Rapport de stage Oxford

Louis Maestrati
École Centrale de Lille
Cité Scientifique, 59650 Villeneuve D'ascq
louis.maestrati@centrale.centralelille.fr

1 Présentation du stage et de son déroulé

J'ai effectué mon stage de fin de 3ème année au Mathematical Institute de l'Université d'Oxford. Au vu de la situation particulière de cette année (Covid-19), j'ai effectué l'intégralité de mon stage en télétravail. Mon travail était encadré par le professeur Jared Tanner, professeur de Mathématiques à l'Université d'Oxford, avec lequel je faisais des réunions hebdomadaires. Mon partenaire de travail de tous les jours était Constantin Puiu, effectuant son doctorat en Optimisation à l'Université d'Oxford.

Durant le stage, j' ai étudié la robustesse des réseaux neuronaux profonds en réponse à des méthodes de sparsification et de régularisation. Notre but était d'aboutir à un modèle qui soit robuste à des attaques adversarielles, disposant ainsi de faculté de généralisation, mais surtout rendant le réseau apte à résister à l'ajout de bruits sur ses entrées (nommés exemples adversarielles), pouvant nuire à la qualité de ses prédictions. Partant de l'article "State of Sparsity" (Gale [2012]), j'étudiais l'influence de ces méthodes et de l'architecture des réseaux employés.

Mon rôle fut à la fois un rôle de réflexion: j'ai lu, réfléchis et participé activement avec mon partenaire de travail sur les axes de recherche à explorer au cours du stage , mais également un rôle de codeur. J'ai notamment codé de manière totalement indépendante, entre autres choses:

- le modèle Alexnet
- l'attaque adversarielle FGSM
- la low-rank regularization
- l'entraînement à haute température

Ce travail de codage fut très challengeant car je suis partis du code du papier de Gale [2012], extrémement long et complexe à prendre en main, dans lequel aucune de ces méthodes n'était implémentée. Ce code était d'autre part entièrement écrit en Tensorflow que je n'avais jamais exploré, ou encore que très brièvement dans un cours de cette année à Centrale. Il a fallu que je suive des cours en ligne en début de stage pour rapidement pouvoir me mettre à travailler.

Pour finir, j'ai participé activement à l'entraînement des modèles et la mise en forme des résultats. Chaque semaine nous cherchions à rendre une présentation propre de ces derniers au professeur Tanner afin que celui-ci soit dans les meilleures dispositions pour nous aiguiller sur les prochaines étapes à suivre.

2 Introduction aux réseaux neuronaux convolutionnels

Cette section vise à introduire les réseaux de neurones aux lecteurs peu familiers avec ces concepts. Elle peut être aisément passée si le lecteur les a déjà explorés. En premier lieu, nous allons introduire le réseau neuronal le plus simple, le perceptron, et expliquer la motivation des réseaux de neurones.

En second lieu, nous décrirons des réseaux particuliers: les *Convolutional Neural Networks* ou réseaux de neurones convolutionnels.

2.1 Les réseaux neuronaux

2.1.1 Structure

Les réseaux de neurones, parfois appelés réseaux de neurones artificiels, sont des estimateurs dont l'architecture est fondée sur nos connaissances des réseaux de neurones biologiques dans notre cerveau. Les réseaux de neurones artificiels sont des graphes orientés pondérés. Les sommets d'un réseau sont organisés par couche, chaque couche recevant en entrée la sortie de la couche précédente. Chacun des sommets d'une couche réagit indépendamment à l'entrée qu'il reçoit de la couche précédente, la dernière couche étant appelée couche de sortie. Ce sont ces sommets qui sont appelés 'neurones'. Dans cette partie, nous présentons les réseaux de neurones appelés perceptrons, qui sont la forme la plus classique et utilisée des réseaux. Chaque neurone d'une couche est connecté à tous les neurones de la couche suivante (*fully connected layer*). Les neurones sont des formes affines, codés sous la forme de matrices de poids, et ces poids sont 'ajustés' au cours de l'entraînement par descente de gradient, afin de satisfaire une performance la plus haute possible du modèle. Nous reviendrons sur ce dernier point dans la section suivante.

Soit une couche C constituée de m neurones $\{N_j\}_{j\in[1,m]}$ et recevant une entrée de taille n $(x_1, x_2, ..., x_n)$. Chaque neurone N_j est associé à une matrice w_j de taille $(1, n)$, et un terme de biais b_j correspondants à une application affine. Passer par une couche revient ainsi à passer au travers de chaque neurone, et chacun d'entre eux applique à l'entrée de la couche $(x_1, x_2, ..., x_n)$ sa matrice de poids et son biais: $O_j = w_j^T \times (x_1, .., x_n) + b_j$. La sortie de chaque neurone O_j devient alors une valeur d'entrée de la couche suivante $(O_1, O_2, ..., O_m)$. Laissés tels quels, les réseaux ne seraient ainsi qu'une succession d'application affine, impropre à approximer des fonctions continues plus complexes, c'est pourquoi les sorties des neurones sont suivis d'une fonction d'activation ψ, choisis expressement non-linéaires, afin que la composition des opérations puisse approcher toute fonction continue. On dit alors que le neurone *s'active*, processus analogue à celui observé en biologie, où les neurones s'activent si leurs sorties franchissent un certain seuil (voir schéma d'un neurone en figure 1 et des exemples de fonctions d'activation en figure 2). La sortie de chaque neurone prend ainsi la forme:

$$O_j = \psi(w_j^T \times (x_1, .., x_n) + b_j)$$

$$O_j = \psi(\sum_{i=1}^{n} w_j[i]x_i + b_j)$$

Ce dernier point est sous-tenu par le théorème 1 d'approximation universelle. Toute la motivation de création des réseaux de neurones est basée sur ce théorème, nous garantissant l'approximation uniforme de toute fonction continue par un réseau neuronal, quand bien même celui-ci ne possède qu'une seule couche cachée (soit au moins 2 couches au total, une couche est dite cachée si elle n'est pas la couche de sortie).

Theoreme 1. *Soient $n, m \in \mathbb{N}$. Soit C un sous-espace compact de $\mathbb{R}^n$. Toute fonction f continue de C dans $\mathbb{R}^m$ peut être approximée par un réseau neuronal à une seule couche cachée à n neurones R_n. L'erreur uniforme $sup_{x\in C}(|f(x) - R_n(x)|)$ tendant vers 0, lorsque la taille de la couche $n \to +\infty$.*

On notera que ce théorème ne fournit néanmoins aucun ordre de grandeur sur la taille n (capacité) du réseau à utiliser pour obtenir une erreur uniforme donnée.

Une manière conventionnelle de représenter un réseau de neurones est de représenter chaque matrice de poids w_j (associée au neurone j de la couche) par des flèches reliant l'entrée correspondante au neurone (voir figure 3). Le noeud représente l'activation du neurone, où sont sommées les entrées pondérés par les poids et où la fonction d'activation s'applique.

Chaque couche du réseau peut ainsi se voir comme une "grande" application linéaire suivis d'une fonction d'activation et si l'on note z de taille (1,n) l'entrée de la couche et f sa fonction correspondante, la sortie de la couche s'écrira:

$$f(z) = \psi(W^T \times z + b)$$

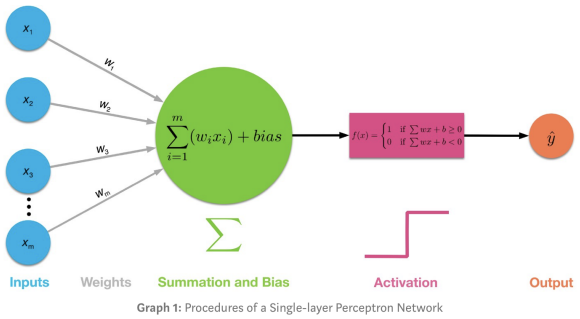

Figure 1: Schéma d'un neurone, tiré du blog de Saumya Ranjan Giri

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

Figure 2: Exemples de fonctions d'activation tirés d'un article de Matthew Stewart

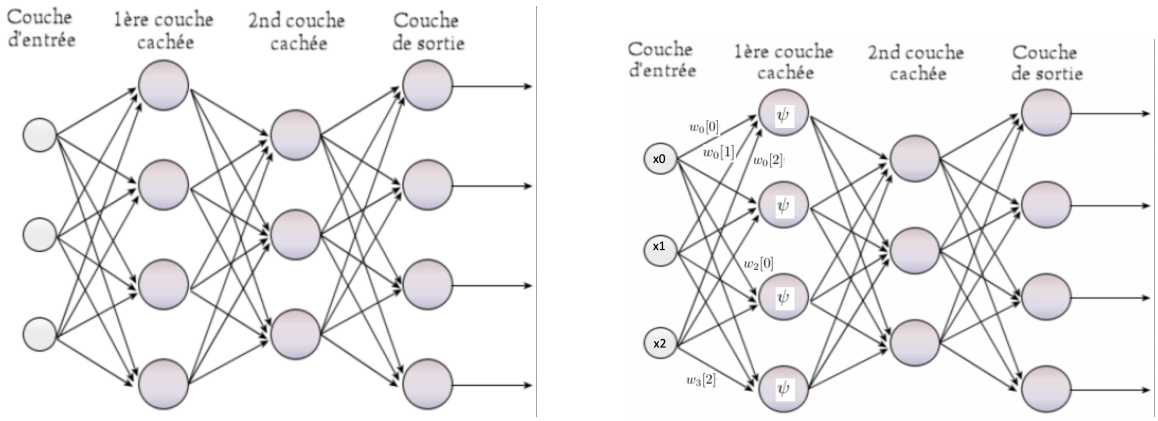

Figure 3: **1ère figure**: Schéma classique d'un Perceptron à 2 couches cachées tiré d'un article de theconversation, **2ème figure**: Schéma annoté par nos soins sur la première couche avec les poids correspondants

où $W = [w_1, w_2, ..., w_m] \in \mathbb{R}^{n \times m}$ est la matrice de poids de la couche, $b = [b_1, b_2, ..., b_m] \in \mathbb{R}^m$ son biais, et où m est le nombre de neurones de la couche

La réseau entier, à d couches, correspond ainsi à la formule:

$$f(x) := \psi_d\big(W_d^T \psi_{d-1}(...(\psi_1(W_1^T x + b_1))) + b_{d-1}\big) + b_d\big) \tag{1}$$

où $f(x) \in \mathbb{R}^{n_d}$ est la sortie du réseau, $\{W_i \in \mathbb{R}^{n_{i-1} \times n_i}\}_{i=2}^{d}$, $W_1 \in \mathbb{R}^{n \times n_1}$ les matrices de poids des couches, $\{b_i \in \mathbb{R}^{n_i}\}_{i=1}^{d}$ les biais, n_i est le nombre de neurones de la couche i, et n est la taille de l'entrée x du réseau.

