# Attaques adversariales

État de l'art

Cindy Hartmann et Antoine Montier

Novembre-Décembre 2025



# 1. Contexte

Cette étude est réalisée dans le cadre du projet de mi-année du *Master Cybersécurité* (*Université de Caen Normandie* et *ENSICAEN*). Elle est encadrée par M. Christophe Charrier, chercheur au laboratoire du *GREYC* ainsi que M. Emmanuel Giguet, chercheur au *CNRS*.

Réalisée en binôme, l'étude vise à comprendre et à mettre en application les principales attaques adversariales. Ce travail fait partie d'un sujet plus large, qui est la détection améliorée de *DeepFakes* et de *FaceSwapping* malgré des tentatives de camouflage par attaque adversariale.

Au cours des dernières années, les technologies de génération fake news, notamment à travers le face swapping et les deepfakes, ont connu une évolution rapide et spectaculaire grâce aux modèles génératifs tels que les GANs (Generative Adversarial Networks) et, plus récemment, les modèles de diffusion. Ces techniques permettent de produire des vidéos hyperréalistes où l'identité d'une personne est remplacée par celle d'une autre, avec un niveau de fidélité visuelle tel qu'il devient difficile, voire impossible, de distinguer le réel du faux à l'oeil nu.

Face à cette menace, la communauté scientifique a développé de nombreux systèmes de détection automatique de deepfakes, exploitant des architectures de vision profonde (CNN, Vision Transformers, ou modèles spatio-temporels). Ces détecteurs apprennent à identifier les artefacts subtils produits par les modèles génératifs tels que les déformations locales du visage, incohérences d'éclairage, traces de compression, ou ruptures temporelles. Cependant, ces systèmes restent vulnérables à une nouvelle génération de menaces : les attaques adversariales.

Ce projet s'inscrit dans ce contexte de cybersécurité appliquée à l'IA visuelle, et vise à évaluer la robustesse des détecteurs de face swapping face à des attaques adversariales. Il comprendra ainsi une mise en application des principales attaques adversariales.

## 1.1 Définitions

1. **Deepfake** : Un *deepfake* est un contenu (image ou vidéo par exemple) faux créé en utilisant des techniques d'apprentissage profond pour modifier un contenu réel. Son nom vient de deep (profond), cela fait référence aux algorithmes d'apprentissage profond utilisés et de fake (faux), qui est lié au fait que le contenu généré est faux (Weiss, 2019).

   Il a été popularisé sur le forum Reddit, par un utilisateur sous le pseudonyme u/deepfakes, qui a publié des vidéos truquées où le visage de célébrités était remplacé par celui d'autres personnes à l'aide de réseaux de neurones génératifs (GANs).

2. **Face swapping** : Le *face swapping* (échange de visages) est un type de deepfake qui vise à remplacer le visage d'une personne par celui d'une autre dans une vidéo ou une photo (Xu et al., 2022).

   Ce terme est antérieur au deepfake et remonte au début des années 2010 bien avant la démocratisation du deep learning . Il est l'une des principales applications techniques qui ont donné naissance au phénomène des deepfakes.

   Note : dans ce rapport, nous utiliserons principalement le mot deepfake, qui englobe le face swapping.

## 1.2 Attaque Adversariale

### 1.2.1 Intuition

L'objectif de ces attaques est de modifier une image de manière indétectable pour l'œil humain dans le but de tromper un modèle et obtenir une prédiction différente de celle attendue. La plupart de ces attaques cherchent à tromper de manière la plus efficace le modèle tout en restant le plus discret pour l'œil humain.

### 1.2.2 Qui utilise ces attaques ?

Le profil des attaquants est très varié. En effet, cela va de la simple personne soucieuse de sa vie privée qui pourrait utiliser du maquillage pour contourner un système de reconnaissance faciale au hacker qui chercherait à accéder à un bâtiment protégé par un système biométrique. Les créateurs de deepfake cherchent à rendre leur création indétectable par des modèles dont le but est de filtrer un contenu en fonction de sa véracité.

### 1.2.3 Qu'est-ce qui est attaqué ?

Les cibles sont multiples. On peut attaquer directement le modèle de reconnaissance présent dans les systèmes d'authentification ou d'identification. Il est aussi possible d'attaquer le modèle en empoisonnant les données. Pour cela, on peut empoisonner les données d'entraînement ou modifier les conditions de capture (en modifiant la luminosité par exemple). Dans le cas des deepfakes, on cherche à tromper le modèle de détection afin que le contenu soit classé réel.

### 1.2.4 Quelles sont les motivations des attaquants ?

Encore une fois, c'est très varié. Dans le cas de l'évasion, le but est de contourner les systèmes de surveillance. Cela concerne par exemple une caméra dans la rue ou un modèle de détection de deepfake. Pour ce qui est de l'usurpation, l'idée est de se faire passer pour quelqu'un d'autre en utilisant, par exemple, le face swapping. Finalement, dans le cas de la désinformation, on pourrait chercher à partager et à diffuser de faux contenus sans qu'ils soient identifiés comme étant faux par un modèle.

### 1.2.5 Est-ce difficile à mettre en place ?

Maintenant que nous comprenons l'ampleur et l'importance de telles attaques, nous pouvons nous demander si cela est accessible. En réalité, il existe plusieurs types d'attaques, dont certaines seront détaillées dans ce document. L'attaque *Fast Gradient Sign Method* (FGSM - Goodfellow et al., 2015) fait partie des attaques utilisant les gradients du modèle, c'est à dire la direction, au sens mathématique dans laquelle il faut perturber l'image de manière à avoir une prédiction cible. Cette attaque est plutôt accessible, elle ne nécessite pas beaucoup de ressources et est facilement réalisable de par la présence de nombreuses bibliothèques existantes (notamment en python). L'attaque *Projected Gradient Descent* (PGD - Madry et al., 2019), faisant partie de la même famille, est destinée aux intermédiaires. En effet, disposant moins de librairies et nécessitant plus de ressources, cette attaque, bien que plus efficace, n'est pas facilement réalisable. D'autres attaques telles que DeepFool (Moosavi-Dezfooli et al., 2016) ou *Jacobian-based Saliency Map Attack* (JSMA - Papernot et al., 2015) reposent sur des principes différents et sont nettement plus complexes. Il existe un autre type d'attaque intéressant, cela concerne les attaques en temps réel, que l'on pourrait, par exemple, utiliser pour de la diffusion en direct ou des appels vidéo. Ces attaques relèvent un défi lié au fait qu'elles soient en direct. On cherche à limiter la latence afin de rester le plus discret possible. On observe donc un compromis entre efficacité et légèreté de l'attaque. Finalement, il est important de mentionner les attaques physiques (maquillage, lunettes...) que nous ne traiterons pas dans ce document, mais qui restent un axe majeur dans le thème des attaques adversariales.

## 1.3 Modèle utilisé

Comme nous l'avons compris, les attaques adversariales visent à tromper un modèle. Pour les tester, nous avons développé un modèle assez simple, qui comporte une couche de convolution suivie d'un *flatten*, puis deux couches denses (dont une de sortie). Il a été entraîné sur l'ensemble des données du *MNIST* (LeCun and Cortes, 2010). Son objectif est de classifier une image selon le chiffre dessiné sur cette dernière. L'*accuracy* de validation du modèle est de 0.97. Cela signifie que 97% des images ont été correctement classifiées, ce qui dénote un système globalement fiable. Le notebook de création du modèle est disponible en annexe, sous le nom de `train_model.ipynb`. Il a, par la suite, été adapté afin de produire une sortie avant la couche de *softmax*, ce qui est nécessaire pour certaines attaques. La figure 1 montre les prédictions réalisées par le modèle.

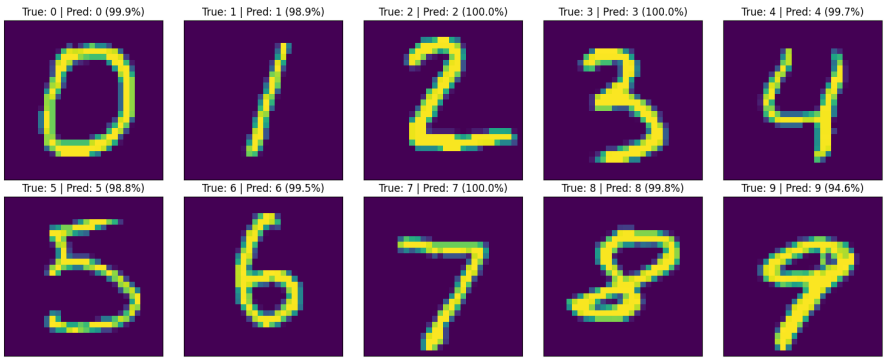


Figure 1: Prédictions du modèle sur un échantillon du dataset *MNIST*

## 1.4 Méthodologie d'étude

Étant débutants dans ce domaine, nous avons mis en place un protocole que nous avons suivi pour chaque attaque étudiée. L'objectif final de ce document est de pouvoir comparer ces différentes attaques afin de pouvoir sélectionner les plus adaptées à notre problème final : attaquer un système de détection de deepfakes.

Dans un premier temps, nous nous sommes renseignés en croisant différentes sources dans l'idée d'avoir une vue globale du paysage des attaques adversariales.

Dans un second temps, nous avons cherché des exemples d'implémentation pratique des attaques. Nous avons principalement utilisé les sites *github* et *kaggle*. L'idée de cette étape était de comprendre le résultat obtenu par les différentes attaques ainsi que de découvrir plusieurs manières de les implémenter. Cela nous a beaucoup aidés à mettre en lumière les différences entre chaque attaque.

Ensuite, nous avons implémenté ces attaques. Mettre en pratique nos connaissances théoriques nous a permis de confirmer les notions apprises, reconnaître les forces et faiblesses de chaque attaque, ainsi que de comprendre l'influence des différents paramètres sur l'efficacité de l'attaque.

En croisant ces trois axes de recherches, nous avons identifié le fonctionnement de chaque attaque étudiée, leurs forces et leurs faiblesses ainsi que leur difficulté d'implémentation. En liant ces informations, nous avons renforcé notre compréhension globale des attaques adversariales, tout en cherchant à maîtriser les subtilités de chaque attaque.

