}"
    left = "{desc}: {percentage:3.0f}%"
    right = "{elapsed} - ETA:{remaining} - {rate_fmt}"
    bar_format = left + " |{bar}| " + right
    return tqdm(data, desc=epoch, ncols=74, unit='b', bar_format=bar_format)


```

```python
train_accs, val_accs = [], []
train_losses, val_losses = [], []

try:
    # Main loop over each epoch
    for e in range(epochs):

        # Secondary loop over each mini-batch
        for (x, y) in bar(train_loader, e):

            # Computes the network output
            y_pred = model.train()(x)
            loss = loss_fn(y_pred, y)

            # Optimizer step
            model.zero_grad()
            loss.backward()
            optimizer.step()

        # Calculates accuracy and loss on the train database.
        train_acc = accuracy(train_images, train_labels, k)
        train_loss = big_loss(train_images, train_labels)
        train_accs.append(train_acc)
        train_losses.append(train_loss)

        # Calculates accuracy and loss on the validation database.
        val_acc = accuracy(val_images, val_labels, k)
        val_loss = big_loss(val_images, val_labels)
        val_accs.append(val_acc)
        val_losses.append(val_loss)

        # Prints the losses and accs at the end of each epoch.
        print(f" |-> train_acc@{k}: {train_acc:5.2f}%",
              f" -- train_loss: {train_loss:6.4f}")
        print(f" |->   val_acc@{k}: {val_acc:5.2f}%",
              f" --   val_loss: {val_loss:6.4f}\n")

# Allows to manually interrupt the training (early stopping).
except KeyboardInterrupt:
    pass

# Saves the network if stated.
if save_model:
    path = os.path.join("..", "models", dset_name, model_name + ".pt")
    torch.save(model, path)
    # Saves the accs history graph
    plot.train_history(train_accs, val_accs)
    plt.savefig(path + model_name + ".png", transparent=True)
```

e) accuracy.py

Ce fichier, qui reprend la même syntaxe de paramètres que le précédent, sert à évaluer la performance d'un modèle (réseau entraîné).

accuracy.py

```
"""
Computes the Top-k error of a trained model, over a given subset of a dataset.

---

usage: python3 accuracy.py [-k K] model dataset subset

positional arguments:
  model       Trained model to evaluate
  dataset     Dataset used for training
  subset      Subset to calculate the error on

optional arguments:
  -split SPLIT  Images proportion in val (default: 1/6)
  -k K          Top-k error metric (default: 1)

Note: to compute the accuracy on `train` or `val`, the -split argument is
needed in order to determine the right repartition of all training images into
`train` and `val`.  To compute the accuracy over all training images at once,
use:
        python3 accuracy.py [-k K] model dataset train -split 0

"""


import sys
import argparse

import torch
from torch.utils.data import DataLoader, TensorDataset
from tqdm import tqdm

from basics import load_model
import data_loader


# Parameters parsing
parser = argparse.ArgumentParser()
parser.add_argument("model", type=str, help="Trained model to evaluate")
parser.add_argument("dataset", type=str, help="Dataset used for training")
parser.add_argument("subset", type=str, help="Subset to compute the error on")
parser.add_argument("-split", type=str, default="1/6",
                    help="Images proportion in val (default: 1/6)")
parser.add_argument("-k", type=int, default=1,
                    help="Top-k error metric (default: 1)")
args = parser.parse_args()
```

```
47
48 model_name = args.model
49 dset_name = args.dataset
50 subset = args.subset
51 val_split = eval(args.split)  # Allows to pass fractions in parameters
52 k = args.k
53
54
55 # Loads the model
56 model = load_model(dset_name, model_name)
57
58
59 # Loads the specified subset from the dataset.
60 images, labels = getattr(data_loader, subset)(dset_name, val_split)
61
62
63 # Custom progress bar.
64 def bar(data):
65     bar_format = "{percentage:3.0f}% |{bar}| {elapsed} - ETA:{remaining}"
66     return tqdm(data, ncols=74, bar_format=bar_format)
67
68
69 # Computes the Top-k acccuracy of the model.
70 # (computing the accuracy mini-batch after mini-batch avoids memory overload)
71 def accuracy(images, labels, k=1):
72     data = TensorDataset(images, labels)
73     loader = DataLoader(data, batch_size=100, shuffle=False)
74     count = 0
75     position = 0
76     for (x, y) in bar(loader):
77         position += len(x)
78         y_pred = model.eval()(x)
79         y_pred_k = y_pred.topk(k, 1, True, True)[1]
80         count += sum(sum((y_pred_k.t() == y).double()))
81         # .double(): ByteTensor sums are limited at 255.
82     return 100 * count / len(images)
83
84
85 # Prints the losses and accuracies at the end of each epoch.
86 print(f"Computing the Top-{k} error on the {len(images)} {subset} images...")
87 acc = accuracy(images, labels, k)
88 error = 100 - acc
89 print(f"Top-{k} error: {error:0.2f}%")
```

f) `attack.py`

Ce fichier sert à exécuter les attaques adversaires par descente de gradient itérée, et calculer la résistance d'une image ; ainsi que les attaques adversaires de la bibliothèque `Foolbox`.

attack.py