2.1.2 Apprentissage

Dans cette sous-partie nous détaillons la méthode d'apprentissage des perceptrons, que l'on peut généraliser aisément à tout réseau de neurones, si tant est que l'apprentissage se fait par descente de gradient (méthode de très loin la plus classique et qui est utilisée dans tout notre travail). Toute tâche d'apprentissage requiert tout d'abord une métrique, ou encore une fonction de perte que l'on cherche à minimiser. Celle-ci correspond à la perte engendrée par ce que l'on obtient en sortie du réseau (en couche de sortie) vis à vis de ce que l'on cherche à obtenir. Dans le cadre de notre travail, on ne s'intéresse uniquement qu'à l'apprentissage supervisé, c'est à dire que l'on connaît ce que l'on souhaite obtenir comme sortie de notre réseau. On dispose ainsi d'un dataset de données "étiquetées" par les résultats que l'on souhaite obtenir. Prenons le cas qui nous intéressera par la suite, celui de la classification d'images: on dispose d'un ensemble d'images $\{x_i\}_{i \in D}$ (qui sont nos données) que l'on a étiquetées par leur classe d'appartenance $\{y_i\}_{i \in D}$. L'ensemble $\{x_i, y_i\}_{i \in D}$ nous fournit une base de données d'apprentissage, à partir de laquelle on va apprendre à notre réseau à reconnaître ces classes $\{y_i\}_{i \in D}$. En pratique, le réseau prend une image x en entrée et renvoie un vecteur de probabilité $p(x)$ d'appartenance aux classes.

Ce vecteur de probabilité est obtenu en sortie de la dernière couche du réseau qui possède comme fonction d'activation la fonction $softmax$. Cette fonction d'obtenir un vecteur dans $[0, 1]^c$, où c est le nombre de classes d'images , et dont la somme des termes vaut 1. Il s'agit donc bien d'un vecteur de probabilité sur les classes. En reprenant les notations de (1), la sortie de notre réseau à d couches peut s'écrire:

$$p(x) = \sigma(W_d^T f(x)) + b_d \in \mathbb{R}^c \tag{2}$$

où $W_d \in \mathbb{R}^{n_{d-1} \times c}$, $b_d \in \mathbb{R}^c$ et où σ est la fonction d'activation softmax.
On remarquera notamment que cela impose au réseau de posséder une couche de sortie possèdant $n_d = c$ neurones.

On note g(x) la valeur pré-activation de la couche de sortie:

$$g(x) = W_d^T f(x) \in \mathbb{R}^c \tag{3}$$

$$p(x) = \sigma(g(x)) \tag{4}$$

Les composants du vecteur g(x) sont appelés les **logits**, la fonction d'activation σ est quant à elle de formule:

$$\sigma_k(g(x)) := \frac{\exp(g_k(x))}{\sum_{i=1}^{c} \exp(g_i(x))}, \qquad \forall k \in \{1, 2, ..., c\}, \tag{5}$$

$$\sigma(g(x)) = (\sigma_1(g(x), \sigma_2(g(x)), \ldots, \sigma_c(g(x)))$$

Les étiquettes y_i sont quant à elles les vecteurs de probabilité 'certains' de forme (0,0,0,..,1,0,0,...,0), où 1 est en position j si la classe de l'image est la j-ème classe. Il s'agit ainsi, pour le réseau, d'apprendre à renvoyer pour chaque image x_i un vecteur $p(x_i)$ égale à y_i. Cet apprentissage s'effectue en différentiant notre fonction de perte par rapport à chaque poids du réseau, puis à modifier ces poids par retour de gradient de cette fonction de perte. Du fait de la forme 'par couches mises en série' du réseau, cette différentiation est aisée. Il s'agit d'appliquer la régle de différentiation à la chaîne. Un réseau n'étant rien d'autre qu'une succession de couche (donc de neurones) paramétrés par des poids, on peut différentier notre fonction de perte suivant l'ensemble de ses poids en appliquant la règle:

$$\frac{df}{dX} = \frac{df}{dY} \times \frac{dY}{dX}$$

Il est ainsi possible de calculer la dérivée de notre fonction de perte L par rapport à chaque neurone (w_j, b_j) de notre réseau, et ainsi ajuster nos paramètres par descente de gradient:

$$w_j \leftarrow w_j - \eta \times \frac{dL}{dw_j}$$

$$b_j \leftarrow b_j - \eta \times \frac{dL}{db_j}$$

où η est appelé 'learning rate' et ajustable au cours de l'entraînement (en pratique, on le rend de plus en plus petit au cours de l'entraînement afin de converger plus facilement). Dans le cas de notre exemple de classification d'image, la fonction de perte -autrement appelé **loss**- utilisée est la 'cross-entropy', de formule:

$$L = -\sum_{i \in D} \sum_{k=1}^{c} y_i[k] log(p(x_i)[k]) \tag{6}$$

où $p(x_i)$ est le vecteur de probabilité prédit par notre réseau ayant pour entrée l'image x_i.

Cette somme peut se simplifier en enlevant les termes nuls. Si l'on note c_i la classe de l'image x_i, il vient $y_i[c_i] = 1$ et $y_i[k] = 0 \; \forall k \neq c_i$. La loss se réécrit:

$$L = -\sum_{i \in D} log(p(x_i)[c_i]) \tag{7}$$

où c_i est la classe de l'image x_i

On remarque que cette fonction de perte est positive et vaut 0 si chaque image x_i a été correctement classée par notre réseau avec une probabilité certaine ($p(x_i)[c_i] = 1$, soit $p(x_i) = y_i$). On remarque également qu'elle est fonction de 3 variables: les données $\{x_i, y_i\}_{i \in D}$, mais également les paramètres $\{w_j, b_j\}_{j \in [1,d]}$ (souvent noté θ dans la suite)

Ce procédé rencontre souvent des problèmes tels que le bien connu problème de "*vanishing gradient problem*": lorsque le réseau possède un nombre élevé de couches, le gradient de notre fonction de perte est partagé par un grand nombre de poids, entraînant des termes $\frac{dL}{dw_j}$, $\frac{dL}{db_j}$ très petits (rendant notre descente de gradient peu efficace), d'où ce terme de 'vanishing'. Il est alors possible d'augmenter notre learning rate et de nombreuses méthodes sont également employés pour lutter contre ce phénomène (drop-out, Batch-Normalisation etc..), nous ne détaillerons pas ces méthodes, le lecteur intéressé pourra se référer au *Deep Learning Book* (Goodfellow et al. [2016]).

Il reste à préciser quelle méthode est appliquée pour effectuer de la prédiction de classe une fois le réseau entraîné. Différentes options sont possibles. Certains choisissent de tirer aléatoirement la classe de l'image d'entrée x suivant la distribution du vecteur p(x) (4). Dans notre cas et ce dans toute la suite, la classe prédite est celle de plus grande probabilité:

$$\hat{y}(x) = indmax(p(x)) \tag{8}$$

où $\hat{y}(x)$ est la classe prédite par le modèle pour l'image x et où indmax est la fonction renvoyant l'indice (en commençant à zéro!) correspondant à la valeur maximale d'un vecteur (indmax((0.1, 0.5,0.4))=1).

2.2 Les réseaux neuronaux convolutionnels

Les *convolutional neural networks* (CNN) sont des variantes aux réseaux classiques, particulièrement adaptés à l'analyse d'image. Ils se sont avérés hautement performants dans la classification d'images, on citera notamment des datasets connus comme MNIST (dataset de chiffres, référence ici), CIFAR-10 (dataset d'objets ou animaux, référence ici) ou encore Imagenet (dataset extrémement riche avec plus de 14 millions d'image et plus de 20 000 classes, référence ici) dont tous les modèles state-of-the-art font usage de CNNs. Ils servent également à prédire des quantités continues, on peut penser notamment à l'imagerie médicale et aux algorithmes prédisants avec un certain succès un état d'avancement d'une tumeur.

Une image est une matrice tri-dimensionnelle constituée de pixels, elle est de taille $H \times V \times D$, où H est la taille horizontale de l'image, V sa taille verticale et D sa profondeur (ou nombre de channels). Une image en couleur possède ainsi une profondeur D égale à 3 (3 channels: R,G et B) tandis qu'une image en échelle de gris ne possède qu'une profondeur de 1.

Au contraire des perceptrons, où chaque neurone est une fonction affine suivie d'une fonction d'activation, un neurone est ici un filtre, autrement appelé *kernel* qui est appliqué le long de l'image. L'idée des CNNs est d'appliquer aux images ces kernels dont l'application nous renseigne sur l'information que l'on peut obtenir localement sur une image. C'est en quelque sorte ce que font naturellement nos yeux lorsqu'ils observent une image: on passe nos yeux le long de l'image et observe localement l'information, avant de se faire une idée globale de ce que l'image représente. L'application d'un kernel consiste à multiplier 2 à 2 chaque pixel de notre image par la valeur du kernel correspondant, puis à sommer ces valeurs afin d'obtenir une valeur d'entrée à notre image de sortie (voir figure 4, où $H = V = 5$ et $D = 1$). La sortie du neurone/kernel est ainsi une nouvelle image, de taille réduite. Le kernel est défini par deux paramètres d'importance, sa taille et son paramètre de déplacement (autrement appelé 'stride'). Ce dernier est le saut effectué par le filtre le long de l'image, après chacune de ces applications. La taille du kernel est toujours de la forme $k \times k \times D$ où D est la profondeur de l'image auquel il s'applique.