# 2. Etude

Nous allons détailler le fonctionnement de cinq attaques adversariales. Nous commencerons par les attaques utilisant le gradient du modèle. FGSM est connue pour être simple à implémenter et nous permettra d'avoir une première approche dans ce domaine. Ensuite, nous nous intéresserons à l'attaque PGD. C'est une version améliorée de FGSM. Ensuite, nous découvrirons en quoi la carte de saillance liée à un modèle peut permettre de l'attaquer en implémentant l'attaque JSMA. L'attaque DeepFool sera l'avant dernière attaque expérimentée. Nous l'avons choisie pour sa capacité à tromper un modèle de manière plus discrète que les précédentes attaques. Finalement, nous avons choisi d'étudier les perturbations universelles, connues, entre autres, pour leur rapidité d'application.

## 2.1 FGSM

### 2.1.1 Principe

L'attaque Fast Gradient Sign Method (FGSM) consiste à générer une image adversariale en ajoutant un bruit calculé à l'aide de la dérivée du coût du réseau. Le but ici est de maximiser la probabilité d'erreur du modèle sur cette image. Contraint par une norme $l^\infty$ et un paramètre d'intensité $\epsilon$, le bruit reste souvent imperceptible à l'œil nu, mais est capable de tromper le modèle.

L'idée est de "*pousser*" l'image dans la direction qui augmente le coût afin qu'elle ne soit plus affectée à la bonne classe ("*untargeted*"), ou de l'amener vers une classe précise ("*targeted*"). En pratique, FGSM fonctionne en mode *white-box*, c'est-à-dire qu'il a besoin d'avoir accès aux poids du modèle. Notons tout de même qu'il existe des variantes *black-box* utilisant par exemple le transfert d'exemples adversariaux (Goodfellow et al., 2015).

### 2.1.2 Formulation mathématique

La formulation classique de FGSM (*untargeted*):

Soient $x$ une image d'entrée, $y$ la classe d'appartenance associée, $f$ le modèle utilisé et $J$ la fonction de perte du modèle. L'attaque FGSM génère une image adversariale $x_{adv}$ selon la formule :

$$x_{adv} = x + \epsilon \cdot \text{sign}(\nabla_x J(f(x), y)) \tag{1}$$

- $\epsilon$: amplitude de la perturbation, typiquement entre 0.01 et 0.3 selon la normalisation (0-1 ou 0-255).
- sign: fonction *signe*, appliquée élément par élément.

Pour l'attaque ciblée ("*targeted*"), on diminue le coût associé à la cible $t$ désirée :

$$x_{adv} = x - \epsilon \cdot \text{sign}(\nabla_x J(f(x), t)) \tag{2}$$

### 2.1.3 Pseudo algorithme

1. Entrée : image $x$, label $y$, modèle $f$, epsilon $\epsilon$.
2. Calculer la dérivée du coût par rapport à $x$: $\nabla_x J(f(x), y)$
3. Fabriquer la perturbation: $\delta = \epsilon \cdot \text{sign}(\nabla_x J)$
4. Générer l'image adversariale: $x_{adv} = x + \delta$

5. Clamper (normaliser) le résultat à l'intervalle valide de pixels
6. Retourner : $x_{adv}$

### 2.1.4 Implémentation

Pour ce qui est des implémentations, nous avons choisi d'utiliser *DeepNote*. Cet outil nous permet de collaborer en temps réel sur des notebook, le tout avec assez de ressources pour mener à bien nos attaques adversariales. N'étant pas familier avec *Torch*, nous avons choisi d'utiliser *Tensorflow*. Le notebook d'implémentation de l'attaque FGSM est disponible en annexe, sous le nom de `fgsm.ipynb`.

**Attaque** Nous avons commencé par la variante non ciblée (*untargeted*). Le but ici est d'augmenter la valeur de *loss* afin de tromper au mieux le modèle. Pour un $\epsilon$ fixé à 0.20, nous avons obtenu des résultats satisfaisants. La figure 2 contient un échantillon de ces résultats. Par exemple, le chiffre 2 est prédit comme étant un 6 par le modèle avec une probabilité égale à 97.5%.

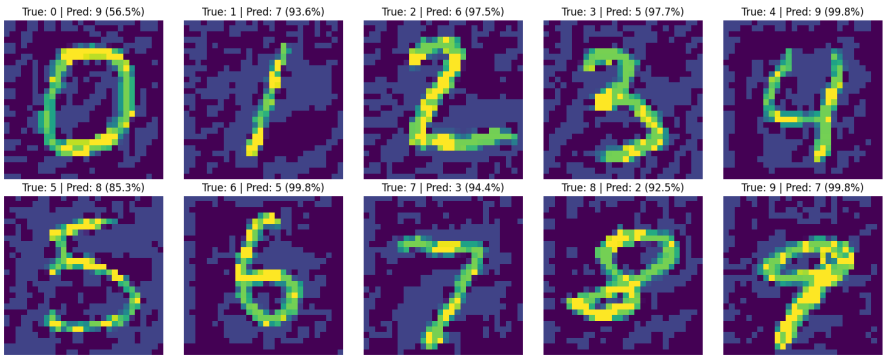


Figure 2: Attaque FGSM *untargeted*

Nous sommes allés plus loin en nous demandant à partir de quelle image il est le plus facile de tromper le modèle. Nous avons obtenu le graphique présent dans la figure 3.

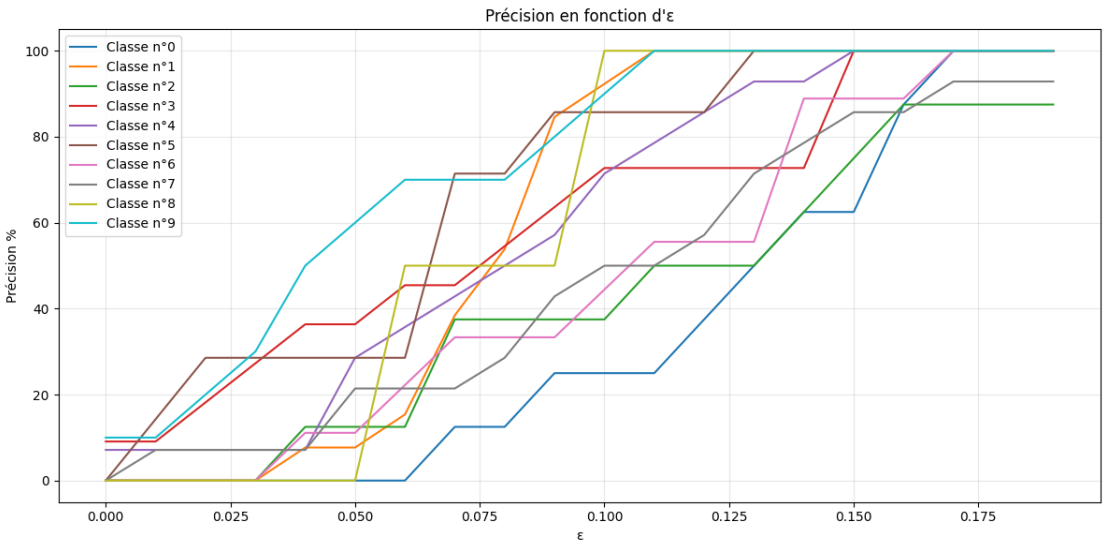


Figure 3: Précision d'attaque en fonction du chiffre - Attaque FGSM *untargeted*

Tout d'abord, nous remarquons que si $\epsilon$ augmente, l'efficacité de l'attaque augmente elle aussi. Cela vient directement de la manière dont l'attaque fonctionne. Ce mécanisme se retrouve dans la plupart des attaques adversariales. Ensuite, nous pouvons voir que tous les chiffres ne sont pas égaux face au modèle. Par exemple, il sera plus simple de tromper le modèle à partir d'une image de `9` ou de `5` qu'une image de `0` ou de `2`.

Dans un second temps, nous avons exploré la variante ciblée (*targeted*). Cette fois-ci, le but est d'avoir un loss minimisé selon une certaine classe (la classe ciblée). Cela nous permet de tromper le modèle dans le sens souhaité. Avec $\epsilon$ fixé à 0.20, nous avons obtenu les résultats présents dans la figure 4.

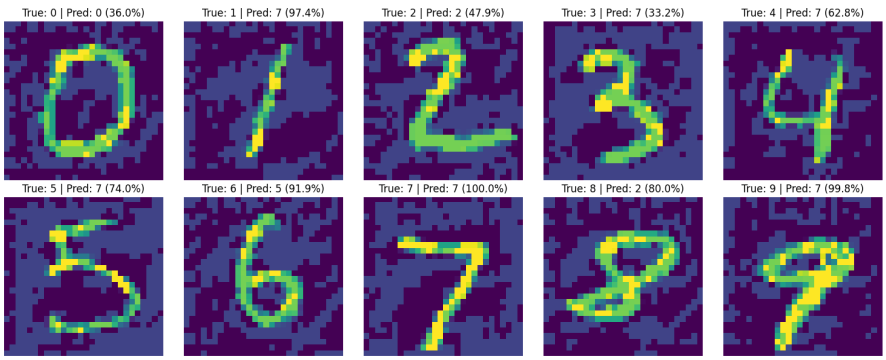


Figure 4: Attaque FGSM *targeted*

Avec le même état d'esprit que pour l'attaque *untargeted*, nous nous sommes demandé pour quel nombre il était le plus facile de tromper le modèle. Le résultat est présenté dans le graphique de la figure 5.