```
"""
Adversarial attacks

---

usage: python3 -i attack.py model dataset

positional arguments:
  model       Trained model to evaluate
  dataset     Dataset used for training

"""


import os
import sys
import shutil
import argparse

import torch
from torch import nn
import matplotlib.pyplot as plt
import foolbox
import numpy as np
import cv2

from basics import load_model
import data_loader
import plot


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


# Parameters parsing
parser = argparse.ArgumentParser()
parser.add_argument("model", type=str, help="Trained model to evaluate")
parser.add_argument("dataset", type=str, help="Dataset used for training")
args = parser.parse_args()

model_name = args.model
dataset_name = args.dataset


# Loads the model
model = load_model(dataset_name, model_name).to(device)
```

```
47
48
49 # Loads the specified subset from the specified database
50 images, labels = data_loader.test(dataset_name, None)
51
52
53 # BASIC FUNCTIONS
54 # ---------------
55
56 # Loads an image from the webcam
57 def webcam():
58     cap = cv2.VideoCapture(0)
59     ret, frame = cap.read()
60     cap.release()
61     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
62     x, y = len(gray[0]), len(gray)
63     d = int((x-y) / 2)
64     gray = np.array([line[d:y-d] for line in gray])
65     gray = cv2.resize(gray, (28, 28)) / 255
66     return torch.Tensor(gray).view(1, 1, 28, 28).to(device)
67
68
69 # Loads the #img_id image from the test database (torch.Tensor).
70 def load_image(img_id):
71     return images[img_id].view(1, 1, 28, 28).to(device)
72
73
74 # Loads the #img_id label from the test database (int).
75 def load_label(img_id):
76     return labels[img_id].item()
77
78
79 # Returns the label prediction of an image (float).
80 def prediction(image):
81     return model.eval()(image).max(1)[1].item()
82
83
84 # Returns the predictions of of the network on an image (ndarray).
85 def predictions(image):
86     return model.eval()(image).data[0].numpy()
87
88
89 # Returns the confidence of the network that the image is `digit`.
90 def confidence(image, category):
91     return model.eval()(image)[0, category].item()
92
93
94 # Yields the indices of the first n wrong predictions.
95 def errors(n=len(images)):
96     i = 0
97     l = len(images)
98     while i < l and n > 0:
99         image, label = load_image(i), load_label(i)
```

```
        if prediction(image) != label:
            yield i
            n -= 1
        i += 1


# Yields the indices of the first n correct predictions.
def not_errors(n=len(images)):
    i = 0
    l = len(images)
    while i < l and n > 0:
        image, label = load_image(i), load_label(i)
        if prediction(image) == label:
            yield i
            n -= 1
        i += 1


# ATTACK FUNCTIONS
# ----------------

class Attacker(nn.Module):
    def __init__(self, p, lr):
        super(Attacker, self).__init__()
        self.p = p
        self.r = nn.Parameter(torch.zeros(1, 1, 28, 28))
        self.optimizer = torch.optim.Adam(self.parameters(), lr=lr)

    def forward(self, x):
        return (x + self.r).clamp(0, 1)

    def loss_fn(self, image, digit):
        adv = self.forward(image)
        conf = model(adv)[0, digit]
        norm = (adv - image).abs().pow(self.p).sum()
        if conf < 0.2:
            return norm
        elif conf < 0.9:
            return conf + norm
        else:
            return conf - norm


def GDA(image, steps=500, p=2, lr=1e-3):
    norms, confs = [], []
    digit = prediction(image)
    attacker = Attacker(p, lr)
    attacker = attacker.to(device)
    optim = attacker.optimizer
    for i in range(steps):
        # Training step
        loss = attacker.loss_fn(image, digit)
        attacker.zero_grad()
```