En figure 4, l'image est mono-channel (D=1), la taille du kernel est $3 \times 3 \times 1$ ($k = 3$) et le stride est fixé à 2. L'image filtrée obtenue est ainsi de taille 2x2.

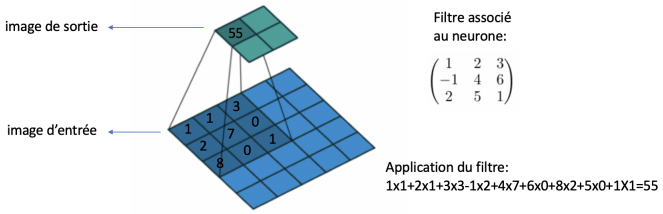

Figure 4: Application d'un kernel de taille $3 \times 3 \times 1$ à une image de taille $5 \times 5 \times 1$. Schéma de deeplearning.net et annoté par nos soins.

De la même manière que pour un réseau perceptron, les neurones convolutionnels peuvent posséder une fonction d'activation (appliquée à la nouvelle image obtenue après application du kernel).

Ils sont également placés par couche. Une image en entrée d'une couche sera filtrée autant de fois que la couche possède de kernels. On obtient donc au sein d'une couche autant d'images que de kernels appliqués. Chacune de ces images est prise comme channel de l'image de sortie de la couche, qui contient donc autant de channels que la couche contient de kernels. Dit différemment, les images sont concaténées suivant le paramètre de profondeur D et l'image de sortie a donc une profondeur D égale au nombre de kernels présents dans la couche. Les dimensions H et V de l'image sont également modifiés par la taille des kernels:

$$H \leftarrow \left\lfloor \frac{H-k}{s} \right\rfloor + 1$$
$$V \leftarrow \left\lfloor \frac{V-k}{s} \right\rfloor + 1$$
$$D \leftarrow N$$

où k est la largeur des kernels, s le stride et N le nombre de kernels de la couche (voir figure 5 pour un exemple).

Une couche convolutionelle possède donc une matrice de poids (regroupant ses kernels) de taille totale $N \times k \times k \times C$, où N est le nombre de kernels de la couche ($D = N$ après application des kernels à l'images), k la largeur des kernels et C le nombre de channels de l'image d'entrée ($D = C$ avant l'application des kernels à l'image).

Le même exemple que celui de la figure 5 mais cette fois dynamique est trouvable ici.

Dans le cas de la classification d'image, les couches de CNNs sont toujours suivis de couches perceptrons, afin d'obtenir en sortie du réseau un vecteur de probabilité d'appartenance aux classes.

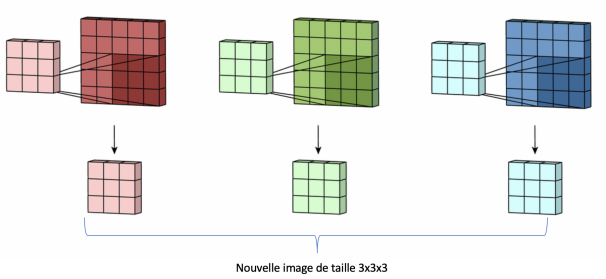

Figure 5: Application de 3 kernels de taille $3 \times 3 \times 1$ avec pour stride s=1, on obtient en sortie une nouvelle image de taille $3 \times 3 \times 3$ ($3 = \frac{5-3}{1} + 1$)

L'image de sortie des couches de convolution est ainsi 'aplati' (on dit 'flatten' dans la littérature) en un vecteur de taille $(1, H \times V \times D)$ pour être appliqué au perceptron possédant une entrée de taille correspondante. Dans la figure 6, on observe un modèle de classification composé de deux couches de convolutions, suivis d'un perceptron mono-couche. En sortie du réseau on obtient les probabilités d'appartenance aux différentes classes de véhicules (voiture, camion etc..). Le *pooling* est une technique de réduction de dimension consistant à ne conserver que certaines valeurs de l'image suivant un certain soeuil. En figure 7 est présenté un exemple: le max-pooling, consistant à ne garder que les valeurs maximales des images suivant des fenêtres d'application $m \times m$ (ici m=2).

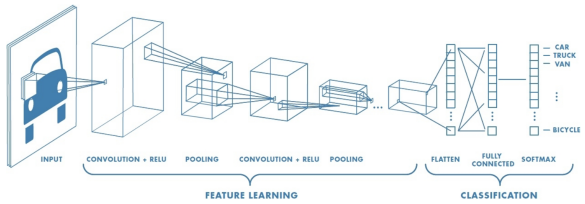

Figure 6: Exemple de réseau de classification: classification d'images de véhicules. Schéma tiré de mathworks.com

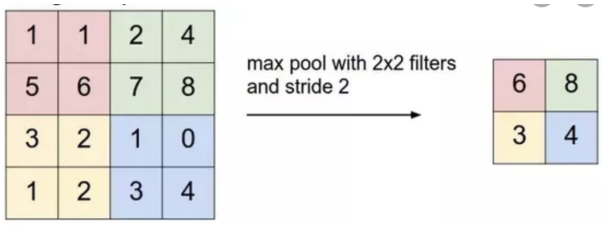

Figure 7: Application du max-pooling avec un paramètre de pooling m=2 et un stride (paramètre de déplacement) égal à 2. Schéma tiré de mc.ai.com

3 La robustesse et les attaques adversarielles

Dans la plupart des modèles d'intelligence artificielle les fonctions que l'on cherche à apprendre varient énormément sur l'ensemble des données d'apprentissage. Considérons par exemple le cas d'un modèle de classification: il s'agit de classifier des images , pourtant très différentes et ne se trouvant pas nécessairement dans un même voisinage, en une même classe (Y. [2016], Szegedy [2014]). Imaginez par exemple une image d'un chien: on peut y appliquer des opérations aussi diverses que rotations, translations et zoom, notre modèle sera toujours censé classifier l'image en tant que chien; et pourtant les images vu sous l'angle de leurs pixels se trouvent dans des ensembles disjoints très éloignés. Cette *sensibilité* se traduit en des modèles qui ne sont pas lisses et dont la sortie peut changer par un simple ajout de bruit à l'entrée. En d'autres termes, de petites perturbations peuvent résulter en des classes prédites complétements différentes. Ces perturbations, si on les

choisit expressement afin de tromper l'algorithme - pourtant imperceptibles à l'oeil humain- sont ce que l'on appelle les **perturbations adversarielles** dont l'addition aux images forment les **exemples adversarielles** (Szegedy [2014]). Enfin, on nommera l'application du modèle à un tel exemple une **attaque adversarielle**. La résistence à ces attaques est ce que l'on dénomme la **robustesse** des réseaux.

Dans ce rapport nous allons nous intéresser aux moyens que l'on peut mettre en oeuvre pour favoriser la robustesse d'un réseau: la *sparse regularization*, la *low-rank regularization*, ainsi que l'entraînement à haute température. En premier lieu nous introduirons plus formellement la notion de robustesse et nous décrirons comment générer des attaques adversarielles. Ensuite nous introduirons le *Magnitude Prunning*, la méthode la plus utilisé de *sparse regularization*, pour enfin s'intéresser à la *low-rank regularization* et l'entrainement à haute température. Nous avons effectués nos tests au travers de différentes architectures: 2 réseaux convolutionnels -à savoir le Resnet18 (He [2015]) et l' Alexnet (Krizhevsky [2012])- et un réseau perceptron à 5 couches. Le dataset utilisé est le dataset Imagenette (qui est une sous partie du bien connu ImageNet).

3.1 Robustesse

Initiallement, il fut hypothétisé que la présence d'exemples adversarielles était dû à la grande complexité des réeaux de neurones. Cependant, de récents travaux sur la compréhension des frontières de décisions et sur les "linearization based-attacks" ont révélé que c'est surtout la linéarité excessive des réseaux qui cause cette vulnérabité aux attaques adversarielles. Intuitivement, changer chaque pixel de l'image d'entrée, par une quantité pourtant très faible mais choisis dans une direction bien choisis, peut résulter en un changement très important en sortie du réseau. Plus formellement, on peut faire usage des constantes de Lipschitz pour évaluer la déviation de la sortie du réseau f(x) en fonction de la perturbation que l'on opère sur l'entrée x. Reprenons les notations de l'équation (1), pour un réseau à d couches, la sortie du réseau $f(x) := \psi_d\big(W_d^T \psi_{d-1}(...(\psi_1(W_1^T x + b_1))) + b_{d-1}) + b_d\big)$ est la compostion de d fonctions d'activations $\{\psi_i\}_{j=1}^d$ et d fonctions affines $f_j(z) = W_j^T + b_j$. Il vient:

$$\|f(x+\delta) - f(x)\| \leq L_f \|\delta\| \leq \|\delta\| \prod_{i=1}^{d} (L_{\psi_i} \times L_{g_i}) \leq \|\delta\| \prod_{i=1}^{d} \big(L_{\psi_i} \|W_i\|\big), \tag{9}$$

où L_f, L_{ψ_i}, L_{g_i} sont les constantes de Lipschitz associés à $f(.)$, $\psi_i(.)$, $g_i : x \longrightarrow g_i(x) = W_i^T \times x + b_i$.

En pratique $L_{\psi_i} \leq 1$ pour les fonctions d'activations classiques (ReLU, Max-Pooling), cependant $\|W_i\|$ a tendance a devenir relativement élevé (autour des valeurs 2, 5 et 11 pour Alexnet par exemple). Ainsi la constante de Lipschitz L_f explose exponentiellement en fonction de la profondeur du modèle. On voit ainsi poindre le fait que nous ne possédons aucune garantie sur le caractère "lisse" de f: de petites pertubations peuvent causer de grandes déviations dans la valeur de f(x), chose que l'on observe empiriquement (Goodfellow [2015],Moosavi-Dezfooli [2016], Goodfellow [2015]).

3.2 Attaques adversarielles

L'idée des attaques adversarielles est ainsi de "tromper" notre modèle en exploitant des déviations δ_x de faibles amplitudes, nous faisant franchir notre frontière de décision, et ainsi classifié incorrectement notre nouvelle entrée $x + \delta_x$, apparaissant pourtant à l'oeil nue comme identique à notre entrée d'origine x (Papernot [2016a]). Une attaque adversarielle cherche à trouver la plus faible perturbation δ_x^* provoquant une classification incorrecte:

$$\delta_x^* = \arg\min_{\delta_x} \|\delta_x\| \qquad \text{s.c.} \qquad \hat{y}(x + \delta_x) \neq \hat{y}(x). \tag{10}$$

où $\hat{y}$ est la fonction de prédiction introduite en équation .