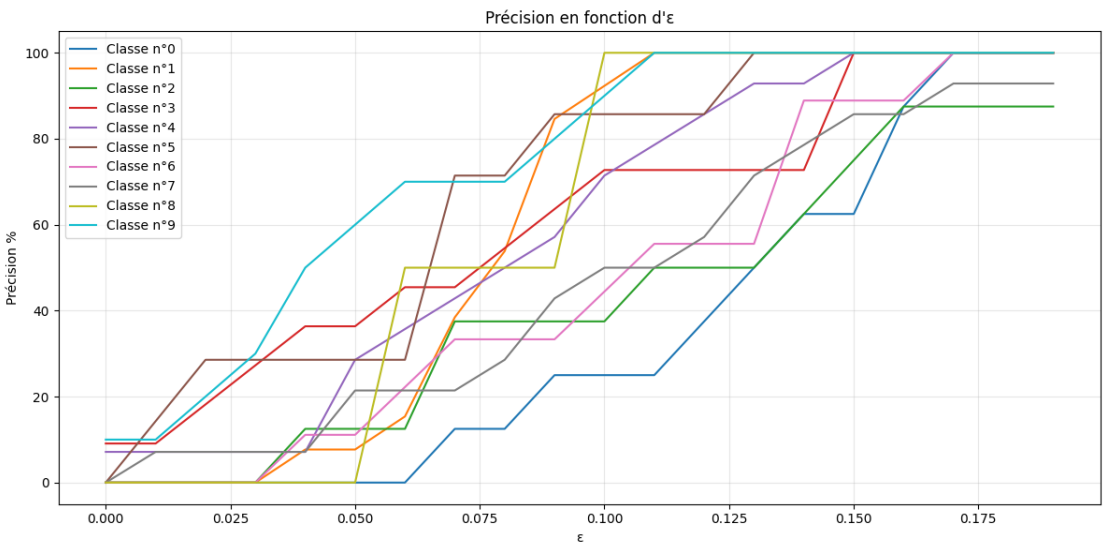


Figure 5: Précision d'attaque en fonction du chiffre - Attaque FGSM *targeted*

On remarque que, comme dans le graphique de la figure 4, si $\epsilon$ augmente, la précision générale augmente également. Toutefois, toutes les prédictions n'augmentent pas de manière égale. En effet, on remarque qu'il est plus simple de tromper le modèle de manière à ce qu'il pense lire un `3` ou un `7` plutôt qu'un `0` ou un `8`, par exemple.

**Influence du paramètre $\epsilon$** Nous avons cherché à manipuler le paramètre $\epsilon$ afin de comprendre comment il influe sur les résultats en pratique. Pour cela, nous l'avons simplement fait varier et nous avons noté la précision de l'attaque atteinte pour chaque valeur d'$\epsilon$. Cela nous a permis de dresser le graphique représenté en figure 6.

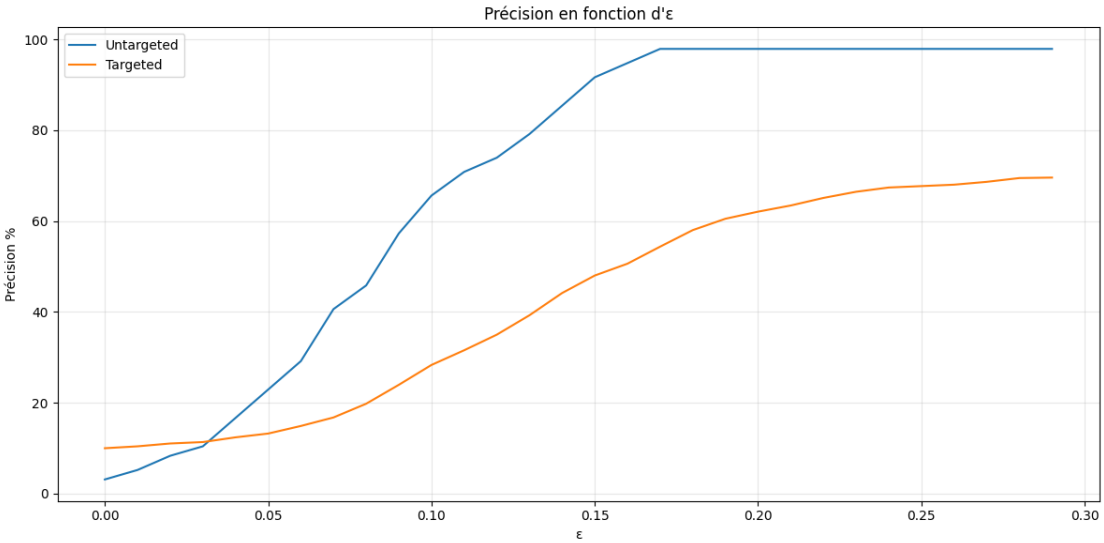


Figure 6: Comparaison des précisions des versions *targeted* et *untargeted* de l'attaque FGSM

Nous remarquons, sans surprise, que les précisions des attaques augmentent avec $\epsilon$. Les allures des courbes sont semblables. Toutefois, notons que la précision maximale atteinte par une attaque *targeted* est moins importante que celle obtenue par une attaque untargeted. Cela s'explique certainement par la complexité de ce type d'attaque. En effet, il est plus difficile de tromper un modèle dans un sens voulu que dans un sens aléatoire.

En connaissant ces graphiques un attaquant serait capable de fixer une valeur d'$\epsilon$ de manière à générer une image adversariale discrète mais efficace.

### 2.1.5 Application aux deepfakes et faceswapping

Quand on veut tromper un détecteur de deepfake ou de face swapping, on peut utiliser FGSM pour "*cacher*" un deepfake en perturbant l'image. FGSM peut également être utilisé pour empoisonner les données d'entraînement en leur ajoutant un bruit particulier, rendant alors le modèle moins efficace (Tolosana et al., 2020).

### 2.1.6 Limites

FGSM est rapide mais peu discret face à des défenses modernes (distillation, adversarial training, détecteurs robustes) (Kurakin et al., 2017). Des attaques plus sophistiquées sont généralement moins facilement détectables. Cette étude de l'attaque FGSM nous a permis de faire un premier pas dans le paysage des attaques adversariales, renforçant notre compréhension du fonctionnement global de ce type d'attaques.

### 2.1.7 Variante - *iFGSM*

*iFGSM* aussi appelée *BIM* est la variante itérative de FGSM. Le principe ne change pas, on avance toujours dans la direction du gradient pour maximiser la valeur de *loss*. On procède par petits pas, cela permet d'obtenir une attaque plus efficace. Cette version de FGSM est proche de la prochaine attaque, *PGD* mais reste moins efficace notamment à cause du manque de randomisation en début de parcours (Kurakin et al., 2017).

## 2.2 PGD

### 2.2.1 Principe

L'attaque Projected Gradient Descent (PGD) peut être vue comme une version itérative de l'attaque précédente, FGSM. L'idée générale est de répéter plusieurs fois une attaque FGSM minimisée dans le but d'obtenir une image adversariale dont la norme $l^\infty$ reste, à chaque étape, inférieure à $\epsilon$. À chaque itération, on avance d'un petit pas dans la direction du gradient du *loss*, puis on "*projette*" le résultat pour afin de vérifier que la norme reste correcte. On parle d'un résultat borné par une boule $l^\infty$ de rayon $\epsilon$ autour de l'image originale.

Ce procédé est plus coûteux en termes de calculs, mais est plus efficace que FGSM. PGD fonctionne, lui aussi avec des modèles *white-box* (Madry et al., 2019).

En pratique, avoir plusieurs itérations permet à PGD d'éviter les "*pièges locaux*" laissés par une unique montée de gradient, rendant la génération d'exemples adversariaux plus résistante aux pires cas.

### 2.2.2 Formulation mathématique

La formulation classique de PGD (*untargeted*):

Soient $x$ une image d'entrée, $y$ la classe d'appartenance associée, $f$ le modèle utilisé, $J$ la fonction de perte du modèle et $k$ le nombre d'itérations. L'attaque PGD génère une image adversariale $x_{adv}$ selon la formule :

On part de $x_0 = x$, puis, à chaque itération $t$ :

$$x_{t+1} = \text{Proj}_{B_\epsilon(x)} \left[ x_t + \alpha \cdot \text{sign} \left( \nabla_{x_t} J(f(x_t), y) \right) \right] \tag{3}$$

Avec $\text{Proj}_{B_\epsilon(x)}$ la projection dans la boule $l^\infty$ de rayon $\epsilon$ autour de $x$.

Souvent, on ajoute un "*random start*" : la première image est alors $x_0 = x + \text{Uniform}(-\epsilon, \epsilon)$. On utilise le "*random start*" pour éviter de bloquer l'attaque dans des minima locaux. Cela augmente significativement du taux de succès de l'attaque sans réellement complexifier l'algorithme (Madry et al., 2019).

### 2.2.3 Pseudo algorithme

1. Entrée : image $x$, label $y$, modèle $f$, paramètre $\epsilon$, taille de pas $\alpha$, nombre d'itérations $k$
2. Initialiser $x_0 = x + \text{Uniform}(-\epsilon, \epsilon)$
3. Pour $t = 0$ à $k - 1$ :
   - Calculer $g_t = \nabla_{x_t} J(f(x_t), y)$
   - Mettre à jour : $x'_{t+1} = x_t + \alpha \cdot \text{sign}(g_t)$
   - Projeter : $x_{t+1} = \text{clip}(x'_{t+1}, x - \epsilon, x + \epsilon)$
   - Clamper entre les valeurs de pixels permises (généralement entre 0 et 1)
4. Retourner : $x_k$

### 2.2.4 Implémentation

De la même manière que dans la dernière attaque, nous avons choisi d'implémenter PGD avec *Tensorflow*. L'attaque *untargeted* consiste toujours à maximiser le *loss*, tandis que la version *targeted* vise à minimiser le *loss* de la classe cible. Le notebook d'implémentation de l'attaque PGD est disponible en annexe sous le nom de `pgd.ipynb`.

**Attaque** Les images adversariales ont été générées avec les paramètres suivants : ($\epsilon = 0.2$, $\alpha = 0.01$, $k = 40$).

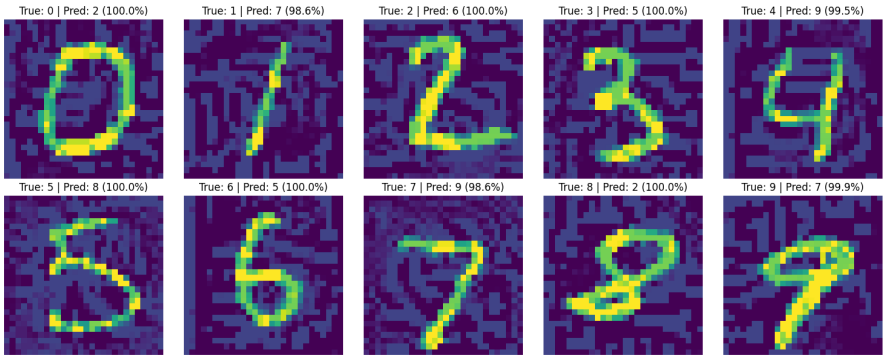


Figure 7: Attaque PGD *untargeted*

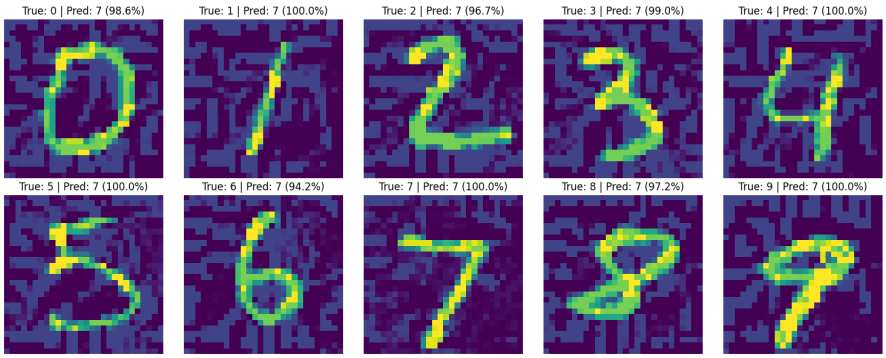


Figure 8: Attaque PGD *targeted*

On constate que les attaques *untargeted* (figure 7) et *targeted* (figure 8) fonctionnent bien. L'image trompe le modèle avec succès. La perturbation reste moins "*brutale*" et les images adversariales sont généralement de meilleure qualité, avec un bruit moins visible que celui généré par FGSM.