```
        loss.backward()
        optim.step()
        # Prints results
        adv = attacker.forward(image)
        conf = confidence(adv, digit)
        norm = (adv - image).norm(p).item()
        print(f"Step {i:4} -- conf: {conf:0.4f}, L_{p}(r): {norm:0.20f}",
              end='\r')
        norms.append(norm)
        confs.append(conf)
    print()
    success = prediction(adv) != digit
    return(success, adv, norms, confs)


def GDA_break(image, max_steps=500, p=2, lr=1e-3):
    norms, confs = [], []
    digit = prediction(image)
    attacker = Attacker(p, lr)
    attacker = attacker.to(device)
    optim = attacker.optimizer
    adv = attacker.forward(image)
    steps = 0
    while confidence(adv, digit) >= 0.2 and steps < max_steps:
        steps += 1
        # Training step
        loss = attacker.loss_fn(image, digit)
        attacker.zero_grad()
        loss.backward()
        optim.step()
        # Prints results
        adv = attacker.forward(image)
        conf = confidence(adv, digit)
        norm = (adv - image).norm(p).item()
        print(f"Step {steps:4} -- conf: {conf:0.4f}, L_{p}(r): {norm:0.10f}",
              end='\r')
        norms.append(norm)
        confs.append(conf)
    print()
    return(steps, adv, norms, confs)


def GDA_graph(img, steps=500, p=2, lr=1e-3):
    success, adv, norms, confs = GDA(img, steps, p, lr)
    plot.attack_history(norms, confs)
    plt.show()
    path = os.path.join("..", "results", "last_attack_his±±±ry.png")
    plt.savefig(path, transparent=True)
    confc = lambda i: confidence(i, prediction(img))
    if success:
        print("\nAttack suceeded")
        img_pred = prediction(img)
        img_conf = confidence(img, img_pred)
```

```
        adv_pred = prediction(adv)
        adv_conf = confidence(adv, adv_pred)
        plot.attack_result(model_name, p,
                           img, img_pred, img_conf,
                           adv, adv_pred, adv_conf)
        path = os.path.join("..", "results", "last_attack_result.png")
        plt.savefig(path, transparent=True)
        plt.show()
    else:
        print("\nAttack failed")


def GDA_break_graph(img, max_steps=500, p=2, lr=1e-3):
    success, adv, norms, confs = GDA_break(img, max_steps, p, lr)
    plot.attack_history(norms, confs)
    path = os.path.join("..", "results", "last_attack_history.png")
    plt.savefig(path, transparent=True)
    plt.show()
    img_pred = prediction(img)
    img_conf = confidence(img, img_pred)
    adv_pred = prediction(adv)
    adv_conf = confidence(adv, adv_pred)
    plot.attack_result(model_name, p,
                       img, img_pred, img_conf,
                       adv, adv_pred, adv_conf)
    path = os.path.join("..", "results", "last_attack_result.png")
    plt.savefig(path, transparent=True)
    plt.show()


# RESISTANCE FUNCTIONS
# --------------------

# A resistance value greater than 1000 is not possible.
# Which is why 10000 will represent infinity when the attack fails.

def resistance_N(image, steps=500):
    success, _, norms, _ = GDA(image, steps)
    if success:
        return norms[-1]
    return 10000


def resistance_max(image, steps=500):
    success, _, norms, _ = GDA(image, steps)
    if success:
        return max(norms)
    return 10000


def resistance_min(image, max_steps=500):
    steps = attack_break(image, max_steps)[0]
    if steps < max_steps:
```

```
        return steps
    return 10000


# Computes the N-resistance, max_resistance and min_resistance in a single pass
def resistances(image, steps=500):
    success, _, norms, confs = GDA(image, steps)
    if success:
        res_N = norms[-1]
        res_max = max(norms)
        res_min = 1 + next((i for i, c in enumerate(confs) if c <= 0.2), steps)
        return (res_N, res_max, res_min)
    else:
        return (10000, 10000, 10000)


# Computes the N-resistance, max_resistance and min_resistance of an image list
def resistances_list(images_list, steps=500):
    L_res_N, L_res_max, L_res_min = [], [], []
    i, l = 1, len(images_list)
    for image in images_list:
        print(f"{i}/{l} : ")
        res_N, res_max, res_min = resistances(image, steps)
        L_res_N += [res_N]
        L_res_max += [res_max]
        L_res_min += [res_min]
        i += 1
    return (L_res_N, L_res_max, L_res_min)


# FOOLBOX ATTACKS
# ---------------


fmodel = foolbox.models.PyTorchModel(model, (0, 1), num_classes=10,
                                     channel_axis=1,
                                     cuda=torch.cuda.is_available())


# Runs the specified attack on an image, ensurint that the confidence of the
# network's prediction on the adversarial example is above `p`.
def foolbox_attack(img, attack_name, p=0.95):
    path = f"../results/{dataset_name}/{attack_name}/"
    if not os.path.exists(path):
        os.mkdir(path)
    img_pred = prediction(img)
    img_conf = confidence(img, img_pred)
    try:
        # Finds an adversarial example
        attack = getattr(foolbox.attacks, attack_name)(fmodel)
        np_adv = attack(np.array(img).reshape(1, 28, 28), img_pred)
        adv = torch.Tensor(np_adv).view(1, 1, 28, 28).to(device)
        adv_pred = prediction(adv)
```