Deux choses importantes sont à noter ici: la première est que l'on souhaite qu'un changement important (à l'oeil nu) sur l'image soit reflété par une norme $\|\delta_x\|$ également importante, la deuxième est que l'on souhaite une norme qui soit mathématiquement agréable et interprétable. Ces deux points nous poussent ainsi à choisir la norme infinie $\|\|_\infty$, qui présente l'avantage d' être "facilement repérable" et interprétable par l'oeil humain (une norme de 2 indiquera que chaque pixel a été modifié

d'au plus 2 unités). Dans toute la suite de ce travail, la norme infini sera ainsi utilisée. Le deuxième point à noter est qu'une attaque adversarielle semble dépendre de l'architecture du modèle, des hyperparamètres et des données d'entraînement. La réalité est autre: il fut démontré que des exemples adversarielles sont partagés par une multitude de modèles (Szegedy [2014], Papernot [2016b]), et qu'il était même possible d'entraîner des modèles à attaquer d'autres modèles, et ce avec grand succès (Papernot [2016b]).

Résoudre 10 est difficile en pratique, en ce qu'il s'agit de trouver le plus court chemin -dans un espace de très haute dimension- nous permettant de franchir la frontière de décision. On peut ainsi chercher à trouver une solution approchée, sous forme de séquence de problèmes d'optimisation:

$$\delta^*_{x,k} = argmax_{\delta_{x,k}} L(x + \delta_{x,k}, y, \theta) \text{ s.c. } \|\delta_{x,k}\|_\infty \leq M_k \tag{11}$$

où$\{M_k > 0\}_{k=1,2,...}$ est une suite croissante; et ainsi prendre comme solution δ^*_x la plus petite valeur $\delta^*_{x,k}$ telle que $\hat{y}(x + \delta_x) \neq \hat{y}(x)$. En d'autre terme, il s'agit de se placer succcessivement dans un voisinage de x, d'y chercher la valeur de perturbation nuisant le plus à notre loss, puis observer si cette perturbation a causé une classification incorrecte, si ce n'est pas le cas, on reproduit l'expérience dans un voisinage plus grand etc... L'idée derrière cette approche est bien sûr que l'augmentation de notre loss provoque une classification incorrecte. C'est bien le cas dans notre approche, où la fonction de perte choisis est la cross-entropie (7); car on a $L(x, y, \theta) = -log(p(x)[c_x])$ qui est bien une fonction croissante de la probabilité de la classe de x. Maximiser la loss revient bien ainsi à minimiser cette probabilité et ainsi traverser notre frontière de décision.

3.3 Les attaques FGSM

Résoudre (11) demeure computationellement difficile. Il faut bien avoir en tête que ce processus d'optimisation sera répété autant de fois que l'on possède d'images à attaquer, ce qui représente à terme un très grand nombre d'opérations. Cette équation peut se simplifier par un développement de Taylor au premier ordre. On a ainsi:

$$L(x + \delta_x, y, \theta) = L(x, y, \theta) + \delta_x^T \nabla_x L(x, y, \theta) + \Theta(\|\delta_x\|^2)$$

La solution optimale de l'équation (11) peut ainsi s'approximer par:

$$\delta^*_{x,k} = argmax_{\delta_{x,k}} \delta^T_{x,k} \nabla_x L(x, y, \theta) \text{ s.c. } \|\delta_{x,k}\|_\infty \leq M_k$$

dont la solution est classiquement $\delta^*_{x,k} = M_k \times sign(\nabla_x L(x, y, \theta))$. Ainsi, résoudre (10) ne revient plus ici qu'à calculer le gradient $\nabla_x L(x, y, \theta)$ et à poser successivement $\delta^*_{x,k} = M_k \times sign(\nabla_x L(x, y, \theta))$; la solution totale du problème étant ainsi $\delta^*_x = \epsilon \times sign(\nabla_x L(x, y, \theta))$ pour le plus petit $\epsilon > 0$ tel que $\hat{y}(x + \epsilon \times sign(\nabla_x L(x, y, \theta))) \neq \hat{y}(x)$. En pratique, on calcule le gradient $\nabla_x L(x, y, \theta)$ et on augmente epsilon jusqu'à observer un changement de classe. C'est cette méthode de résolution approchée que l'on nomme le **Fast Gradient Sign Method** (FGSM). Bien sûr, il ne s'agit que d'une méthode de résolution approchée, introduisant même plus d'erreur du fait de la discrétisation des valeurs ϵ testées mais il s'agit d'une bon moyen d'évaluation de la robustesse d'un réseau. En effet, plus cette valeur ϵ de changement de classe est élevée, plus le réseau est robuste, signifiant que les images possèdent des voisinages où le réseau continue à classifier corrrectement, celui-ci étant alors peu sensible aux bruits.
Toute nos expériences furent testées grâce à cette méthode FGSM. Nous tracerons ainsi la courbe d'accuracy du réseau en fonction de la norme infinie du bruit adversarielle (soit $\|\delta^*_x\|$, soit finalement ϵ au vu de l'approximation FGSM). Il s'agit d'un moyen classique d'étudier la robustesse d'un réseau et une méthode largement utilisée dans les papiers comme par exemple Papernot [2016b].

4 Les moyens mises en oeuvre pour favoriser la robustesse des réseaux

4.1 La sparsification et le Magnitude Prunning

4.1.1 La sparsification

Un réseau de neurone est dit "sparse" si un nombre important de ces poids peuvent être supprimés, c'est à dire que le réseau est rendu capable, au cours de son entraînement, d'effectuer sa tâche en utilisant qu'un nombre très réduit de ses poids. De nombreux travaux ont cherché à démontrer

que tout réseau de neurones peut être "élagué" sans pour autant perdre en performance, c'est la *Lottery Ticket Hypothesis* formulé par Jonathan Frankle [2018], qui stipule que tout réseau "cache" un sous-réseau de performance équivalente, mais au nombre de poids très réduits (on parle de conserver moins de 10% des poids). Concrétement, les méthodes de sparsification reviennent ainsi à ne conserver, au cours de l'entraînement, qu'une partie très faible des poids du réseau (fixant les autres à la valeur nulle), partant ainsi du principe que celui-ci ne nécessite plus qu'un nombre faible de poids pour accomplir sa tâche au bout d'un certain temps d'entrainement. Ces méthodes sont extrémement populaires car permettent de "compresser" les réseaux et de diminuer drastiquement les consommations énergétiques, qui sont un enjeu important en Deep Learning. La *sparsity* d'un réseau est le nombre de poids du réseau à valeur nulle. Un réseau qui possède une sparsity de 90% est un réseau dont 90% des poids sont nuls. Il existe tout un pan de recherche travaillant sur le lien entre le caractère "sparse" des réseaux de neurones et leurs robustesse (Justin Cosentino [2019]). Beacoup de chercheurs tendent à penser que ces deux concepts sont liés: si un réseau est capable d'accomplir sa tâche en usant que d'un nombre faible de poids, c'est que ces poids sont "assez expressifs et utiles", et c'est pourquoi il serait moins sensible aux attaques adversarielles. De grands nombres de méthodes de sparsification existent, on peut citer notamment le Variational Dropout (VD), l0-regularization et enfin le magnitude prunning. C'est cette dernière méthode qui nous a la plus intéressée, et ce pour deux raisons. La première, c'est que la plupart de résultats numériques présents dans la littérature font usage de méthodes en liens avec le Magnitude Prunning (MP) (Luyu Wang [2018], Ye [2018], Guo [2018]). La deuxième raison est que MP est bien plus facile à implémenter que les autres méthodes et surtout atteint quasi-systématiquement les même performances que les méthodes plus complexes comme VD (Gale [2012]). Notre approche étant focalisée sur l'obtention d'une robustesse par le biais de la sparsification, ces deux raisons nous ont poussés à préférer l'usage de MP.
Nous démontrerons, au travers d'une batterie de tests, que l'usage de MP peut, au contraire, éventuellement mener à une dégradation de la robustesse des réseaux, prenant ainsi le contre-pied de Guo [2018], affirmant que des réseaux sparses sont généralement plus robustes. Il ne s'agira pas d'affirmer qu'il n'y a pas de moyens de rendre un réseau plus robuste par sparsification, mais plutôt de montrer que sparsifier un réseau n'augmente pas systématiquement sa robustesse.

4.1.2 Magnitude Prunning

L'idée de MP est simple: affecter aux poids $\{W_j, b_j\}_{j=1}^{d}$ possédant la valeur absolue la plus faible, la valeur nulle, jusqu'à ce que la sparsity (nombre de poids nuls) désirée soit atteinte (Gale [2012], Zhu [2017]). Cette sparsification se fait progressivement, par étape, au cours de l'entraînement. Il est ainsi décidé à l'avance d'une étape de début de sparsification (t_{start}), d'une fréquence des étapes d'élaguage des poids $f_{\text{élaguage}}$ (étapes pendant lesquels les poids les plus faibles sont fixés à zéro), ainsi que de la sparsity finale désirée s_f. A chaque étape d'élaguage, un certain pourcentage (croissant) des poids sont affectés à la valeur 0. Entre chacune de ces étapes d'élaguage, le réseau continue à s'entraîner, de sorte qu'il lui est laissé le temps de "récupérer" de ce brutale changement de poids et de recouvrir une accuracy staisfaisante.
Un exemple pratique: on décide d'entraîner un réseau Alexnet avec pour but d'obtenir une sparsity finale de $s_f = 90\%$ avec pour paramètres $t_{\text{start}} = 1000$ et $f_{\text{élaguage}} = 100$. Pendant 1000 itérations, le réseau s'entraîne normalement et conserve une sparsity nulle, puis à la 1000-ième itération 50% de ses poids les plus faibles sont fixés à zéro, le réseau subissant par la même occasion une grande perte de performance (son accuracy sur le testset chute considérablement). Pendant 100 itérations, le réseau s'entraîne et récupère peu à peu son accuracy initiale, la deuxième étape d'élaguage est alors effectuée à la (1000+100)-ième itération et 20% suplémentaires de ses poids sont fixés à zéro, aboutissant à une sparisty de 70%... ceci ainsi de suite jusqu'à l'obtention de la valeur de 90%...
En pratique le nombre d'étapes d'élaguage $n_{\text{élaguage}}$ (et ainsi le pourcentage de poids fixés à zéro à chaque étape) est également fixé à l'avance. Toutes ces quantités sont reliés par l'équation du planning d'élaguage, que nous avons empruntés à Zhu [2017] et qui présente l'avantage d'être déjà accessible via Tensorflow. Posons $s_t \in [0, 1]$ la sparsity à l'itération t, le planning d'élaguage est le suivant:

$$s_t = s_f - s_f(1 - \frac{t - t_{\text{start}}}{n_{\text{élaguage}} * f_{\text{élaguage}}})^3 \text{ pour } t \in \{t_{\text{start}}, t_{\text{start}} + f_{\text{élaguage}}, ..., t_{\text{start}} + n_{\text{élaguage}} f_{\text{élaguage}}\}.$$