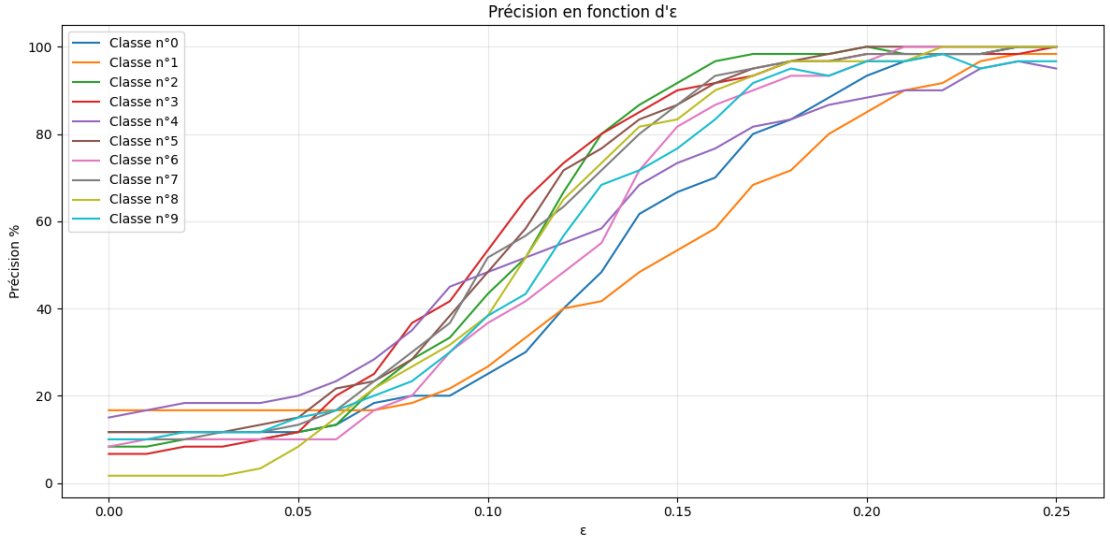


Figure 9: Précision d'attaque en fonction du chiffre - PGD *targeted*

On observe dans le graphique de la figure 9 que les patterns sont similaires : si $\epsilon$ augmente, la précision globale de l'attaque augmente. Cependant, tous les chiffres ne sont pas égaux face à cette augmentation. Par exemple, on remarque qu'il est simple de tromper le modèle pour lui faire prédire un `2` ou un `3`. Au contraire, les `1` et les `0` sont compliqués à cibler.

**Influence des paramètres** En faisant varier le nombre d'itérations, nous obtenons le graphique représenté par la figure 10. On constate alors que plus le nombre de pas est élevé, plus l'attaque est efficace jusqu'à saturation. Dans notre cas, avec $\epsilon$ réglé à 0.10, la précision se stabilise quand $k > 20$.

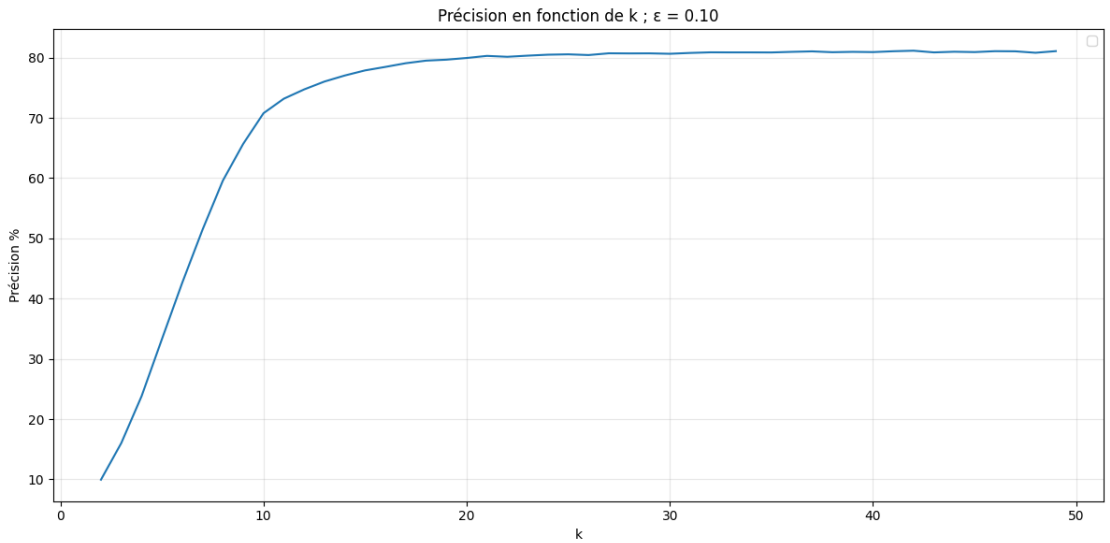


Figure 10: Précision d'attaque en fonction du nombre d'itérations $k$ - PGD *untargeted*

### 2.2.5 Application aux deepfakes et faceswapping

D'un côté, PGD est intéressant pour les attaques sur les détecteurs de deepfakes. En effet, sa capacité à générer des perturbations fines, mais efficaces sur des images permettrait certainement de tromper un détecteur sur une donnée.

D'un autre côté, PGD détériore généralement davantage les performances du modèle, même pour des petites valeurs de $\epsilon$ c'est certainement pour cela que PGD est considéré comme l'une des plus puissantes attaques adversariales de son type (Madry et al., 2019).

### 2.2.6 Limites

D'abord, PGD nécessite des ressources nettement supérieures à FGSM le rendant inutilisable pour les attaques en temps réel. Ensuite, son approche itérative peut amener à être bloqué dans un minimum local et donc, réaliser des itérations similaires pour un résultat peu satisfaisant (Liu et al., 2020). On peut également aborder le sujet du nombre important de paramètres qu'il faut régler correctement pour que l'attaque fonctionne avec succès.

### 2.2.7 Une variante de PGD : *MIM*

**Principe** Momentum Iterative Method (MIM) est une variante de PGD qui ajoute un terme de momentum (inertie) dans la "*direction*" de perturbation. L'idée est de stocker un historique des perturbations afin d'avoir une "*direction*" de mise à jour stable le tout pour éviter de s'enfermer dans des minima locaux de la fonction de *loss*. Cette version ajoute un paramètre et complexifie les calculs, mais répond à l'un des problèmes critiques de l'attaque PGD.

**Formulation mathématique** On conserve les notations vues dans la formulation mathématique de l'attaque PGD. On introduit deux éléments :

- $\mu$ : facteur de décroissance du momentum
- $g_t$ : un vecteur de momentum initialisé à 0

À chaque itération :

$$g_{t+1} = \mu\, g_t + \frac{\nabla_{x_t} J(f(x_t), y)}{\|\nabla_{x_t} J(f(x_t), y)\|_1}$$

$$x_{t+1} = \mathrm{Proj}_{B_\epsilon(x)} \left[ x_t + \alpha \cdot \mathrm{sign}(g_{t+1}) \right]$$

La normalisation par la norme $l^1$ du gradient permet de stocker les valeurs de manière stable. Quant à $\mu$ (souvent autour de 0.9) contrôle à quel point l'inertie a une décision importante dans la direction choisie.

**Influence du paramètre $\mu$** D'un côté, si $\mu$ est trop faible, on retrouve le comportement PGD de base, l'amélioration procurée par MIM est limitée. D'un autre côté, si $\mu$ est trop élevé, l'inertie devient excessive, l'attaque se comporte alors davantage comme FGSM. L'article publié par Dong et al. (2018) met en lumière ce fonctionnement particulier lié à l'inertie.

## 2.3 JSMA

### 2.3.1 Principe

À la différence des attaques basées uniquement sur la descente de gradient comme FGSM ou PGD, l'attaque JSMA (Jacobian-based Saliency Map Attack) ne modifie pas tous les pixels en même temps selon un gradient global. En effet, l'attaque sélectionne individuellement les pixels les plus *"influents"* sur la décision du modèle. L'idée générale est de construire une carte de saillance à partir de la jacobienne du réseau, qui mesure pour chaque pixel, l'impact d'une variation sur la probabilité de chaque classe.