```
312         # Increases its classification probability above p
313         criterion = foolbox.criteria.TargetClassProbability(adv_pred, p)
314         attack = getattr(foolbox.attacks, attack_name)(fmodel, criterion)
315         np_adv = attack(np.array(img).reshape(1, 28, 28), img_pred)
316         adv = torch.Tensor(np_adv).view(1, 1, 28, 28).to(device)
317         adv_pred = prediction(adv)
318         adv_conf = confidence(adv, adv_pred)
319         plot.attack_result(model_name, 2,
320                            img, img_pred, img_conf,
321                            adv, adv_pred, adv_conf)
322         plt.show()
323         shutil.move("../results/latest/attack_result.png",
324                     path + f"{img_id:04d}.png")
325         torch.save(adv, path + f"{img_id:04d}.pt")
326         return adv
327     except KeyboardInterrupt:
328         break
329     except:
330         pass
331
332
333 def foolbox_attacks_list(size, attack_name, p=0.95):
334     adv_list = []
335     for img_id in not_errors():
336         if len(adv_list) >= size or img_id >= 10000:
337             break
338         print(len(adv_list), "/", img_id, end='\r')
339         img = load_image(img_id)
340         adv = foolbox_attack(img, attack_name, p)
341         if adv is not None:
342             adv_list.append(adv)
343     return adv_list
```

g) plot.py

Ce fichier permet de tracer les différentes courbes de ce document.

plot.py

```python
"""
Plotting functions used in attack.py
"""


import matplotlib.pyplot as plt
from matplotlib import rcParams
from tqdm import tqdm


rcParams['text.usetex'] = True
rcParams['text.latex.unicode'] = True
rcParams['font.family'] = "serif"
rcParams['font.serif'] = "cm"


# Plots an image (Variable)
def plot_image(image):
    plt.imshow(image.data.view(28, 28).numpy(), cmap='gray')


# Plots and saves the comparison graph of an adversarial image
def attack_result(model_name, p,
                  img, img_pred, img_conf,
                  adv, adv_pred, adv_conf):
    model_name = model_name.replace('_', '\\_')  # Escapes '_' characters
    r = (adv - img)
    norm = r.norm(p)
    # Matplotlib settings
    rcParams['axes.titlepad'] = 10
    rcParams['font.size'] = 8
    fig = plt.figure(figsize=(7, 2.5), dpi=180)
    # Image
    ax1 = fig.add_subplot(1, 3, 1)
    ax1.imshow(img.data.view(28, 28).cpu().numpy(), cmap='gray')
    plt.title(f"\\texttt{{{model_name}(img)}} = {img_pred} \\small{{({100*img_conf:0.0f}\\%)}}")
    plt.axis('off')
    # Perturbation
    ax2 = fig.add_subplot(1, 3, 2)
    ax2.imshow(r.data.view(28, 28).cpu().numpy(), cmap='RdBu')
    plt.title(f"Perturbation : $\\Vert r \\Vert_{{{p}}} = {norm:0.4f}$")
    plt.axis('off')
    # Adversarial image
    ax3 = fig.add_subplot(1, 3, 3)
    ax3.imshow(adv.data.view(28, 28).cpu().numpy(), cmap='gray')
    plt.title(f"\\texttt{{{model_name}(img+r)}} = {adv_pred} \\small{{({100*adv_conf:0.0f}\\%)}}")
    plt.axis('off')
```

```
    # Save and plot
    fig.tight_layout(pad=1)
    plt.subplots_adjust(left=0.05, right=0.95, top=0.80, bottom=0.05)
    plt.savefig("../results/latest/attack_result.png", transparent=True)


# Plots the history of a model training
def train_history(train_accs, val_accs):
    rcParams['font.size'] = 12
    t = list(range(len(train_accs)))
    plt.plot(t, train_accs, 'r')
    plt.plot(t, val_accs, 'b')
    plt.title("Network training history")
    plt.legend(["train accuracy", "val accuracy"])


# Plots the history of an attack
def attack_history(norms, confs):
    rcParams['font.size'] = 14
    t = list(range(len(norms)))
    plt.plot(t, norms, 'r')
    plt.plot(t, confs, 'b')
    plt.legend(["$\\Vert r \\Vert$", "$\\mathrm{Conf}_c$"])
    plt.savefig("../results/latest/attack_history.png", transparent=True)
```