4.1.3 Résultats

Les résultats présentés sont pour 3 modèles le Resnet18 (He [2015]), modèle à 18 couches (particulièrement profond donc) et représentant les modèles à forte profondeur, le modèle Alexnet

qui est un modèle à 8 couches (Krizhevsky [2012]) et un modèle perceptron à 5 couches. Les deux premiers modèles font notamment usage de couches convolutionnels présentés dans la section 2.

Chacun de ses réseaux fut entraîné et sparsifié, à différents degrés, puis attaqués sur 1/5 du validation set (soient 785 images) qui est choisis à chaque attaque de manière aléatoire.

Comme expliqué en section 3.2, sont présentés dans les graphes l'accuracy du réseau, à différents degrés de sparsification $(0\%, 50\%, 80\%, 90\%, 95\%, 98\%)$, sur les exemples adversarielles FGSM pour différentes valeurs de ϵ. De manière alternative, on présente également la décroissance de l'accuracy du réseau suivant la valeur ϵ ($drop(\epsilon) = accuracy(\epsilon = 0) - accuracy(\epsilon = \epsilon)$), afin d'observer si cette décroissance est plus faible ou non, suivant le degré de sparsification. On s'attend ainsi à observer une décroissance de l'accuracy plus faible pour les modèles les plus robustes: la classe prédite étant alors moins affectée par l'exemple adversarielle $x + \epsilon \times sign(\nabla_x L(x, y, \theta))) \neq \hat{y}(x)$, où x est l'image initiale placée en entrée.

Résultats sur Resnet18

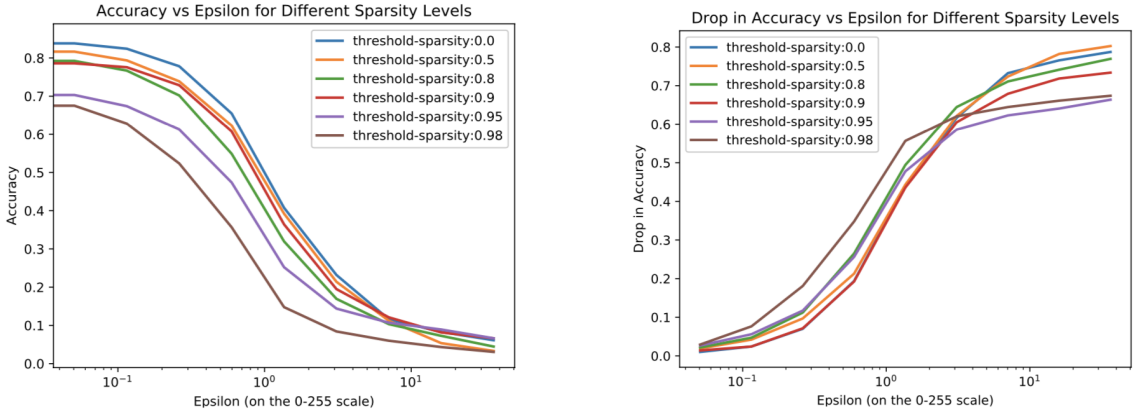

Figure 8: Figure **a**: Accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 512. Figure **b**: Décroissance de l'accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 512.

On observe en figure 8-b que la décroissance en accuracy est généralement plus importante pour des réseaux hautement sparsifiés. Ceci vient contredire l'hypothèse suivant laquelle un réseau hautement sparsifié est plus robuste. On observe néanmoins que, pour de grandes valeurs de ϵ, les réseaux fortement sparsifiés possèdent des décroissances en accuracy plus faibles, mais ceci s'explique avant tout par le fait que les performances de ces réseaux sont dèjà trop faible -autrement dit proche de l'aléatoire- comme on peut le voir en figure 8-a. Enfin, il est notable de voir dans cette même figure qu'un réseau possède systématiquement une accuracy plus faible qu'un réseau plus sparsifié, la figure 8-a montrant effectivement des courbes qui ne se croisent pas. Ainsi pas seulement la robustesse, mais également les performances intrinséques du modèle sont diminués par la sparsification par magnitude prunning. Nous avons fait l'hypothèse que ce phénomène puisse être dû à un batchsize choisis trop grand (512 ici), ce qui a pu mener à un sous-apprentissage, c'est pourquoi une nouvelle batterie d'expériences fut menée avec un batchsize plus petit.

En figure 9, les même expériences furent conduites mais cette fois ci avec un batchsize de 128. On observe cette fois qu'une plus grande sparsity mène à une robustesse légèrement supérieure, sans pour autant observer de résultats déterminants. La différence en décroissance d'accuracy reste tout de même extrémement faible et on ne peut conclure sur de relations de cause à effet étant donné que la méthode FGSM reste une méthode approchée d'identification de robustesse et est sujette aux bruits.

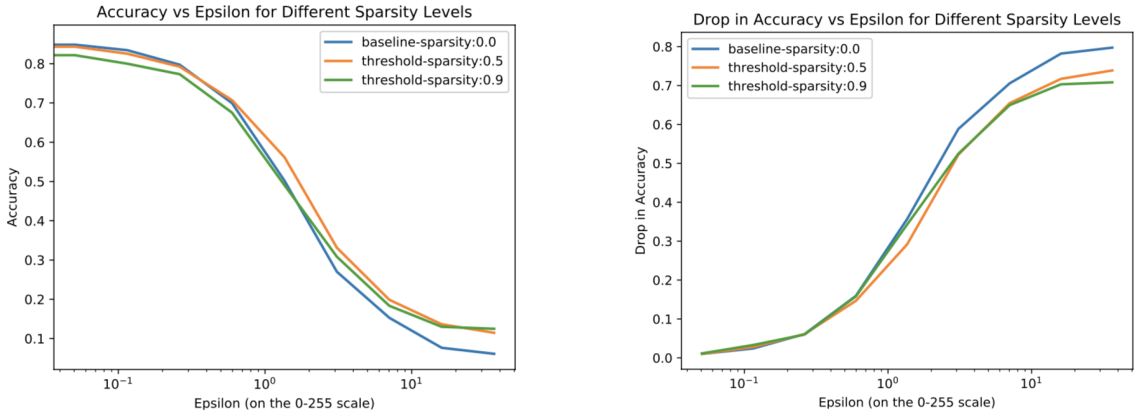

Figure 9: Figure **a**: Accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 128. Figure **b**: Décroissance de l'accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 128.

Résultats sur Alexnet

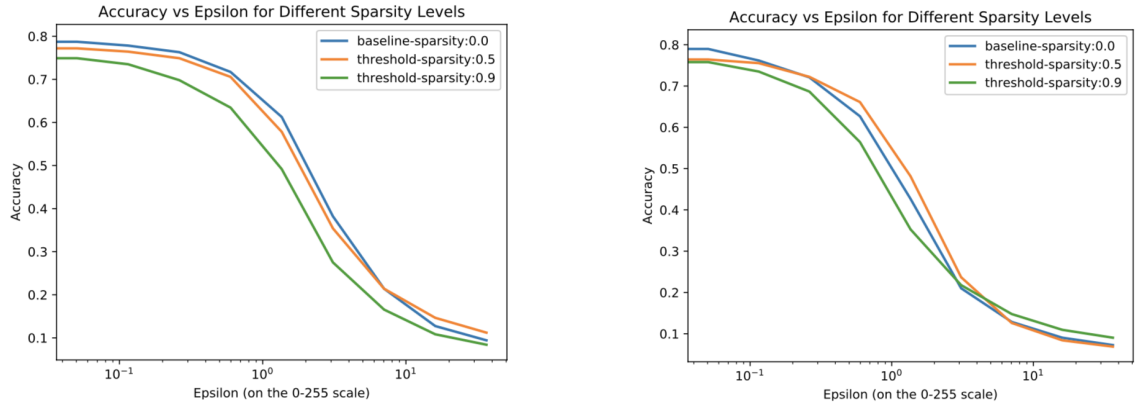

Figure 10: Figure **a**: Accuracy du Alexnet pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 128. Figure **b**: Décroissance de l'accuracy du Alexnet pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 512.

Nous avons également conduits des expériences sur le modèle Alexnet. Il s'agit d'un exemple de réseau convolutionnel à faible profondeur (5) et surtout ne possédant pas de bloques résiduelles, contrairement au Resnet18. Nous ne détaillons pas ici ce qu'est un bloque résiduelle mais le lecteur pourra se référer à He [2015]. Les courbes de l'accuracy semblent cette fois-ci plus claires et propres à tirer des conclusions. En figure 10-a, on observe que pour un batchsize de 128, augmenter la sparsity du réseau résulte en une robustesse plus faible. Au contraire, en figure 10-b, pour un batchsize de 512, on observe qu'une sparsity de 50% est plus robuste que le modèle baseline (à 0%), et que le phénomène opposé se déroule pour une sparsity de 90%.