### 2.3.2 Formulation mathématique

Soient $x$ une image d'entrée, $f$ le modèle utilisé et $J$ la matrice Jacobienne et $S$ la carte de saillance. L'attaque JSMA calcule la Jacobienne du modèle $J_{ij}(x)$ pour chaque $i$ et $j$ selon la formule :

$$J_{ij}(x) = \frac{\partial f_j(x)}{\partial x_i}, \tag{4}$$

où $x_i$ désigne le pixel n°$i$ de l'image, et $f_c(x)$ la sortie associée à la classe $c$.
Pour une attaque *targeted* vers une classe cible $t$, on a :

$$\alpha_i = \frac{\partial f_t(x)}{\partial x_i} \quad \text{(influence du pixel } i \text{ sur la classe cible)},$$

$$\beta_i = \sum_{j \neq t} \frac{\partial f_j(x)}{\partial x_i} \quad \text{(influence globale du pixel } i \text{ sur les autres classes)}.$$

La carte de saillance associe à chaque pixel une valeur

$$S_i = \begin{cases} 0 & \text{si } \alpha_i < 0 \text{ ou } \beta_i > 0, \\ \alpha_i \cdot |\beta_i| & \text{sinon.} \end{cases}$$

On remarque qu'un pixel n'est considéré comme *"utile"* que s'il augmente la probabilité de la classe cible ($\alpha_i < 0$). Le tout, en diminuant la somme des probabilités des autres classes ($\beta_i < 0$). On modifie alors les pixels ayant une forte valeur de saillance en premier. (Papernot et al., 2015)

### 2.3.3 Pseudo algorithme

1. Entrée : image $x$, label cible $t$, amplitude de modification par itération $\epsilon$, limite (%) de pixels modifiables $\gamma$
2. Initialisation
   - $x_0 = x$
   - Définir l'ensemble des pixels modifiables $\Omega_0 = \{1, \ldots, d\}$
   - compteur : $k = 0$
3. Tant que $f(x_k) \neq t$ et que la proportion de pixels modifiés $< \gamma$ :
   - Calculer la jacobienne $J(x_k)$
   - Pour chaque pixel $i \in \Omega_k$, on calcule $\alpha_i$, $\beta_i$ et $S_i$.
   - On choisi les pixels $p$ ayant la valeur de $S_i$ la plus grande
   - On met à jour ces pixels: $x_{k+1} = \epsilon \cdot \text{sign}(\alpha_p)$
   - On retire de $\Omega_k$ les pixels extrêmes (0 ou 1)
   - $k = k + 1$
4. Retourner : $x_k$

Cet algorithme peut être adapté de manière à modifier un seul pixel par itération, ou un petit groupe de pixels. Cela dans le but d'accélérer la convergence ou améliorer la probabilité de succès. (Papernot et al., 2015)

### 2.3.4 Implémentation

De la même manière que pour les autres attaques, nous avons choisi d'implémenter JSMA avec TensorFlow. Au lieu de chercher à optimiser la valeur de *loss* générale, nous avons cherché à calculer la carte de saillance relative à la sortie du modèle. Cela nous permet de précisément sélectionner quels pixels modifier. Comme pour les autres attaques, la version *targeted* vise toujours à augmenter la probabilité de la classe cible en diminuant celle des autres classes. Le notebook d'implémentation de l'attaque JSMA est disponible en annexe sous le nom de `jsma.ipynb`.

**Attaque** Nous avons concentré nos tests sur la version *targeted* de l'attaque.

- $\epsilon > 0$ : La figure 11 montre l'attaque JSMA sur les images du MNIST. Elle a été réalisée avec un paramètre $\epsilon$ positif. Cela a pour effet d'ajouter des pixels. On remarque alors que les pixels sont modifiés de manière très localisée, ce qui rend l'attaque moins discrète. Toutefois, les images adversariales générées trompent avec succès le modèle.

- $\epsilon < 0$ : La figure 12 montre l'attaque JSMA sur les images du MNIST. Elle a été réalisée avec un paramètre $\epsilon$ négatif. Cela a pour effet d'affaiblir voir mettre à zéro des pixels. Les images adversariales générées sont peu efficaces pour tromper le modèle.

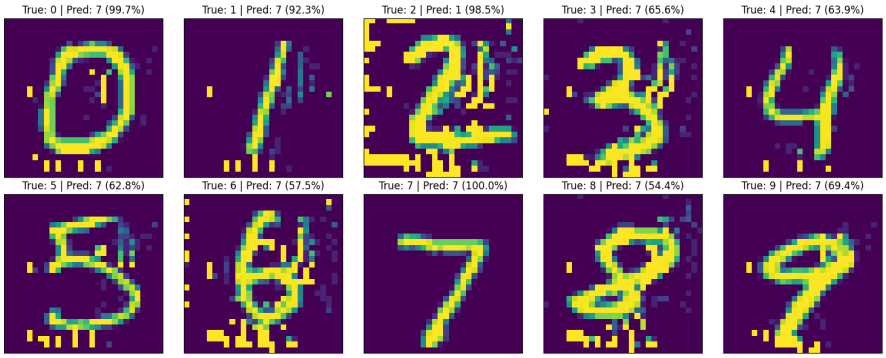


Figure 11: Attaque JSMA *targeted* - paramètre $\epsilon$ positif

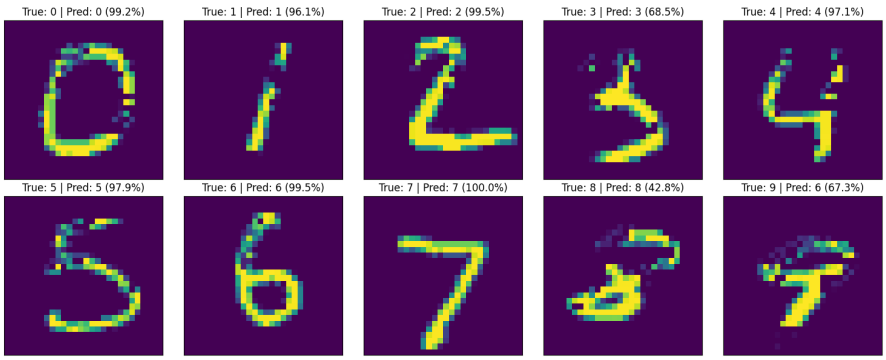


Figure 12: Attaque JSMA *targeted* - paramètre $\epsilon$ négatif

**Influence des paramètres** Nous avons cherché à manipuler les paramètres de JSMA afin de comprendre comment ils influent sur les résultats en pratique.

- $\epsilon$ (amplitude de la perturbation par pixel) : D'un côté, un $\epsilon$ trop faible nécessite un grand nombre itérations pour atteindre la classe cible, ce qui rend le calcul plus long. D'un autre côté, un $\epsilon$ trop grand modifie de manière brutale les pixels, au risque de rendre la perturbation évidente ou de saturer les valeurs de pixels (0 ou 1). La figure 13 montre l'influence du paramètre $\epsilon$ sur le taux de succès de l'attaque, et ce, en fonction de la classe cible.

- $\gamma$ (pourcentage max. de pixels modifiés) : Ce paramètre contrôle directement l'étendue de l'attaque. D'un côté, un $\gamma$ faible force l'attaque à trouver peu de pixels très influents, au risque de ne pas atteindre la classe cible. D'un autre côté, si on augmente $\gamma$, le taux de succès de l'attaque augmente, mais l'image est davantage modifiée.

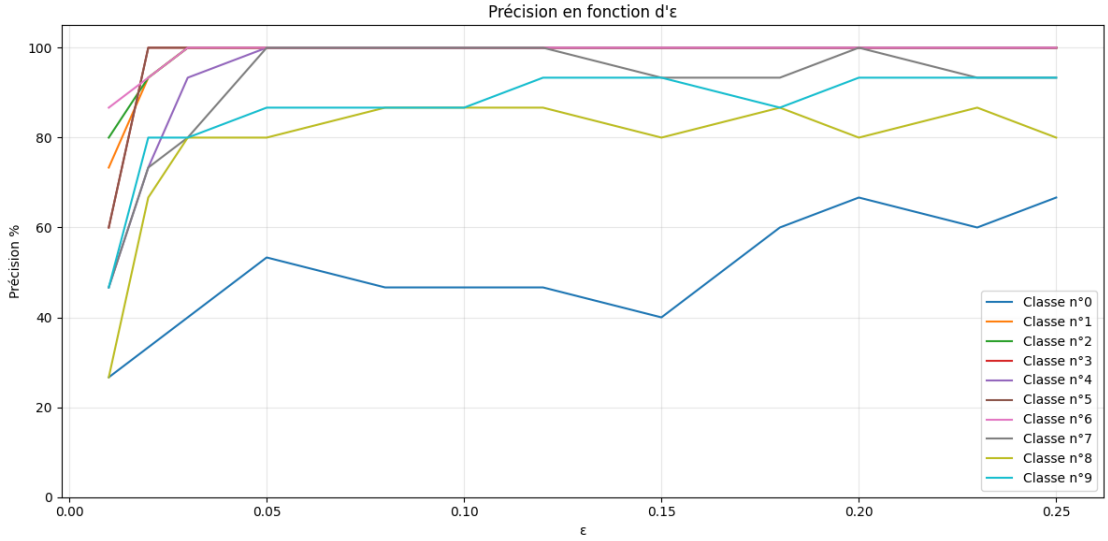


Figure 13: Précision d'attaque pour chaque classe cible en fonction de $\epsilon$ - JSMA *targeted*

De manière générale, on retrouve des comportements similaires aux autres attaques. En effet, si on augmente la liberté accordée à JSMA (via $\epsilon$ et $\gamma$), le modèle est mieux trompé. Cependant, contrairement aux attaques telles que FGSM, pour lesquelles on contrôle directement $\epsilon$ sur l'ensemble de l'image, ici les paramètres portent essentiellement sur le nombre de pixels modifiés et leur amplitude.

### 2.3.5 Application aux deepfakes et faceswapping

Nous l'avons compris : la force de JSMA est d'être capable d'identifier des régions très locales de l'image qui influencent fortement sur la décision du modèle. Cela s'avère très intéressant dans des perspectives liées aux deepfakes et au face swapping.

Il serait facile d'utiliser JSMA sur un détecteur de deepfakes pour cibler directement la sortie *"real"* et ainsi trouver les pixels qui font basculer la prédiction du modèle. Contrairement à un bruit global (comme avec FGSM), JSMA modifie seulement quelques zones clé de l'image. Cette perturbation agrandie alors les possibilités d'attaque.

Notons également que la *saliency map* issue de JSMA peut être utilisée comme outil d'analyse. En effet, elle met en lumière les parties de l'image influençant le plus le modèle. Cela ouvre la voie à de nouveaux schémas d'attaque, où un générateur de deepfakes pourrait apprendre à corriger ces régions sensibles dans le but de réduire la probabilité d'être reconnu. Le tout, en conservant la qualité visuelle de l'image. (Wiyatno and Xu, 2018)

### 2.3.6 Limites

JSMA présente plusieurs limites :

- **Coût de calcul élevé** : Le calcul de la jacobienne complète, puis de la carte de saillance est assez coûteux, surtout pour des images de grande dimension ou avec des réseaux profonds. En effet, chaque itération nécessite au moins un passage complet dans le réseau avec un calcul de gradients par rapport à tous les pixels. (Wiyatno and Xu, 2018)

- **Peu adapté aux images haute résolution** : Sur MNIST ($28 \times 28$), JSMA fonctionne en un temps cohérent (environ 5 secondes). Sur des images de visages en haute résolution, la dimension d'entrée devient rapidement importante rendant l'attaque inutilisable sans techniques de réduction. (*downsampling*, sélection de régions d'intérêt...). (Combey et al., 2020)

Ces limitations expliquent en partie pourquoi JSMA est aujourd'hui surtout utilisé comme outil d'analyse ou comme base pour des attaques plus efficaces, plutôt que comme attaque de référence sur des modèles de vision. (Combey et al., 2020)

### 2.3.7 Variantes de JSMA

Plusieurs variantes de JSMA ont été proposées pour améliorer son efficacité, réduire son coût de calcul ou l'adapter à d'autres scénarios.

