Introduction à l'apprentissage profond

Culture Sciences de l'Ingénieur

Édité le 30/05/2022

école normale supérieure paris—saclay

Steeven JANNY - Ludovic DE MATTEIS – Wenqi SHU-QUARTIER

Cette ressource est issue d'une publication du numéro 108 de La Revue 3EI d'avril 2022. Steeven Janny, ancien élève du DER Nikola Tesla de l'ENS Paris-Saclay, est doctorant au laboratoire LAGEPP sur l'établissement de modèles profonds pour la simulation et la prédiction de phénomènes physiques. Ludovic De Matteïs, élève du DER Nikola Tesla, est auteur de l'article « Personalized Human-Swarm Interaction Through Hand Motion » [6] issu de son stage à l'EPFL sur l'apprentissage des mouvements intuitifs de l'utilisateur pour le contrôle d'une flotte de drone. Wenqi Shu-Quartier, ancienne élève du DER Nikola Tesla, est doctorante au laboratoire Neurospin du CEA sur l'application de l'apprentissage par ordinateur à la comparaison de la maladie d'Alzheimer avec son analogue chez le chimpanzé.

Cette ressource, la deuxième du « Dossier Intelligence Artificielle » [7], présente l'outil à la source des progrès déterminants de l'IA au 21^{e} siècle : les réseaux de neurones et leur entraînement dit apprentissage profond. Après une présentation détaillée des réseaux de neurones, la ressource développe les méthodes d'apprentissage en soulignant l'importance des données. Elle présente ensuite quelques évolutions des réseaux de neurones : convolutionnels (CNN), récurrents (RNN), auto-encodeurs (AE), adversariaux génératif (GAN) et transformers pour finir sur une implémentation pratique d'un entraînement de réseau de neurones pour la classification de chiffres manuscrits.

1 – Introduction

Depuis les bureaux spacieux des grands groupes industriels, jusqu'aux locaux des incubateurs à start-ups accompagnant de jeunes entrepreneurs ambitieux, tous annoncent la venue d'une quatrième révolution industrielle sonnant le début de l'ère de l'intelligence artificielle et du Big Data. Comme ses grandes sœurs, cette révolution s'appuie sur une nouvelle sémantique dont certains éléments de langage vous sont sûrement déjà familiers : Smart Tech, Cloud computing, IA, etc... La quatrième révolution industrielle serait donc celle d'une nouvelle génération de machines et d'une automatisation plus poussée et plus généralisée, une transition permettant d'améliorer l'efficience et la productivité. Pour cela, le monde moderne dispose de deux nouveaux outils redoutables : l'expansion fulgurante d'Internet, et les récentes avancées en matière d'intelligence artificielle.

La dernière décennie donne en partie raison à ces nouveaux prophètes grâce à un alignement des astres technologiques particulièrement opportun : (1) une explosion de la puissance de calcul des ordinateurs, (2) une mise à disposition de quantités massives de données numériques (le fameux Big Data) et (3) la remise au goût du jour d'une technologie vieille de 60 ans dont le potentiel est enfin pleinement libéré grâce aux deux premiers points, nous voulons bien sûr parler des réseaux de neurones.

L'apprentissage profond (c'est le nom de la discipline) est un domaine aujourd'hui soutenu par une communauté très active d'industriels et de chercheurs, et compte plusieurs succès à son palmarès. Ses premières grandes avancées sont concentrées sur la vision par ordinateur : reconnaissance d'objets, localisation de défauts dans les chaînes de production, détection de maladies grâce à

l'imagerie médicale, etc... Récemment, le champ d'application s'est élargi : traitement du langage naturel (GPT-3 développé par OpenAI, chatbots), robotique (apprentissage par renforcement), reconnaissance vocale (Alexa, Siri, Cortana...), et même physique, avec notamment la réussite spectaculaire d'un modèle d'apprentissage profond pour la stabilisation du plasma dans le réacteur à fusion ITER, là où les techniques de contrôles traditionnelles avaient échoué.

Malheureusement, tout succès médiatique vient avec son lot de contre-vérités et d'exagérations. Nombre d'articles relatent chaque jour les dernières prouesses de l'IA. Souvent dithyrambiques, parfois même alarmistes, il est difficile de démêler la science du travail d'écriture. Après une première ressource [8] présentant différents domaines et méthodes de l'IA, cet article a pour ambition de poser les bases du fonctionnement et de la mise en œuvre des réseaux de neurones et de mettre en avant les principaux défis que ceux-ci doivent encore relever.