Résultats sur le réseau fully-connected à 5 couches (FC5)

Poour terminer, nous souhaitions observer ce qui se passerait si nous n'utilisions aucune couche convolutionnelle dans le modèle, en utilisant un modèle à couches fully-connected (autrement appelé perceptron, voir 3 pour un schéma de ce type de couche et la partie 2 pour une présentation). La structure du réseau est la suivante:

- couche 1: 1344 neurones,
- couche 2: 672 neurones
- couche 3: 588 neurones

- couche 4: 294 neurones
- couche 5: 10 neurones

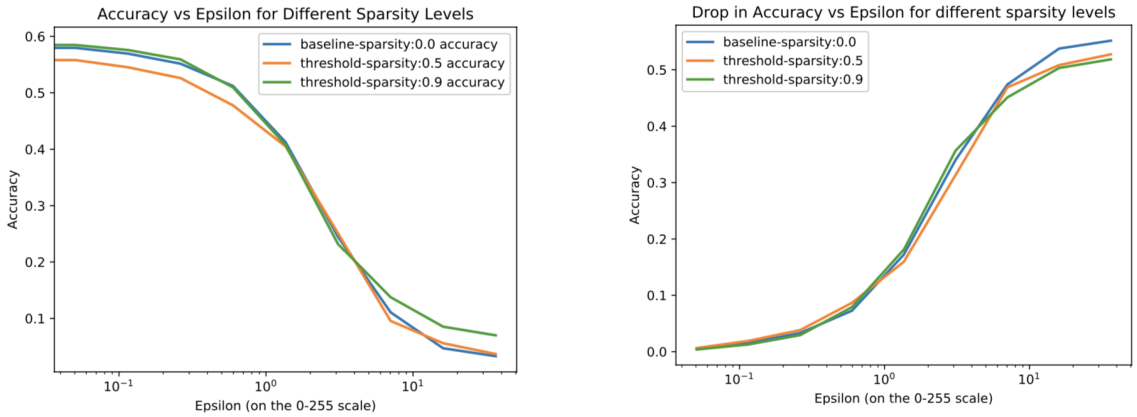

Figure 11: Figure **a**: Accuracy du FC5 pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 512. Figure **b**: Décroissance de l'accuracy du FC5 pour différentes valeurs de ϵ, suivant la méthode FGSM. Le batchsize utilisé est 512.

En figure 11, nous observons que la robustesse des réseaux n'est pas affectée par leur sparsity. Les différences observées peuvent tout simplement être dûes à du bruit.

Conclusion sur les résultats

Après avoir étudié plusieurs architectures, plusieurs batchsizes, et explorer diverses niveaux de sparsity, il est possible d'affirmer que sparsifier un réseau peut impacter de manière positive ou négative la robustesse de ce réseau. Nous avons en effet observer des réseaux sparses qui sont marginallement moins robustes que les réseaux non sparses, à quelques exceptions prêts (du moins en ce qui concerne les attaques FGSM). Ces résultats sont consistents avec d'autres de la littérature (Luyu Wang [2018], Ye [2018]) et suggère que si l'intuition de départ de Guo [2018] est correcte; stipulant que la robustesse et la sparsification sont deux concepts liés; c'est l'interprétation qui en est faite qui est trop attive. **Même si les réseaux robustes ont tendance à être sparses, sparsifier un réseau ne conduit pas nécessairement à un réseau robuste**.

4.2 La low-rank regularization

La motivation originelle de la low-rank regularization était d'augmenter la rapidité d'entraînement et d'usage des réseaux de neurones, tout en réduisant le coût en mémoire; ceci afin de rendre les CNNs utilisables pour les applications mobiles. Il s'agit ici d'étudier l'impact de cette méthode sur la robustesse des réseaux. En effet, un récent article (Langenberg [2019]) a démontré que l'adversarial training (qui consiste à entraîner le réseau avec des exemples adversarielles afin de gagner en robustesse) induit simultanément des matrices de poids sparses et surtout à faible rang. D'une manière différente, ils montrent également qu'user simultanément dans la loss du modèle la nuclear norm minimization (NNM) -pour induire des matrices de poids de faible rang- et la 1-Norm -pour augmenter la sparsity du modèle- améliore la robustesse du réseau, mais pas autant que l'adversarial training. De manière analogue à la section précédente, ce n'est pas parce que la version la plus robuste d'un modèle est à rang faible que les méthodes existantes pour limiter le rang d'un modèle permettraient d'obtenir le modèle le plus robuste. Ceci expliquerait pourquoi les auteurs n'ont pas obtenus les même robustesses lorsqu'ils ont artificiellement induit un rang faible dans leurs modèles, que dans le cas de l'adversarial training. Enfin, ceci pose la question de l'existence de méthodes de low-rank regularization qui mène à des solutions plus robustes que leur méthode. Nous avons ainsi tester l'effet de l'approche de Tai [2016]. Nous avons obtenu des résultats négatifs: cette méthode réduit la robustesse des CNNs et établit encore une fois le même constat: entraîner un modèle en le forçant à être de faible rang ne permet pas nécessairement d'obtenir une robustesse plus élevée quand bien même le modèle le plus robuste est à faible rang (comme démontré par l'adversarial training).

Il existe beaucoup d'approches de régularisation du rang des CNNs (Tai [2016],Denton [2014],Jaderberg [2014], Lebedev [2014]). Nous avons utilisé celle de Tai [2016] car celle-ci est capable de produire des modèles à faible rang sans introduire aucun effort computationel suplémentaire, tout en obtenant des modèles aux accuracy semblables à celles des modèles non régularisés. Cette méthode est basée sur la décomposition des matrice de kernels en matrices de rangs faibles. Plus précisemment, reprenons les notations de la section 2.2 et plaçons nous dans une certaine couche de notre réseau de neurone, associée à la matrice de kernels 4-dimensionelles $W \in \mathbb{R}^{N\times k\times k\times C}$, où N est le nombre de kernels de la couche, C le nombre de channels de l'image d'entrée de la couche, k la largeur des kernels de la couche. On peut voir W, comme expliqué en section 2.2, comme un tableau de kernels tri-dimensionnels $W = [W_1, W_2, \dots, W_n, \dots, W_N]$ où $W_n \in \mathbb{R}^{k\times k\times C}, \forall n \in \{1, \dots, N\}$.
Posons également $\mathbb{Z} \in \mathbb{R}^{X^{'}\times Y^{'}\times C}$ l'image d'entrée de la couche et $\mathbb{F} \in \mathbb{R}^{X\times Y\times N}$ l'image de sortie de la couche. Une fois encore W_n et les images $\mathbb{Z}, \mathbb{F}$ peuvent être perçus comme des tableaux suivant le paramètre channels (profondeur). On obtient ainsi $W_n = [W_n^1, \dots, W_n^C]$, $\mathbb{Z} = [\mathbb{Z}^1, \dots, \mathbb{Z}^C]$, $\mathbb{F} = [\mathbb{F}^1, \dots, \mathbb{F}^N]$ où $\{W_n^c\}_{c=1}^C \in \mathbb{R}^{k\times k}$, $\{\mathbb{Z}^c\}_{c=1}^C \in \mathbb{R}^{X^{'}\times Y^{'}}$ et $\{\mathbb{F}^n\}_{n=1}^N \in \mathbb{R}^{X\times Y}$.

L'opération de convolution $\mathbb{F}^n = W_n * \mathbb{Z}$ peut ainsi s'écrire:

$$\mathbb{F}^n(x,y) = \sum_{c=1}^{C}\sum_{x'=1}^{X^{'}}\sum_{y'=1}^{Y'} \mathbb{Z}^c(x',y')W_n^c(x-x',y-y') \ , \forall x \in \{1,\dots,X\}, y \in \{1,\dots,Y\} \quad (12)$$

où $\mathbb{Z}^c(x', y')$ représente le pixel de position (ligne x', colomne y', channel c) de l'image d'entrée $\mathbb{Z}$.

L'idée de la low-rank regularization est d'approximer W_n^c par un produit matriciel de rang borné:

$$\widetilde{W_n^c} = \sum_{d=1}^{K} V_d^c H_n^d \quad (13)$$

où $H \in \mathbb{R}^{N\times 1\times k\times K}$, $V \in \mathbb{R}^{K\times k\times 1\times C}$ sont des matrices 4-dimensionelles associées à des filtres horizontaux et verticaux respectivement (notez que $H_n^d \in \mathbb{R}^{1\times k}$ et que $V_d^c \in \mathbb{R}^{k\times 1}$).

L'image de sortie $\mathbb{F}$ est donc approximée par l'équation:

$$\widetilde{\mathbb{F}^n} = \widetilde{W_n} * \mathbb{Z} = \sum_{c=1}^{C}(\sum_{d=1}^{K} V_d^c H_n^d) * \mathbb{Z}^c = \sum_{d=1}^{K} H_d^n * (\sum_{c=1}^{C} V_d^c * \mathbb{Z}^c) \quad (14)$$

L'équation 13 permet de borner l'espace dans lequel évolue la matrice de poids W. En effet, à **n fixé** et **de manière indépendante à c**, chaque élément $\widetilde{W_n^c}$ évolue dans l'espace Vect $\{H_n^d\}_{d=1}^K$, qui est un espace de dimension maximale K. Ainsi l'ensemble $\{\widetilde{W_n^c}\}_{c\in[1,C],n\in[1,N]}$ est inscrit dans un espace de dimension maximale $N \times K$, le rang de la matrice $\widetilde{W}$ étant donc de rang maximal $N \times K$. Cette valeur est à comparer à celle de la matrice W qui est de rang maximale $N \times k \times k \times C$. L'équation 13 limite donc le rang de la matrice de poids convolutionelle à condition que $K < k^2 \times C$. C'est ainsi le paramètre K qui constituera l'hyperparamètre des expériences de low-rank regularization traîtées dans la partie résultats.

On remarque également que l'équation 14 permet de facilement implémenter cette méthode. En effet, comme l'indique le membre de droite de l'équation, la décomposition de la matrice W peut être perçue comme deux convolutions successives: une dite 'verticale' et de matrice V, l'autre dite 'horizontale' et de matrice H. Ainsi, pour obtenir une version à faible rang de notre réseau de neurone, chaque couche du réseau est remplacé par un doublet couche convolutionelle vertical-couche convolutionnelle horizontale, et le réseau est entraîné suivant ses nouveaux paramètres H et V. Cette méthode a pour inconvénient de doubler la profondeur du réseau qui peut souffrir d'un 'vanishing gradient'. Ce problème a été traîté en usant de la méthode de batch-normalization (Ioffe [2015]).