**NT-JSMA (Non-Targeted JSMA)** NT-JSMA adapte JSMA pour incorporer une version *untargeted*. L'objectif n'est plus de forcer la prédiction vers une classe cible, mais simplement de faire sortir l'image de sa classe correcte.

En pratique, la définition de la saillance est modifiée dans le but de favoriser les pixels qui réduisent la probabilité de la vraie classe, sans chercher à optimiser la répartition sur les autres classes. (Wiyatno and Xu, 2018)

**MJSMA (Maximal JSMA)** MJSMA cherche à prendre le meilleur de JSMA et NT-JSMA. En effet, plutôt que de fixer à l'avance si l'on doit augmenter ou diminuer chaque pixel, on cherche, pour chaque pixel, la mise à jour qui mène vers la version souhaitée (ciblé ou non ciblé).

L'idée est de choisir la combinaison *"mise à jour + signe de la perturbation"* qui approche le plus de l'objectif, d'où le terme *"maximal"*. MJSMA réduit ainsi plusieurs choix de JSMA (Par exemple : fixer le signe de $\epsilon$) et peut obtenir, un meilleur taux de réussite sans changer $\gamma$. (Wiyatno and Xu, 2018)

**WJSMA (Weighted JSMA)** WJSMA introduit une pondération pour chaque pixel, dans le but de mieux tenir compte de la structure de l'image et de l'architecture du modèle.

Plutôt que de se baser uniquement sur $\alpha_i$ et $\beta_i$, WJSMA ajoute un poids à la saillance par une fonction de poids $w_i$. L'objectif est alors de guider l'attaque vers des pixels efficaces pour tromper le modèle tout en respectant une contrainte additionnelle (Par exemple : limiter les perturbations dans des zones visibles pour l'œil humain). (Combey et al., 2020)

## 2.4 DeepFool

### 2.4.1 Principe

À la différence des autres attaques, DeepFool ne cherche pas à maximiser directement une valeur de *loss* ou à "*pousser*" dans la direction du gradient. En effet, cette attaque s'applique à chaque image pour trouver le plus petit bruit, suffisant pour changer la prédiction du modèle. Pour cela, DeepFool commence par linéariser le modèle autour de l'image avant de chercher le plus petit chemin franchissant la frontière de décision. On avance alors de manière itérative vers la première classe différente. Cette approche génère une perturbation à peine perceptible. DeepFool est conçu pour fonctionner de manière *untargeted* (Moosavi-Dezfooli et al., 2016).

### 2.4.2 Formulation mathématique

Soient $x_0$ une image d'entrée, $f$ le modèle utilisé et $k_0$ la prédiction initiale. L'attaque DeepFool génère une perturbation adversariale $r_t$ selon les formules :

On procède en estimant localement la frontière de décision de chacune des classes différentes de la classe d'origine. Ces frontières sont représentées sous la forme d'hyperplans.

À chaque itération, on calcule, pour chaque classe $k \neq k_0$ :

$$w_k = \nabla f_k(x_t) - \nabla f_{k_0}(x_t) \tag{5}$$

$$l_k = \frac{|f_k(x_t) - f_{k_0}(x_t)|}{\|w_k\|_2} \tag{6}$$

La perturbation qui trompera le modèle sera donnée par :

$$r_t = \frac{|f_{k^*}(x_t) - f_{k_0}(x_t)|}{\|w_{k^*}\|_2^2} w_{k^*} \tag{7}$$

où $k^*$ est la classe la plus proche selon la distance $l_k$.

### 2.4.3 Pseudo-algorithme

1. Entrée : image $x_0$, modèle $f$, classe initiale $k_0$
2. Tant que la prédiction reste $k_0$ :
   - Linéariser localement le classifieur autour de $x_t$
   - Pour chaque classe $k \neq k_0$, calculer le vecteur $w_k$ et distance projetée $l_k$
   - Sélectionner la classe $k^*$ la plus proche
   - Calculer et appliquer la perturbation minimale $r_t$ dans la direction $w_{k^*}$
   - Mettre à jour : $x_{t+1} = x_t + r_t$
3. Arrêt dès que la classe prédite change
4. Retourner : dernière valeur de $x$ calculée.

(Moosavi-Dezfooli et al., 2016, pp. 5).

### 2.4.4 Implémentation

Le notebook de l'implémentation de l'attaque DeepFool est disponible en annexe, sous le nom de `deepfool.ipynb`.

**Attaque** En pratique, DeepFool s'utilise en *white-box*. Cela s'explique par le besoin de calcul des gradients internes pour chaque sortie du modèle. Contrairement à FGSM ou PGD, il n'y a pas besoin de fixer un $\epsilon$ en avance. En effet, l'amplitude de la perturbation ($\epsilon$) est calculée à chaque itération (tout comme les frontières de décision), cela dans le but d'obtenir un bruit minimal (voir figure 14). Cette manière globale de faire rend l'attaque DeepFool très sensible au nombre de classes en sortie du modèle (dans notre cas, il y en a 10). De manière générale, DeepFool est plus léger en calculs que PGD, car il converge généralement en peu d'itérations. Notons que l'image prédite est juste assez modifiée pour que le modèle classifie le nombre comme étant d'une autre classe. C'est pour cette raison que nous obtenons souvent des pourcentages de confiances peu élevés, aux alentour de 50%.

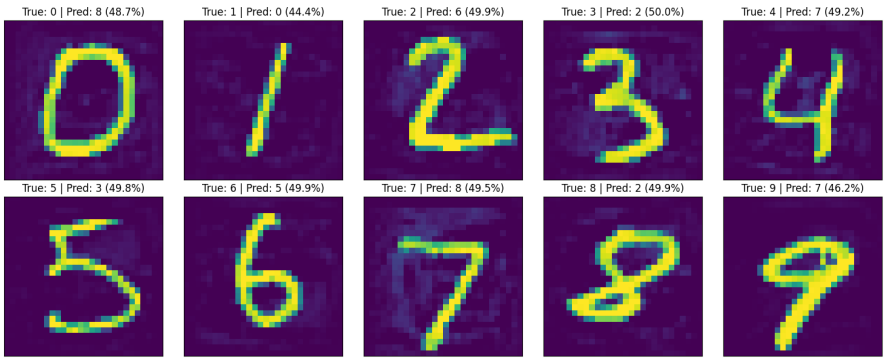


Figure 14: Attaque DeepFool *untargeted*

**Influence des paramètres** Le principal paramètre est le nombre d'itérations maximal. De manière générale, plus la complexité de la frontière de décision autour de l'exemple est grande, plus l'attaque nécessitera d'itération pour réussir.

Comme vu précédemment, le nombre de classes n'est pas réellement un paramètre facilement modifiable par un attaquant mais reste déterminant pour le succès de l'attaque.

Le paramètre *overshoot* détermine à quel point il faut dépasser la frontière de décision. On peut le régler proche de 0 si la discrétion est privilégiée par rapport à l'efficacité de l'attaque. Sinon, on préfère des valeurs supérieures à 0, dans un cas où l'efficacité est plus importante.

L'attaque peut fonctionner avec différentes normes ($l^2$, $l^\infty$), mais le choix par défaut est souvent la norme $l^2$, qui produit des résultats plus "*propres*" dans la mesure de la robustesse (Tsipras et al., 2019, pp. 14–15).

### 2.4.5 Application aux deepfakes et faceswapping

DeepFool est une attaque importante contre les détecteurs de deepfakes. Cela s'explique par le fait que l'attaque s'adapte de manière locale et perturbe de manière minimale l'image. Il est possible d'étudier à quel point un modèle détecteur de deepfakes peut être contourné "*à la frontière*". Cela surtout dans les cas pour lesquels l'image non adversariale est déjà proche d'une frontière. DeepFool permet de mettre en évidence la zone critique à modifier. Ce fonctionnement, basé sur les frontières de décision, permet à DeepFool de révéler les failles d'un modèle permettant une évaluation de la robustesse d'un modèle.

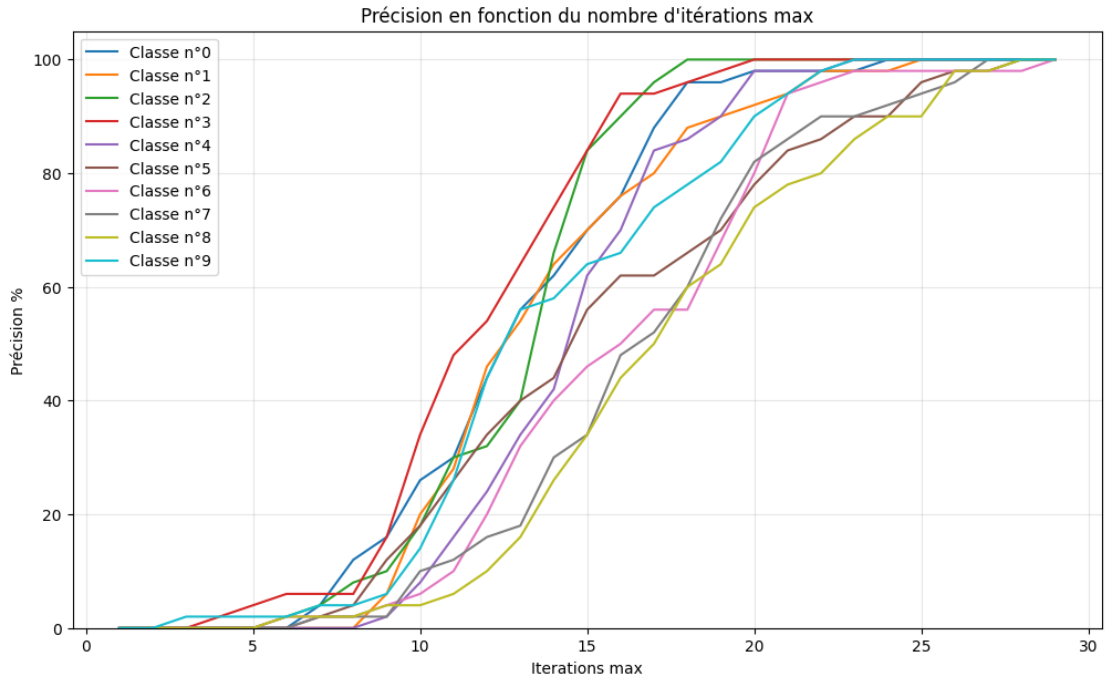


Figure 15: Précision d'attaque en fonction du nombre maximum d'itérations - DeepFool *untargeted*

Le graphique présenté dans la figure 15 montre que l'efficacité de l'attaque est fortement lié au nombre maximum d'itérations. Toutefois, comme dans les précédentes attaques, tous les nombres d'origine ne sont pas égaux face à la réalisation de l'attaque. En effet, tromper le modèle à partir d'un `2` ou d'un `3` est plus simple que depuis un `8` par exemple.