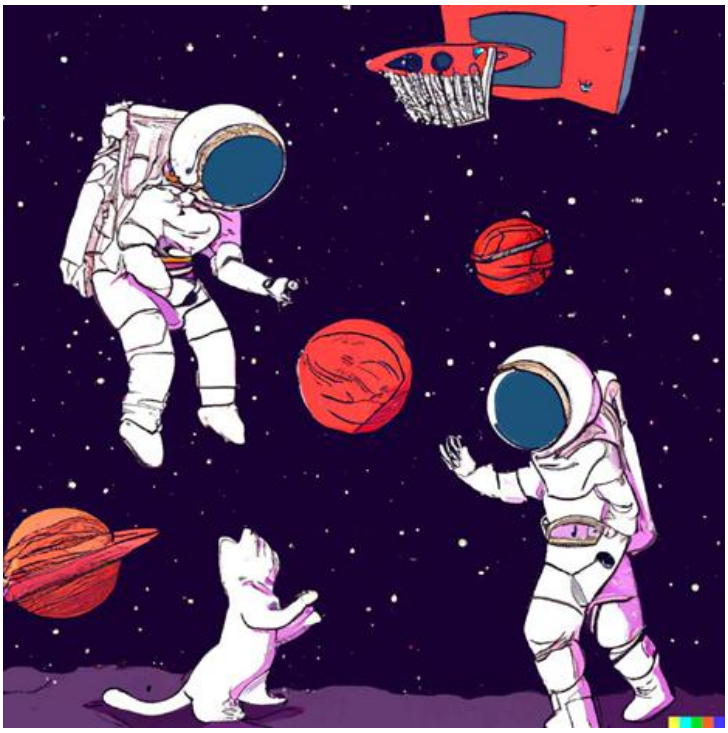

Figure 1 : Image générée par un réseau de neurones à partir de la phrase « Un astronaute jouant au basketball dans l'espace avec des chats, dessiné comme dans un livre pour enfants » (OpenAI Dall-E 2)

2 – Les réseaux de neurones comme modèles paramétriques

2.1 - Les données, ou l'or numérique

Votre âge, votre sexe, le contenu de votre dernier panier de courses, vos performances au dernier jeu mobile à la mode, toutes ces données ont un prix que bon nombre d'entreprises sont prêtes à acheter. Cette ruée vers l'or numérique est en partie motivée par le principe d'application du deep learning.

Mettons-nous quelques instants dans la peau d'un ingénieur en apprentissage profond. Nous nous intéressons donc à la recherche d'une fonction $y = f(x)$ dont nous ignorons complètement la formulation. Il peut s'agir d'une fonction arbitrairement complexe, comme la relation entre les images perçues par une caméra embarquée sur un véhicule et la trajectoire optimale que le véhicule devrait suivre pour atteindre un certain emplacement. À défaut de pouvoir écrire mathématiquement cette fonction, nous disposons de N mesures $(x_i, y_i)_{i=1\dots N}$ de celle-ci, réunies dans un grand jeu de données.

Le problème est alors le suivant : comment construire une approximation de la fonction f à partir de ces seules mesures ? Avant même de s'intéresser aux méthodes envisageables, on peut intuitivement se convaincre qu'une fonction complexe nécessitera plus de points de mesure. Par exemple, deux mesures suffisent pour approcher une fonction linéaire, mais au moins trois mesures sont nécessaires pour approcher une fonction quadratique. Un des principaux enjeux du Big Data est ainsi de collecter une large quantité de données afin de pouvoir approcher des fonctions de plus en plus complexes.

2.2 – Un détour par la régression linéaire

Pour approcher une fonction dont on ne connaît que quelques réalisations, la méthode classique consiste à choisir un modèle paramétrique, c'est-à-dire une fonction dont le comportement sera contrôlé par un certain nombre de paramètres qu'il nous faudra régler (on parle d'identification des paramètres).

La régression linéaire est l'exemple le plus trivial de cette technique : il s'agit de chercher les valeurs de a et b de sorte que l'équation $y = ax + b$ représente au mieux les points de mesure du jeu de données. L'identification de a et b se fait généralement en résolvant un problème de minimisation d'une mesure de l'erreur, comme la distance quadratique :

$$(a, b) = \arg\min_{(a,b)} J = \sum_{i=0}^{N} (y_i - (ax_i + b))^2$$

Cette solution s'adapte sans problème aux plus grandes dimensions en ré-écrivant le modèle linéaire sous forme matricielle $y = Wx + b$. La régression linéaire est intéressante car sa forme, très simple, permet d'obtenir rapidement la solution optimale au problème, c'est-à-dire l'obtention des valeurs de W et b donnant la plus petite erreur J possible. L'inconvénient, c'est que bien peu de phénomènes sont linéaires : il y a peu de chance pour qu'un modèle linéaire parvienne à prédire l'évolution du prix des actions en bourse.