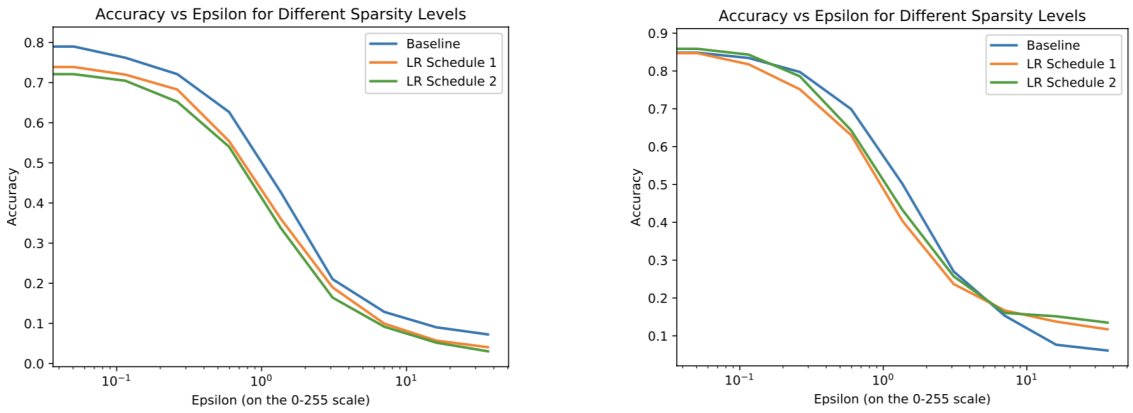

Figure 12: Figure **a**: Accuracy de l'Alexnet pour différentes valeurs de ϵ, suivant la méthode FGSM, et diverses LR-Schedule. Le batchsize utilisé est 512. Figure **b**: Accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM, et diverses LR-Schedule. Le batchsize utilisé est 128.

Résultats

On observe en figure 12, pour le modèle Alexnet et le modèle Resnet18, les résultats de la low-rank regularization pour deux 'LR-Schedule', c'est à dire pour deux hyperparamétrisations différentes de la méthode.
Pour le LR Schedule 1 (LR1) on a pris pour l'Alexnet (en partant de la première couche jusqu'à la dernière) les valeurs de K suivantes: $\{8, 40, 60, 100, 200, \text{non-contraint}\}$ et de manière similaire pour le Resnet18: $\{8, 5, 24, 24, 24, 48, 48, 48, 48, 64, 128, 160, 160, 192, 192, 256, 256, \text{non-contraint}\}$. 'non-contraint' signifiant que l'on a laissé la dernière couche (qui est fully-connected) non reparamétré par la méthode, c'est à dire libre. Pour le LR Schedule-2 (LR2) nous avons pris celles de LR1 divisées par 2. On peut tout d'abord observer en figure 12-a, comme on pouvait s'y attendre, que plus le rang de l'Alexnet est contraint, plus son accuracy baseline (c'est à dire à $\epsilon = 0$) est faible. Cependant aucune robustesse ne semble être observée, les trois courbes se suivant parfaitement lorsque ϵ varie. Au contraire de l'Alexnet, on peut voir pour le Resnet18 en figure 12-b, que les performances intrinséques du modèle (c'est à dire l'accuracy à $\epsilon = 0$) sont inchangées par la méthode de régularisation, les 3 courbes (baseline, LR1, LR2) débutant au même point. Cependant le modèle baseline demeure plus robuste que les modèles à faible rang (LR1 et LR2), même si les différences observées sont quasi négligeables.

Conclusion sur les résultats

Au vu de ces résultats, on peut tirer 2 conclusions. La première fut déjà exprimée précédemment dans cette section. De façon analogue au Magnitude Prunning, ce n'est pas parce que la version le plus robuste d'un modèle est à faible rang, que toute méthode de régularisation du rang permet d'approcher ce modèle robuste. La deuxième conclusion est que la méthode explorée par Langenberg [2019] offre de meilleur résultat qu'une méthode de décomposition. En effet, ce type de méthode (celle de Langenberg [2019]), qui fait usage de la norme nucléaire dans la loss du modèle, permet généralement de lisser les frontières de décision, contrairement à une méthode de décomposition matricielle, qui ne permet que de contraindre le rang.

4.3 L'entraînement à haute température

Dans cette dernière section, on étudie une méthode très différente de celles explorées jusqu'à présent. Rappelons nous dans la section 2, la forme de la dernière couche d'un réseau de neurones qui fait de la classification. Les 'logits' qui sont la sortie de la dernière couche du réseau sont ensuite renormalisés sous la forme d'un vecteur de probabilité. Si l'on a c classes, et un réseau à d couches on obtient ainsi:

La valeur pré-activation (ou logits) de la dernière couche:

$$g(x) = W_d^T f(x) \in \mathbb{R}^c \tag{15}$$

où $W_d \in \mathbb{R}^{n_{d-1} \times c}$, $f(x)$ est la sortie des (d-1) premières couches du réseau.

La valeur post-activation de la denière couche (ou vecteur de probabilité de sortie):

$$p(x) = \sigma(W_d^T f(x)) \in [0,1]^c \tag{16}$$
$$= \sigma(g(x)) \tag{17}$$

où $p(x) \in [0,1]^c$, et où σ est la fonction d'activation softmax, qui s'écrit:

$$\sigma_k(g(x)) := \frac{\exp(g_k(x))}{\sum_{i=1}^c \exp(g_i(x))}, \qquad \forall k \in \{1,2,...,c\}, \tag{18}$$

$$\sigma(g(x)) = (\sigma_1(g(x), \sigma_2(g(x)), \ldots, \sigma_c(g(x)))$$

Ce vecteur de probabilité nous donne ensuite acccès à la fonction de perte/loss du modèle, la cross-entropie. Celle-ci définit la perte engendrée par l'erreur commise entre la prédiction du modèle $p(x)$ et le vecteur étiquette $y \in \{0,1\}^c$ (contitué de la valeur 1 en position classe de x et de valeurs nulles). Pour un élément du dataset (x_i, y_i), la cross-entropie vaut:

$$L(x_i, y_i, \theta) = -\sum_{k=1}^{c} y_i[k] log(p(x_i)[k]) \tag{19}$$
$$= -\sum_{k=1}^{c} y_i[k] log(\frac{\exp(g_k(x_i))}{\sum_{j=1}^c \exp(g_j(x_i))}) \tag{20}$$
$$= -g_{c_i}(x_i) + log(\sum_{j=1}^{c} exp(g_j(x_i))) \tag{21}$$

où θ désigne l'ensemble des poids du réseau, dont dépend éffectivement la loss (puisque la sortie du réseau $p(x)$ en dépend), et où c_i est la classe de x_i (indice k de y_i tel que $y_i[k] = 1$).

L'idée derrière l'entraînement à haute température est simplement de diviser les logits par un nouvel hyperparamètre T (dénommé température). Il vient donc, après l'application de la fonction softmax:

$$L(x_i, y_i, \theta) = -\sum_{k=1}^{c} y_i[k] log(p(x_i)[k]) \tag{22}$$
$$= -\sum_{k=1}^{c} y_i[k] log(\frac{\exp(\frac{g_k(x_i)}{T})}{\sum_{j=1}^c \exp(\frac{g_j(x_i)}{T})}) \tag{23}$$
$$= -\frac{g_{c_i}(x_i)}{T} + log(\sum_{j=1}^{c} \exp(\frac{g_j(x_i)}{T})) \tag{24}$$

Cette température aura pour effet de "lisser" la distribution $p(x)$ produite par le modèle. Plus la température T est importante, plus les logits sont "écrasés". La fonction exponentielle étant super-linéaire, après application de la fonction softmax, le vecteur de probabilité produit se rapproche d'une distribution uniforme, le cas extrême où $T \to +\infty$ donnant $p(x) \to (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$. L'espoir de cette méthode est de contraindre la sortie réseau à ne pas trop se rapprocher des valeurs $y = (0, 0, \ldots, 1, 0, \ldots, 0)$, et ainsi d'introduire plus de doute sur les vecteurs de probabilité de sortie. Si cette approche peut paraître paradoxale, elle permet pourtant au réseau de capturer plus aisément les corrélations des classes.
Prenons un exemple concret, imaginons qu'un réseau cherche à classifier un dataset d'images d'animaux constitué de 3 classes {éléphant, chien, chat}. Imaginons un réseau qui prédit

systématiquement avec certitude (c'est à dire $p(x) \sim y = (0, 0, 1)$ par exemple) mais qui se trompe dans 20% des cas.
Imaginons ensuite un réseau qui est moins sûr de lui, car décèle une ressemblance entre chien et chat (avec des vecteurs de sortie du type (0.1, 0.4,0.5)). Comme vu en équation 3.2, la prédiction du modèle pour une image d'entrée x est indmax(p(x)) et peut donc être semblable pour un grand nombre d'images à celle du modèle sûr de lui (indmax((0.1, 0.4,0.5))=2= indmax((0,0,1)). Cependant, ce modèle moins sûr de lui n'échoue que dans 15% des cas... Une citation du livre *Madame Gervaisais* (1869) des frères Goncourts nous mène sur une piste d'explications: "*Le doute, lui, c'est la suspension de l'intelligence entre deux extrêmes qui offrent tous deux des raisons de probabilité...*". Le fait que le modèle soit moins sûr de lui permet en effet de déceler des ressemblances entre classes et de capturer des paramètres pourtant négligés par un modèle trop sûr de lui et qui sur-apprend la distribution... Ainsi, le paramètre T est un moyen d'effectuer de la régularisation des frontières de décision du modèle.

L'entraînement à haute température fut exploré par Hinton [2015] dans le contexte du "transferring knowledge", avec des objectifs de régularisation. Plus récemment, Papernot [2016b] explora cette méthode suivant le paradigme de teacher-student, méthode employant un réseau entraîné à haute température et enseignant à un réseau fils de nouveaux labels y. Elle fournissait des résultats convaincants sur la robustesse acquise par le réseau fils ainsi entraîné. Ces deux approches soulèvent la question si l'entraînement à haute température permet, à lui seul (c'est à dire sans le paradigme teacher-student), d'augmenter la robustesse d'un réseau.