### 2.4.6 Limites

Les deux principales limitations sont que DeepFool ne permet pas la génération d'images adversariales *targeted* et que l'attaque est sensible au nombre de classes. Dans le cadre d'un classificateur binaire cela ne posera pas forcément problème, mais il s'agit de limites qu'il faut garder en tête.

### 2.4.7 Variantes

**Targeted DeepFool** Le but de cette variante est de répondre au fait que DeepFool ne fonctionne pas de manière *targeted* naturellement. Cela peut être intéressant quand on a plusieurs classes. Cependant, dans le cas d'une classification binaire telle que celle effectuée par un modèle de détection de deepfakes, cette variante ne s'avère pas intéressante. Le fonctionnement est modifié. On ne s'intéresse plus à la frontière la plus proche, mais plutôt à la frontière avec la classe cible. Ainsi, le fonctionnement global est conservé, on se contente simplement de "*pousser*" l'image au-delà de cette frontière de décision (Shiva28, 2022).

Nous avons cherché à en apprendre plus sur la version *targeted* de DeepFool. C'est pour cela que nous l'avons implémenté. En nous basant sur l'idée générale présentée ci-dessus, nous avons obtenu les résultats présentés dans les figures 16 et 17.

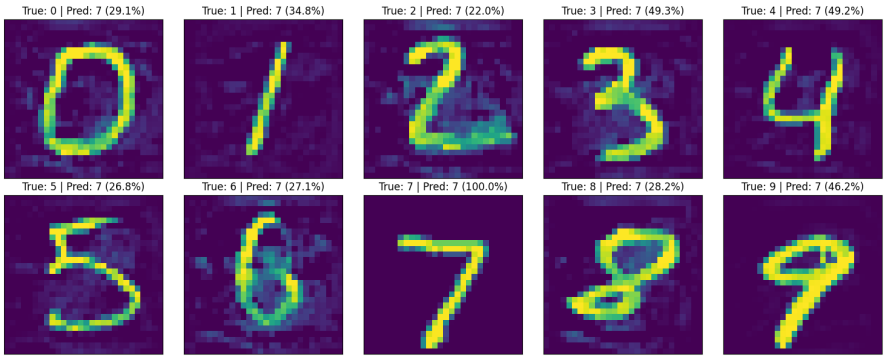


Figure 16: Attaque DeepFool *targeted*

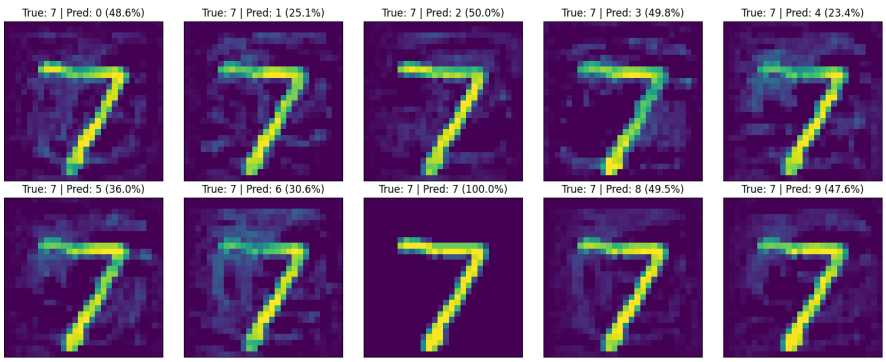


Figure 17: Attaque DeepFool *targeted*, chiffre de départ constant

**Enhanced Targeted DeepFool** Cette variante reprend les grands axes de la variante *Targeted DeepFool* en allant plus loin. En ajustant les paramètres tels que l'*overshoot* et en affinant la direction de la perturbation, l'image adversariale obtenue est de bien meilleure qualité. L'idée globale est toujours la même, toutefois, cette version propose une optimisation de la taille du chemin entre l'image originale et la frontière de la classe ciblée en recalculant les chemins optimaux. Cette technologie permet d'attaquer des modèles complexes ou même robustes, le tout en réduisant le nombre d'itérations. Le niveau de perturbation obtenu dans l'image adversariale reste tout de même faible (Labib et al., 2025).

**SuperDeepFool** L'objectif de cette variante est d'obtenir des perturbations minimales en optimisant l'orthogonalité de la perturbation. Le bruit généré est encore plus difficile à détecter et la longueur du chemin est diminuée. L'avantage de cette attaque est qu'elle s'adapte très bien aux modèles complexes. Cela en fait une alternative solide à *ET-DeepFool* en version *untargeted* (Abdollahpoorrostam et al., 2024).

## 2.5 Universal Adversarial Perturbations (UAP)

### 2.5.1 Principe

Les Universal Adversarial Perturbations (Moosavi-Dezfooli et al., 2017) sont, comme leur nom l'indique, des perturbations universelles : lorsqu'elles sont ajoutées à n'importe quelle image, elles entraînent une mauvaise classification par le modèle avec une probabilité élevée.

Contrairement aux attaques adversariales listées ci-dessus, qui créent une perturbation spécifique pour chaque image, une seule UAP peut servir à tromper le modèle sur une grande partie des images d'un jeu de données.

Pour générer cette perturbation, on la construit pas à pas en itérant sur le jeu de données. À chaque passage, on identifie les images qui sont encore correctement classifiées et on cherche une perturbation minimale pour tromper le modèle. Pour trouver ces perturbation locales, on utilise un algorithme d'attaque adversariale, par exemple, DeepFool. On accumule ensuite ces perturbations locales pour mettre à jour la perturbation universelle.

Par défaut, les UAP sont donc *untargeted*.

### 2.5.2 Formulation mathématique

- $X$ : jeu d'images
- $f$ : modèle
- $v$ : vecteur de perturbation

Soient $x$ jeu d'images en entrée, $f$ le modèle utilisé et $v$ le vecteur de perturbations. L'attaque UAP génère une perturbation adversariale universelle $v$ de cette manière :

Notre but est de trouver un $v$ tel que :

$$f(x+v) \neq f(x) \quad \text{pour la plupart des } x \in X, \tag{8}$$

$v$ est évalué selon deux contraintes :

**Magnitude :**

$$\|v\|_p \leq \xi. \tag{9}$$

où $\xi$ contrôle la magnitude de la perturbation

**Fooling rate (taux de tromperie) :**

$$\Pr_{x \in X}\big(f(x+v) \neq f(x)\big) \geq 1-\delta. \tag{10}$$

où $\delta$ quantifie les images parvenant à tromper le modèle grâce à la perturbation universelle.

### 2.5.3 Pseudo-algorithme

1. Entrée : jeu de données $X$, modèle $f$, taux de précision désiré $\delta$, norme $l_p$, rayon $\xi$
2. Initialiser la perturbation $v = 0$
3. Tant que $Err(X) < 1 - \delta$
   - Pour toute image $x \in X$, si $f(x+v) = f(x)$ :
     (a) Calculer la perturbation minimale $d_r$ pour changer la classe avec l'attaque souhaitée (DeepFool, PGD, etc.)
     (b) Mettre à jour la perturbation : $v = v + d_r$
     (c) Projeter $v$ sur $l^p$ de rayon $\xi$
4. Retourner : perturbation $v$

### 2.5.4 Implémentation

Nous avons choisi d'implémenter deux versions D'UAP :

- **UAP-DeepFool** : Cette version d'UAP calcule la perturbation globale en se basant sur la version *untargeted* de l'attaque DeepFool étudiée précédemment. Cette version présente les avantages d'être facile à implémenter et rapide à exécuter. Toutefois, les nombreuses classes présentes dans le set de données du MNIST rend la perturbation universelle peu discrète.
- **UAP-PGD** : La deuxième version se base sur un UAP construit avec l'attaque PGD. Cela permet de répondre au problème du nombre de classes vu précédemment. L'avantage de l'attaque PGD est qu'elle est aussi rapidement exécutée, réduisant alors le calcul global de la perturbation universelle.

Les notebooks sont disponibles en annexe, respectivement sous les noms de `uap-deepfool.ipynb` et `uap-pgd.ipynb`.

**Attaque** La figure 18 montre la perturbation universelle générée par UAP-DeepFool pour les images du MNIST. La figure 19 montre les résultats obtenus en la combinant aux dites images.

Pour générer cette perturbation, nous avons utilisé un sous-jeu de données composé de 200 images de MNIST. Ce dernier nous a permis d'obtenir un *fooling rate* de 87.5% en une seule itération.

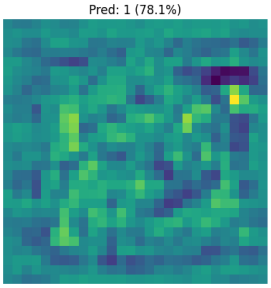


Figure 18: Perturbation universelle générée par UAP-DeepFool

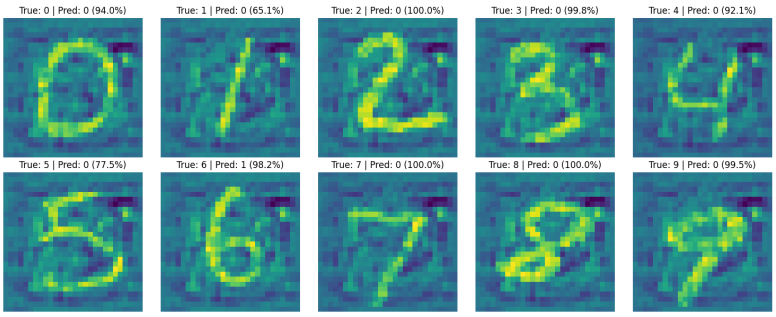


Figure 19: Perturbation Universelle générée par UAP-DeepFool appliquée à MNIST

Nous avons remplacé la méthode de calcul de la perturbation minimale par PGD. La figure 20 montre la perturbation universelle générée de cette manière, et la figure 21 son application sur les images du MNIST.