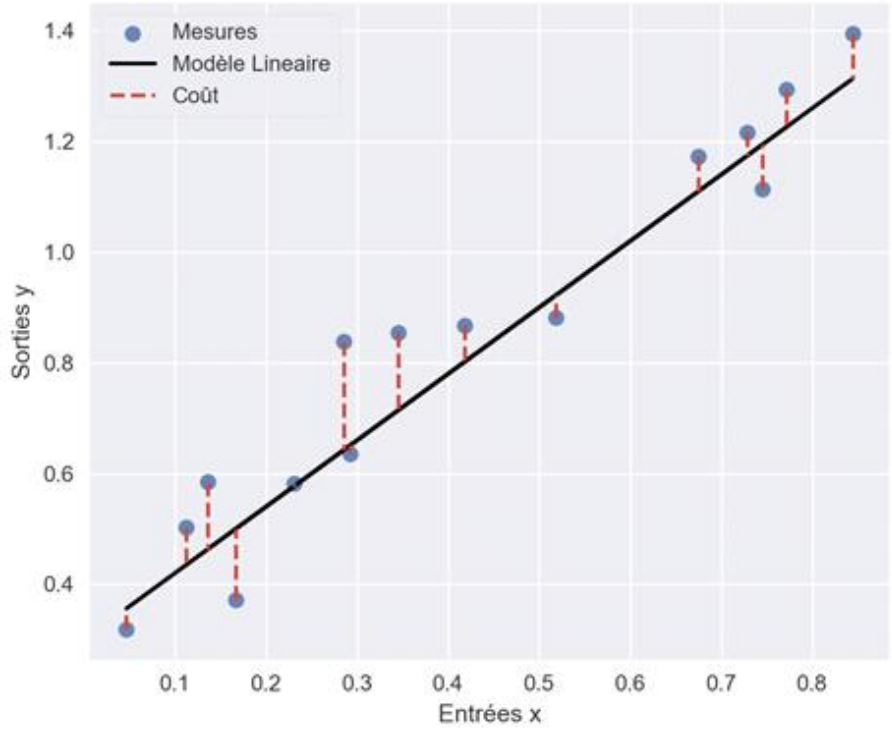

Figure 2 : La régression linéaire consiste à trouver la droite qui minimise la distance aux points de mesures, représentée en rouge sur la figure

2.3 – Un modèle plus puissant

Tout comme la régression linéaire, le réseau de neurones est un autre modèle paramétrique qui, sous sa forme la plus simple, consiste en un enchaînement de modèles linéaires séparés par de petites fonctions non-linéaires :

$$y = W_1\sigma(W_2 \dots \sigma(W_n x + b_n) \dots + b_2) + b_1$$

Cela peut paraître surprenant, mais cette nouvelle structure, appelée perceptron multi-couche (multi-layer perceptron ou MLP) est in fine incroyablement plus puissante que la régression linéaire. Mathématiquement, il s'agit d'un approximateur universel, autrement dit, un MLP peut approcher n'importe quelle fonction avec une précision arbitraire, si tant est qu'il contienne suffisamment de couches linéaires enchaînées.

Il y a bien sûr une contrepartie : dans la quasi-totalité des cas, il est impossible de trouver les valeurs optimales de $W_1, \dots, W_n, b_1, \dots, b_n$ et le mieux que l'on puisse garantir est de trouver un minimum local de J. Cet aspect a longtemps fait fuir les chercheurs en intelligence artificielle, jusqu'à ce que l'on s'aperçoive que ce minimum local était déjà incroyablement performant pour de nombreuses tâches.

2.4 – Petit passage par la biologie

Si les réseaux de neurones s'appellent ainsi, c'est effectivement parce qu'ils sont en quelque sorte reliés à ce que l'on connaît du fonctionnement d'un neurone biologique. Le perceptron (Franck Rosenblatt, 1958) est inspiré du fonctionnement du cerveau : un neurone reçoit des impulsions électriques x depuis ses axones, qu'il combine entre elles. Si le signal résultant W_x est supérieur à un certain seuil - b, le neurone est activé (et émet à son tour une nouvelle impulsion électrique). Mathématiquement, cela se met sous la forme suivante :

$$y = \sigma(W_x + b)$$

où σ est la fonction signe, renvoyant 0 si l'entrée est négative et 1 sinon. En 1969, Minsky et Papert assènent un coup terrible au perceptron en mettant en évidence les faiblesses théoriques qui rendent les perceptrons très peu efficaces (impossibilité de représenter une fonction OU EXCLUSIF). Cette découverte reléguera au placard le perceptron pour une petite décennie, jusqu'aux travaux de Rumelhart et Hinton, qui en 1986 proposent de combiner plusieurs perceptrons organisés en réseau pour former le MLP.

La véritable révolution surviendra au début des années 2010, avec des succès retentissants comme la reconnaissance d'images avec des modèles remplaçant les couches linéaires par des convolutions sur des images (comme AlexNet en 2012), ou la résolution du jeu de Go (avec AlphaGo en 2015). Aujourd'hui, les modèles véritablement utilisés en apprentissage profond n'ont plus grand-chose à voir avec le neurone biologique.