Résultats On a testé cette approche avec différentes valeurs de température $T \in \{20, 50, 70, 90\}$ en comparant nos résultats avec le modèle classique où $T = 1$.

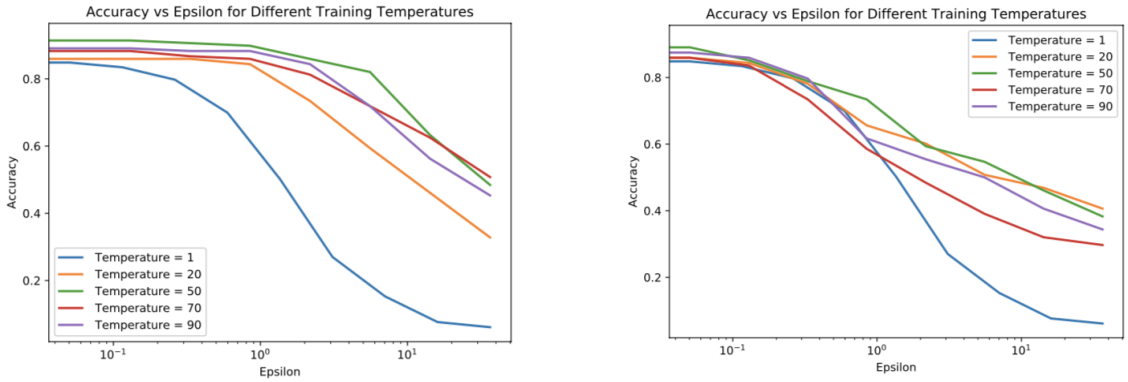

Figure 13: Figure **a**: Accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM. Diverses températures d'entraînement sont utilisées. L'attacker n'a pas connaissance de la température d'entraînement et fixe $T = 1$. Le batchsize utilisé est 128. Figure **b**: Accuracy du Resnet18 pour différentes valeurs de ϵ, suivant la méthode FGSM. Diverses températures d'entraînement sont utilisées. L'attacker a connaissance de la température d'entraînement et calcule le gradient correct. Le batchsize utilisé est 128.

L'équation d'obtention des attaques FGSM (voir section 3.3) faisant intervenir le gradient du modèle par rapport à son entrée, il dépend de la température T choisis. On distingue donc dans les deux courbes de la figure 13 deux interprétations possibles de la méthode FGSM: l'une où 'l'attacker' a connaissance de la température d'entraînement et calcule le gradient en fonction, l'autre où il n'en a pas connaissance et fixe la température à la valeur classique $T = 1$.
On observe en figure 13-b que l'entraînement à haute température augmente la robustesse du réseau pour de grandes valeurs de ϵ. Cette constatation est à moitié suffisante puisque la méthode d'attaque FGSM ne prévaut que pour des valeurs relativement faibles de ϵ. Cependant, un choix a été fait dans notre travail de considérer la robustesse en tant que résistance aux attaques FGSM (ceci du fait de nos ressources computationnelles limitées), et cette approche répond donc en partie à ce problème. La raison pour laquelle elle ne fonctionne pas pour de faibles valeurs de ϵ est certainement dû au fait que le FGSM n'approxime pas correctement la solution de l'équation (11) dans le cas des fortes températures.

Dans le cas de la figure 13-a, on observe une robustesse apparente pour toutes les valeurs de ϵ. La raison apparente est que l'attacker ne connaît pas la température d'entraînement du modèle, son attaque FGSM n'approxime donc que très mal l'équation (11) et la direction du gradient ainsi calculé n'endomage donc pas l'accuracy du modèle.

Conclusion sur les résultats Ainsi, si le fait que cette méthode permette d'obtenir des modèles robustes restent à prouver, elle constitue un excellent mécansime de défense aux attaques FGSM. Un attacker malveillant souhaitant duper un modèle prédictif, n'ayant pas accès à la température du réseau, peut voir ses tentatives échouer. Si cet attacker arrive à deviner la température du réseau, on observe malgré tout que cette méthode reste efficace pour de larges valeurs de ϵ.

5 Conclusion

Ce stage fut extrémement enrichissant. D'abord d'un point de vue technique, j'ai acquis des connaissances solides en programmation tensorflow. Les divers modèles et tâches que j'ai eu à affectuer m'ont en effet rendu très à l'aise avec ce langage. Ce dernier est extrémement utile car perçoit un réseau de neurone comme un graphe orienté d'instructions, le lancement de ce réseau se faisant par l'intermédiaire d'une "session" de calcul, activant les noeuds du graphe. Ce formalisme s'est avéré très utile lorsqu'il s'est agit de calculer des gradients "exotiques", comme le gradient du réseau de neurone vis à vis de son image d'entrée, tâche qui est loin d'être aisé dans un autre langage de programmation. Il fut également très utile lors de l'entraînement du réseau. Grâce à l'outil tensorboard, nous pouvions observer facilement, et ce en même temps que l'entrainement, l'évolution de tous les paramètres d'importance de notre réseau (la sparsity, l'accuracy, le learning rate), ce qui constituait un moyen formidable d'ajustement de nos hyperparamètres au cours de l'entrainement.
Ensuite d'un point de vue théorique, mes lectures et surtout la variété des domaines étudiés (sparsification, robustesse) et des méthodes explorés (low-rank regularization, magnitude prunning, distillation) m'ont permis d'acquérir des connaissances solides sur les thématiques de robustesse et de moyens de compression des réseaux. Cette dernière est un sujet d'actualité et d'importance à deux points de vue, si tant est que l'on considère les besoins énergétiques du Deep Learning comme tel. En effet, récemment, Neil Thompson, chercheur au MIT, a souligné que le coût computationnel des réseaux de neurones à l'état de l'art croit si vite que nous pourrions rapidement atteindre un mur (lien ici). D'autre part, leurs besoins croissants en énergie représentent un enjeu écologique très préoccupant et limiter notre consommation énergétiques apparaît comme un enjeu crucial de ces prochaines années.
Pour terminer, ce stage m'a permis de découvrir le quotidien du télétravail et comprendre la manière la plus efficace de travailler dans de telles circonstances. J'ai beaucoup apprécié cette expérience et trouve à titre personnel que le télétravail est un outil efficace. Les réunions sont plus courtes, concises et informatives. Le fait de ne pas avoir de réelle coupure entre le monde du travail (en bureau) et ma vie personnel m'a également plu car cela m'a permis de m'organiser plus facilement et d'avoir finalement plus de temps à consacrer à ma recherche.

6 Remerciements

Je tiens à remercier le professeur Tanner, Constantin Puiu pour cette expérience, leurs conseils et surtout leur accueil dans le projet.

References

E.; Hooker S. Gale, T.; Elsen. The state of sparsity in deep neural networks. *Journal of the Royal Statistical Society*, 2012.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Bengio Y. Learning deep architectures for ai. *IEEE Symposium on Security Privacy*, 2016.

W.; Sutskever I.; Bruna J.; Erhan D.; Goodfellow I.; Fergus R. Szegedy, C.; Zaremba. Intriguing properties of neural networks. *ICLR*, 2014.

X.; Ren S.; Sun J. He, K.; Zhang. Deep residual learning for image recognition. *arXiv:1512.03385v1*, 2015.

I.; Hinton G. Krizhevsky, A.; Sutskever. Imagenet classification with deep convolutionalneural networks. *Advances in Neural Information Processing Systems*, 2012.

J.; Szegedy. C. Goodfellow, I. J.; Shlens. Explaining and harnessing adversarial examples. *Proceedings of the 2015 International Conference on Learning Representations, Computational and Biological Learning Society*, 2015.

A.; Frossard P. Moosavi-Dezfooli, S.; Fawzi. Deepfool: a simple and accurate method to fool deep neural networks. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

P.; Jha S.; Fredrikson M.; Celik Z. B.; Swami A. Papernot, N.; McDaniel. The limitations of deep learning in adversarial settings. *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, 2016a.

P.; Wu X.; Jha S.; Swami A. Papernot, N.; McDaniel. Distillation as a defense to adversarial perturbations against deep neural networks. *IEEE Symposium on Security Privacy*, 2016b.

Michael Carbin Jonathan Frankle. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv:1803.03635*, 2018.

Dan Pei Jun Zhu Justin Cosentino, Federico Zaiter. The search for sparse, robust neural networks. *arXiv:1912.02386*, 2019.

G. W.; Huang R.; Cao Y.; Lui Y. C. Luyu Wang, L.; Ding. Adversarial robustness of pruned neural networks. *ICLR Workshop submission*, 2018.

S.; Wang X.; Yuan B.; Wen W.; Lin X. Ye, S.; Wang. Defending dnn adversarial attacks with pruning and logits augmentation. *ICLR Workshop submission*, 2018.

Chao.; Zhang Changsui.; Chen Y. Guo, Y.; Zhang. Sparse dnns with improved adversarial robustness. *Advances in neural information processing systems*, 2018.

S. Zhu, M.; Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *CoRR*, 2017.

E. R.; Behboodi A.; Mathar R. Langenberg, P.; Balda. On the effect of low-rank weights on adversarial robustness of neural networks. *arXiv:1901.10371v1*, 2019.

Xiao T.;Zhang Y.;Wang X.;Weinan E. Tai, C.;. Convolutional neural networks with low-rank regularization. *arXiv:1511.06067*, 2016.

W.; Bruna J.; LeCun Y.; Fergus R . Denton, E. L.; Zaremba. Exploiting linear structure within convolutional networks for efficient evaluation. *NIPS*, 2014.

A.; Zisserman A. Jaderberg, M.; Vedaldi. Speeding up convolutional neural networks with low rank expansions. *arXiv:1405.3866*, 2014.

Y.; Rakhuba M.; Oseledets I.; Lempitsky V. Lebedev, V.; Ganin. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv:1412.6553*, 2014.

; Szegedy C. Ioffe, S. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167*, 2015.

O.; Dean J. Hinton, G.; Vinyals. Distilling the knowledge in a neural network. *arXiv:1503.02531*, 2015.