Les temps de calcul étant manifestement plus longs qu'avec l'algorithme UAP-DeepFool, nous avons été contraints de baisser le nombre d'images utilisées à 100, ainsi que de baisser *fooling rate* minimal attendu $\delta$. Ainsi, avec 2 itérations, nous avons obtenu une perturbation universelle avec *fooling rate* de 62%.

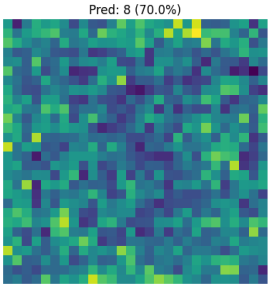


Figure 20: Perturbation universelle générée par UAP-PGD

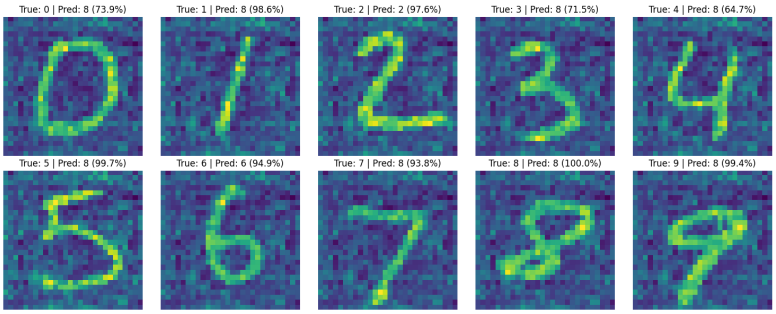


Figure 21: Perturbation Universelle générée par UAP-PGD appliquée à MNIST

**Influence des paramètres** La génération d'une perturbation requiert un réglage précis de plusieurs paramètres. En effet, comme nous l'avons vu, il existe des paramètres réglant la génération en elle-même de l'UAP. Toutefois, il faut aussi régler les paramètres de l'attaque sur laquelle UAP se base. En pratique, si l'on cherche par exemple, à générer une perturbation universelle en se basant sur DeepFool, le résultat dépendra entre autres, de :

- **Paramètres de UAP**
  1. *fooling rate cible* ($\delta$) : Il s'agit d'une cible concernant taux d'images trompant le modèle une fois la perturbation appliquée. De manière générale, un $\delta$ important nécessite plus d'itérations.
  2. taille du jeu de données : Plus le jeu de données est grand, plus la génération est complexe et lente. C'est pour cette raison que nous avons limité la taille du jeu de données.
  3. magnitude de la perturbation ($x_i$) : C'est la vitesse à laquelle on souhaite modifier l'image. De manière générale, une petite magnitude implique un long temps de calcul, mais un résultat optimisé.
  4. éventuellement un nombre max d'itérations autorisées pour la génération d'UAP.
- **Paramètres de l'attaque sur laquelle UAP repose (DeepFool ici)**
  1. *overshoot*
  2. nombre d'itérations maximum
  3. nombre de classes

### 2.5.5 Application aux deepfakes et faceswapping

Comme nous l'avons vu, une fois générée, une perturbation universelle peut être appliquée à de nouvelles images sans nécessiter de calcul supplémentaire. Cette propriété les rend particulièrement adaptées au traitement vidéo en direct.

De plus, un aspect à prendre en compte avec les UAP est leur accessibilité : une perturbation générée peut ensuite être partagée, reprise et appliquée par n'importe quel utilisateur sans expertise technique particulière.

Pour aller plus loin, d'après Neekhara et al. (2020), les UAP générées contre un modèle donné conservent une bonne efficacité même sur d'autres modèles, ce qui les rend intéressantes pour des attaques en *black-box*.

### 2.5.6 Limites

Les UAP présentent plusieurs freins à prendre en compte :

- **Temps de génération** : Contrairement aux attaques adversariales classiques, où une perturbation est calculée une seule fois par entrée, la construction d'une UAP nécessite de parcourir plusieurs fois le jeu de données dans son intégralité. Obtenir un *fooling rate* satisfaisant peut requérir de nombreuses itérations, ce qui entraîne un temps de calcul significatif.
- **Perceptibilité** : Comme les perturbations universelles doivent convenir pour tromper un large ensemble d'images, elles ont tendance à être plus prononcées et donc moins discrètes que les attaques adversariales classiques, optimisées spécifiquement pour une image.

### 2.5.7 Variantes

**Fast-UAP** La variante Fast-UAP telle que proposée dans Dai and Shu (2020) vise à accélérer la génération des perturbations universelles en réduisant le temps de calcul. Pour cela, lors de chaque itération sur les images du jeu de données, on ne cherche plus le vecteur de perturbation minimal permettant de tromper le modèle : on prend une perturbation qui provoque une erreur, même si elle n'est pas minimale. Cette stratégie permet d'obtenir des perturbations universelles atteignant un même taux de tromperie plus rapidement et avec moins d'images. En contrepartie, les perturbations obtenues sont moins discrètes.

**Data-free UAP** Comme son nom l'indique, cette variante d'UAP ne nécessite pas d'accéder au jeu de données d'entraînement pour générer les perturbations. Elle se sert uniquement des paramètres du modèle cible et de techniques de synthèse d'images. Les perturbations générées de cette manière ont des taux de tromperie moins élevés. (Lee et al., 2025)

## Tableau récapitulatif

| **Attaque** | Original | FGSM | PGD | JSMA | DeepFool | UAP-DF | UAP-PGD |
|---|---|---|---|---|---|---|---|
| Temps | – | $\sim 10^{-3}$ s | $\sim 10^{-1}$ s | $\sim 10^{0}$ s | $\sim 10^{-1}$ s | $\sim 10^{3}$ s | $\sim 10^{3}$ s |
| Classification | 9 (94.6%) | 7 (99.8%) | 7 (99.9%) | 7 (69.4%) | 7 (46.2%) | 0 (99.5%) | 8 (99.4%) |
| Image |  | | | | | | |

Table 1: Tableau comparatif des attaques adversariales étudiées

# Conclusion

Ce travail a permis d'explorer les principales familles d'attaques adversariales, depuis des méthodes simples comme FGSM jusqu'à des approches plus complexes telles que JSMA et DeepFool. Les différentes méthodes étudiées montrent qu'il n'existe pas "une" attaque adversariale, mais bien un large éventail de stratégies. Malgré leurs différences, ces attaques montrent à quel point un modèle d'apprentissage est sensible à de minimes perturbations calculées avec précision.

Les implémentations et tests réalisés ont démontrés que l'efficacité de l'attaque dépend fortement de plusieurs paramètres tels que l'image manipulée, le bruit généré ($\epsilon$), le nombre d'itérations... Certaines attaques sont rapides mais moins contrôlables tandis que d'autres, plus coûteuses, produisent des résultats plus précis. Le fonctionnement de ces attaques reste cependant limité par le fait que le modèle attaqué est accessible (attaque *white-box*), assurant la disponibilité des gradients.

Cette étude met également en lumière des techniques particulières comme les perturbations universelles, qui élargissent encore la surface de menace en rendant les attaques plus généralisables et facilement applicables à des données en direct, par exemple des lives ou appels vidéo.

En résumé, comprendre ces attaques est indispensable pour renforcer la robustesse des modèles. Les futurs travaux devront continuer d'avancer sur l'amélioration des attaques afin de mieux diagnostiquer les failles et sur la conception de défenses capables de rendre les systèmes réellement résistants.

# References


Abdollahpoorrostam, A., Abroshan, M., and Moosavi-Dezfooli, S.-M. (2024). Revisiting deepfool: generalization and improvement.

Combey, T., Loison, A., Faucher, M., and Hajri, H. (2020). Probabilistic jacobian-based saliency maps attacks.

Dai, J. and Shu, L. (2020). Fast-uap: An algorithm for speeding up universal adversarial perturbation generation with orientation of perturbation vectors.

Dong, Y., Liao, F., Pang, T., Su, H., Zhu, J., Hu, X., and Li, J. (2018). Boosting adversarial attacks with momentum.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples.

Kurakin, A., Goodfellow, I., and Bengio, S. (2017). Adversarial examples in the physical world.

Labib, S. M. F. R., Mondal, J. J., Manab, M. A., Xiao, X., and Newaz, S. (2025). Tailoring adversarial attacks on deep neural networks for targeted class manipulation using deepfool algorithm. *Scientific Reports*, 15(1).

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

Lee, C., Song, Y., and Son, J. (2025). Data-free universal adversarial perturbation with pseudo-semantic prior.

Liu, C., Salzmann, M., Lin, T., Tomioka, R., and Süsstrunk, S. (2020). On the loss landscape of adversarial training: Identifying challenges and how to overcome them.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2019). Towards deep learning models resistant to adversarial attacks.

Moosavi-Dezfooli, S.-M., Fawzi, A., Fawzi, O., and Frossard, P. (2017). Universal adversarial perturbations.

Moosavi-Dezfooli, S.-M., Fawzi, A., and Frossard, P. (2016). Deepfool: a simple and accurate method to fool deep neural networks.

Neekhara, P., Dolhansky, B., Bitton, J., and Ferrer, C. C. (2020). Adversarial threats to deepfake detection: A practical perspective.

Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. (2015). The limitations of deep learning in adversarial settings.

Shiva28 (2022). Generating targeted adversarial attacks and assessing their effectiveness in fooling deep neural networks. GitHub repository; viewed on 2025-11-08.

Tolosana, R., Vera-Rodriguez, R., Fierrez, J., Morales, A., and Ortega-Garcia, J. (2020). Deepfakes and beyond: A survey of face manipulation and fake detection. Technical report.

Tsipras, D., Santurkar, S., Engstrom, L., Turner, A., and Madry, A. (2019). Robustness may be at odds with accuracy.

Weiss, M.-A. (2019). Deepfakes, fake news, mauvais sosies et fausses nouvelles. *Revue Francophone de la Propriété Intellectuelle*, 9.

Wiyatno, R. and Xu, A. (2018). Maximal jacobian-based saliency map attack.

Xu, Y., Deng, B., Wang, J., Jing, Y., Pan, J., and He, S. (2022). High-resolution face swapping via latent semantics disentanglement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.