3 – Architecture d'un réseau de neurones

3.1 – Définition et structure générale

Comme expliqué précédemment, les réseaux de neurones permettent de modéliser des phénomènes complexes. Ils sont basés sur l'enchaînement de modèles linéaires et de fonctions non linéaires. Les transformations $h_i = \sigma(W_i h_{i-1} + b_i)$ sont appelées couches. On distingue la couche d'entrée, qui reçoit le vecteur d'entrée x, la couche de sortie, qui renvoie un vecteur de sortie y (sans lui appliquer de fonction non linéaire σ), et les couches cachées, qui reçoivent et renvoient

des vecteurs intermédiairesh_i. Ces vecteurs intermédiaires sont appelés représentations cachées (ou représentations latentes) et sont des variables internes au réseau de neurones. Les non-linéarités σ, quant à elles, sont appelées fonctions d'activation et nous les présenterons plus en détail dans la suite de cette partie. Il est intéressant de noter que les fonctions d'activation peuvent en théorie être différentes pour chaque couche (même si ce n'est généralement pas le cas).

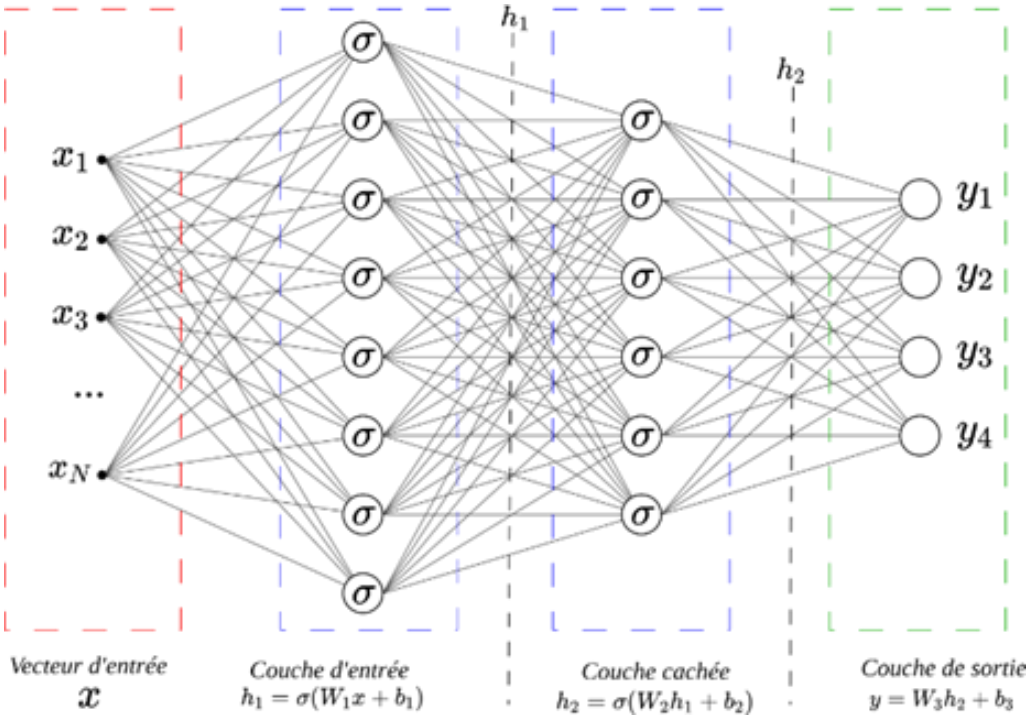

Figure 3 : Représentation classique d'un MLP
Le modèle renvoie un vecteur de sortie y correspondant au vecteur d'entrée x donné

3.2 – Paramètres et hyperparamètres

Le réseau de neurones dispose d'un certain nombre de paramètres et hyperparamètres. Il est important de bien distinguer ces deux notions :

- Les paramètres sont des éléments variables du modèle qui définissent le comportement du modèle. Les paramètres changent au cours de l'apprentissage de manière à obtenir un modèle répondant à la tâche demandée. Dans le cas du MLP, les paramètres sont l'ensemble des poids W_i et des biais b_i.
- Les hyperparamètres sont des réglages du modèle et de l'algorithme d'apprentissage, choisis par le concepteur et qui ne sont pas modifiés par l'optimisation. Ils conditionnent eux aussi le comportement du modèle et doivent donc être choisis judicieusement. Dans le cas d'un MLP, les hyperparamètres sont (entre autres) le nombre de couches cachées, les dimensions des représentations cachées, les fonctions d'activations utilisées et le taux d'apprentissage.

Il est important de bien choisir les hyperparamètres de notre modèle car ces derniers influent sur la phase d'apprentissage et sur les performances finales du modèle. Il s'agit là d'une des principales difficultés de l'utilisation de réseaux de neurones. Le concepteur doit avoir une idée au moins approximative de l'influence de chacun des hyperparamètres pour pouvoir les choisir correctement.

3.3 - Les fonctions d'activation

La fonction d'activation σ utilisée sur chacune des couches est une fonction non-linéaire qui s'applique sur chaque élément des vecteurs h_i Il existe différentes onctions d'activations utilisées

classiquement, ayant chacune leurs avantages et leurs inconvénients. Nous n'allons en présenter ici que quelques-unes parmi les plus courantes.

- Fonction ReLU (Rectified Linear Unit) : Cette fonction est définie comme égale à 0 sur $\mathbb{R}^-$et égale à l'identité sur $\mathbb{R}^+$. Il s'agit de la fonction d'activation la plus couramment utilisée dans le cas général. Il existe de nombreuses variantes de la fonction ReLU réalisant différentes approximations en 0 dans le but de la rendre dérivable. Cependant, la fonction ReLU présente l'avantage d'être linéaire par morceaux, donc rapide à calculer et présentant une dérivée simple.
- Fonction Sigmoïde : Cette fonction est définie par :

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

  Les valeurs renvoyées par cette fonction sont comprises entre 0 et 1. Cette fonction est souvent utilisée sur la dernière couche du réseau de neurones pour des problèmes de classification.
- Fonction tangente hyperbolique : Cette fonction est définie par :

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  Elle renvoie des valeurs comprises entre -1 et 1. Elle présente principalement l'avantage de renvoyer des valeurs bornées et donc d'éviter les divergences lors des prédictions.

Une illustration de ces différentes fonctions est présentée figure 4.

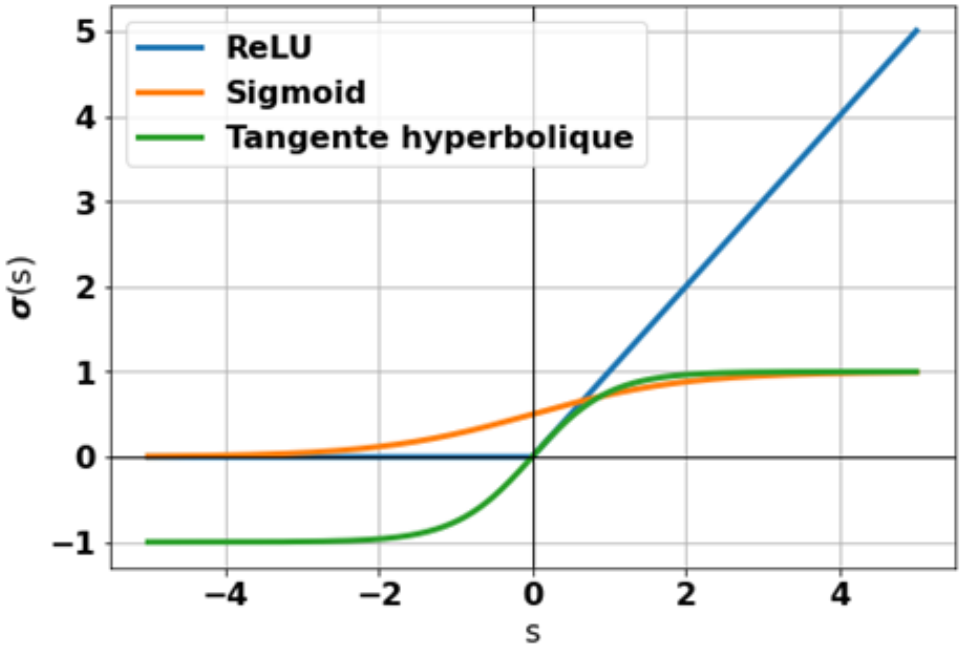

Figure 4 : Évolution des différentes fonctions d'activations présentée.
Les tracés sont faits pour des valeurs des neurones comprises entre -5 et 5

3.4 - Exemple applicatif : classification de chiffres manuscrits

Pour illustrer le fonctionnement d'un réseau de neurones, nous nous intéressons à la classification de chiffres manuscrits. 60 000 exemples peuvent être obtenus grâce à la base de données MNIST. Il s'agit ici d'apprentissage supervisé, on dispose d'exemples d'entrées x_i étiquetées avec les sorties correspondantes y_i souhaitées.

Il est possible de générer un vecteur d'entrée en prenant les pixels un à un. On obtient donc un vecteur d'entrée x de taille 28 ×28 = 784. La sortie y est constituée de 10 valeurs, chacune associée

à une classe possible comprise entre 0 et 9 (estimée par le réseau de neurones). La classe estimée correspond alors à celle pour laquelle la valeur est la plus élevée. On peut en quelque sorte voir cette valeur comme un indice de confiance qu'a le modèle en une classe : plus cette valeur est élevée (en comparaison aux autres), plus le MLP estime que cette classe est la bonne.

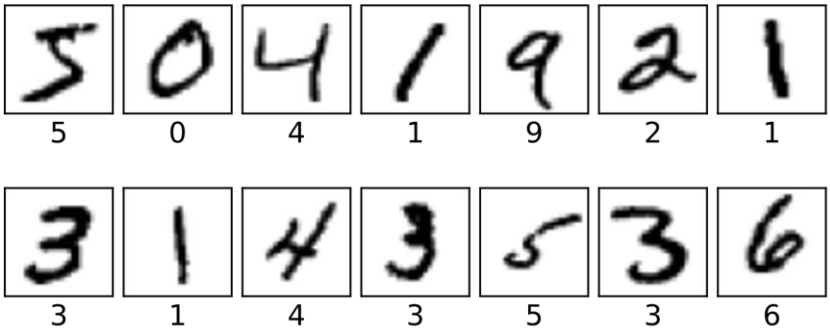

Figure 5 : Exemples d'éléments de la base de données
Les entrées sont des images de 28x28 pixels et les sorties sont des classes entre 0 et 9, correspondant au chiffre identifié

Avant de pouvoir entraîner le modèle (c'est-à-dire trouver les valeurs des paramètres), il nous faut choisir un certain nombre d'hyperparamètres comme le nombre de couches cachées, la taille de ces couches et la fonction d'activation à utiliser. Ce problème étant assez simple, il fonctionne assez bien avec une unique couche cachée de 200 neurones. On observe expérimentalement que la fonction d'activation ReLU permet dans ce cas d'avoir de bons résultats en limitant le nombre de calculs nécessaires et donc le temps d'apprentissage.

Cependant, nous avons expliqué précédemment que la couche de sortie est constituée de 10 valeurs qui sont comparées les unes aux autres afin de choisir une classe à prédire. Pourtant, les valeurs de la couche de sortie (dans le cas de la fonction d'activation ReLU) peuvent théoriquement prendre des valeurs infiniment grandes. Il devient alors difficile de comparer le vecteur de sortie du MLP avec un vecteur cible comme on le voudrait. On choisit alors de normaliser les valeurs de la couche de sortie entre 0 et 1 à l'aide d'une couche appelée Softmax.

La couche Softmax est une couche particulière qui ne traite pas les éléments du vecteur indépendamment. Le nouveau vecteur de sortie $Z = f(y)$ est défini par :

$$Z_i = \frac{\exp(y_i)}{\sum_{j=1}^{N} exp(y_j)}$$

Cette fonction a pour objectif de ramener les valeurs de la couche de sortie entre 0 et 1 afin de les interpréter comme des probabilités (même si ce ne sont pas vraiment des probabilités, plutôt des taux de confiance du modèle).

On dispose donc d'un modèle prenant en entrée une image x de chiffres manuscrits et renvoyant un vecteur y de 10 valeurs permettant de prédire l'appartenance à une classe. On a également un grand nombre de données d'entraînement étiquetées qui serviront d'exemples pour entraîner le modèle (la classe d'un élément de la base d'apprentissage est convertie en un vecteur de taille 10 avec un 1 sur la classe réelle et neuf 0 partout ailleurs). À présent, il ne nous manque plus qu'à entraîner correctement les paramètres du réseau de neurones afin que ce dernier ait le comportement voulu et classe correctement nos images.

4 – Comment entraîner un réseau de neurones ?

4.1 – Minimiser un coût pour résoudre un problème

Dans l'exemple précédent, le réseau de neurones a pour but de classifier l'image qu'on lui fournit en entrée : autrement dit, la sortie souhaitée ne peut prendre que des valeurs discrètes. On parle alors de problème de classification. (Remarquez cependant que la sortie prédite se trouve sous la forme d'un vecteur y listant les « probabilités » pour chaque classe possible.) Lorsque la sortie du modèle prend des valeurs continues, comme dans le cas d'un réseau de neurones qui doit prédire le niveau de l'eau dans 5 ans d'une zone à risque, on parle de problème de régression.

Ces deux types de problèmes sont assez similaires dans leur résolution. Seule la fonction de coût change : cette fonction évalue quantitativement la distance entre la prédiction du modèle et la réponse attendue au cours de l'entraînement, et sa minimisation à l'aide d'algorithmes itératifs permet de guider la mise à jour des valeurs des paramètres, i.e. les poids W_i et les biais pour un MLP. En voici une liste non exhaustive :

- Pour la régression : on utilise souvent une fonction de coût quadratique, qui est l'erreur quadratique moyenne entre les sorties prédites $\hat{y}_i$ par le modèle pour les N données d'entraînement et les sorties réelles y_i correspondantes :

$$J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  On peut aussi utiliser l'erreur absolue moyenne, ou encore la fonction de coût de Huber qui permet d'être moins sensible aux outliers (valeurs plus éloignées du modèle que les autres).
- Pour la classification : on utilise presque toujours la fonction d'entropie croisée. Dans le cas d'une classification binaire (i.e. lorsqu'il n'y a que deux classes possibles, 0 ou 1), la fonction de coût est calculée comme suit, où $\hat{y}_i$ représente donc la « probabilité » (estimée par le modèle) qu'a le i-ième échantillon d'être dans la classe 1:

$$J = -\frac{1}{N}\sum_{i=1}^{N}(y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i))$$

  Si nous avons affaire à un problème de classification multi-classes, il suffit de calculer séparément l'entropie croisée pour chacune des classes à chaque donnée d'entraînement et d'en faire la somme. Mais parfois, on préfère convertir le problème en une succession de problèmes de classification binaire avant de fusionner leurs réponses. Deux stratégies existent : la première, celle du « un-contre-le-reste », consiste à apprendre autant de classifieurs qu'il y a de classes, et chaque modèle doit estimer la probabilité d'une classe contre celles de toutes les autres. La deuxième, qui est paradoxalement préférable lorsque le nombre de classes est grand, s'appuie sur du « un-contre- un » : un classifieur est appris pour chaque paire de classes, et la classe prédite est celle qui a remporté le plus de matchs en un contre un.

La phase d'entraînement d'un réseau de neurones est donc entièrement consacrée à la minimisation d'une fonction de coût, qui représente l'écart entre les prédictions du modèle pour les données d'entraînement et les sorties réelles.

4.2 – L'algorithme de descente du gradient

La fonction de coût ne sert pas seulement à dire au modèle s'il a bien fait son travail de prédiction. Elle permet surtout de guider l'algorithme de minimisation sur le terrain, en lui montrant comment ajuster les valeurs des paramètres à chaque itération pour tendre vers la minimisation de la fonction coût. En effet, lorsqu'un problème d'optimisation n'est pas soluble de manière déterministe, comme pour les réseaux de neurones qui représentent un problème trop complexe pour qu'on puisse calculer une solution analytique, il existe des algorithmes itératifs qui permettent de trouver une solution approchée, à condition toutefois que la fonction à minimiser soit dérivable, ce qui est le cas ici. Classiquement, on utilise l'algorithme de descente du gradient : voyons ce qu'il a sous le capot.

Visualisez-vous les yeux bandés sur le flanc d'une montagne : pour arriver en bas de la montagne, c'est-à-dire au point d'altitude minimale, vous allez évaluer du pied la pente à l'endroit où vous vous trouvez, puis vous ferez un pas dans la direction descendante, et vous recommencerez jusqu'à atteindre un creux, n'est-ce pas ? Mathématiquement, la montagne peut être vue comme la représentation graphique de la fonction coût, et le gradient d'une fonction en un point correspond grossièrement à la pente en ce point. Si le gradient est positif et élevé, cela signifie que la pente est grande et monte vers la droite, donc on peut faire un grand pas vers la gauche ; si le gradient est négatif et petit, la pente est faible et descend vers la droite, donc on fait un petit pas vers la droite (de peur de dépasser le minimum et de devoir revenir sur nos pas). Autrement dit, à chaque itération de l'algorithme, on met à jour chaque paramètre en lui retirant une valeur proportionnelle au gradient du coût en ce point (à une constante multiplicative près, appelée taux d'apprentissage η, fixée arbitrairement) :

$$W_j \leftarrow W_j - \eta \times \frac{\partial J}{\partial W_j}$$

$$b_j \leftarrow b_j - \eta \times \frac{\partial J}{\partial b_j}$$

Il faut donc pouvoir calculer la valeur du gradient de la fonction coût par rapport à chaque paramètre. Pour cela, on réalise deux passages dans le réseau de neurones à chaque itération (ou epoch) :

- Une propagation de l'information vers l'avant (forward propagation) : on fournit les données d'entraînement au réseau en entrée pour obtenir les valeurs des prédictions $\hat{y}$ en sortie, en appliquant à chaque couche de neurones la formule suivante :
$$h_j = \sigma\left(W_j h_{j-1} + b_j\right)$$
(h_0 étant donc ici la donnée d'entraînement x qu'on met en entrée du modèle, et le signal h_M final représente la sortie prédite $\hat{y}$).
- Une propagation de l'information vers l'arrière (backward propagation) : on calcule, à partir des prédictions $\hat{y}$, la valeur des gradients du coût par rapport aux poids W_i et biais b_i du réseau, dont les expressions sont obtenues par la règle de la chaîne en « remontant » les couches de neurones. Prenons l'exemple du coût quadratique :
$$J = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_1)^2$$
où $\hat{y}_i = \sigma(W_M.\sigma(W_{M-1} \dots + b_{M-1}) + b_M)$ est la sortie prédite pour la i-ième donnée d'entraînement et s'exprime en fonction des paramètres W_i et b_i. Reformulons $\hat{y}_i$ de manière à mettre en évidence l'approche itérative du calcul du gradient, on a donc :
$$\hat{y}_i = h_m = \sigma(W_M.h_{M-1} \dots + b_{M-1})$$

pour un MLP à M couches. Puisqu'on dérive des fonctions composées, les gradients du coût par rapport aux paramètres de la dernière couche s'expriment par :

$$\frac{\partial J}{\partial W_M} = \frac{\partial J}{\partial h_M} \times \frac{\partial h_M}{\partial W_M}$$

Et

$$\frac{\partial J}{\partial b_M} = \frac{\partial J}{\partial h_M} \times \frac{\partial h_M}{\partial b_M}$$

Il suffit à présent d'exprimer séparément chacune des dérivées :

$$\begin{cases} \frac{\partial J}{\partial h_M} = \frac{2}{N}\sum_{i=1}^{N}(\hat{y_i} - y_i) \text{ pour le coût quadratique} \\ \frac{\partial h_M}{\partial W_M} = h_{M-1}.\sigma'(W_M.h_{M-1} + b_M) \\ \frac{\partial h_M}{\partial W_M} = \sigma'(W_M.h_{M-1} + b_M) \end{cases}$$

Ainsi, la rétropropagation consiste surtout à calculer les $\frac{\partial J}{\partial h_j}$ successifs, puisque les autres gradients sont faciles à déduire. La formule de récurrence ci-dessous nous permet de le faire, en remontant les couches jusqu'à arriver au bout du réseau de neurones :

$$\frac{\partial J}{\partial h_{j-1}} = W_j.\sigma'(W_j.h_{j-1} + b_j)\frac{\partial J}{\partial h_j}$$

Une fois tous les gradients calculés, on peut mettre à jour les valeurs des paramètres.

L'initialisation des paramètres se fait généralement en prenant une valeur choisie au hasard dans [-1, 1], suivant une loi uniforme ou normale, même s'il existe d'autres stratégies plus sophistiquées (initialisation de Xavier, He...). Le critère d'arrêt se présente souvent sous la forme d'un nombre limité d'itérations afin de limiter la durée de la phase d'entraînement. Enfin, le choix du taux d'apprentissage peut se révéler délicat : un pas trop grand pourrait vous faire louper le minimum et empêcher l'algorithme de converger, mais un pas trop petit peut aussi être pénalisant en vous faisant avancer trop lentement... À vous de tester plusieurs valeurs pour que la fonction de coût converge proprement. Souvent, il est utile de commencer l'entraînement avec un taux suffisamment élevé pour atteindre rapidement des résultats satisfaisants, puis de baisser progressivement le pas afin d'affiner le modèle.

4.3 – En pratique, ça donne quoi ?

L'un des problèmes que présente l'algorithme de descente du gradient réside dans la possibilité qu'on reste bloqué dans un minimum local. Autrement dit, au lieu de parvenir en bas de la montagne, vous êtes malheureusement descendu dans une petite grotte et l'algorithme ne vous permet pas d'en sortir. Compte tenu des courbes d'erreurs très accidentées dessinées par les réseaux de neurones, il existe une multitude de minima locaux. De ce fait, l'apprentissage global converge rarement vers le minimum global de la fonction d'erreur lorsqu'on applique les algorithmes basés sur le gradient global. L'apprentissage avec l'algorithme de descente du gradient stochastique est une solution permettant de mieux explorer ces courbes d'erreurs : au lieu d'utiliser toutes les N données d'entraînement à chaque itération pour calculer les gradients, on ne va en prendre que quelques-uns, choisis au hasard, en mini-lots (ou mini-batchs). Cela permet de plus d'éviter le stockage d'un trop grand nombre de données pour des fonctions ayant parfois quelques milliers de paramètres. Pour des réseaux de neurones plus petits, on peut se permettre de choisir des algorithmes d'optimisation plus coûteux en calculs, qui font intervenir des moments d'inertie (imaginez-vous à nouveau sur la montagne, le fait d'avoir un peu d'élan vous permettrait de sortir d'un creux ponctuel sur votre descente), ou des mises à jour automatiques du pas (si la pente est

la même depuis un certain temps, vous pouvez vous permettre de faire des pas plus grands par exemple). Citons-en quelques-unes : Adagrad, Adam, RMSProp, Nesterov…

Certains problèmes récurrents peuvent empêcher le bon déroulement de la phase d'entraînement, notamment la disparition du gradient (vanishing gradient), ou son explosion (exploding gradient). Dans la première situation, les gradients deviennent de plus en plus petits lorsque l'algorithme progresse vers les couches inférieures du réseau, ce qui amenuise considérablement l'effet de la mise à jour des poids par descente de gradient et empêche une convergence rapide. Fixer un taux d'apprentissage plus petit, ou choisir des fonctions d'activation σ qui ne sont pas nulles pour les valeurs négatives (comme la fonction Leaky ReLU, dont la dérivée est égale à 1 lorsque l'entrée est positive et à une petite constante lorsque l'entrée est négative). Dans la deuxième situation, les gradients deviennent de plus en plus grands, ce qui fait diverger l'algorithme : en effet, lorsqu'on a un réseau très profond, comme les gradients de chaque couche sont multipliés entre eux au cours de la backpropagation, on peut très rapidement avoir un gradient qui explose de manière exponentielle. Le problème peut être partiellement résolu en utilisant un taux d'apprentissage suffisamment faible, ainsi qu'en choisissant judicieusement les fonctions d'activation et la fonction de coût, ou en normalisant les données d'entraînement (i.e. en leur soustrayant leur moyenne et en les divisant par leur écart-type, et en gardant ces valeurs caractéristiques pour traiter les données de test de la même manière).

Enfin, notons l'existence du package Autograd dans Pytorch qui effectue automatiquement les calculs de gradients à partir de la fonction coût en collectant toutes les opérations de manière dynamique dans un graphe.

5 – L'apprentissage profond n'est pas l'Eldorado numérique

5.1 – De l'importance des données

Pour un industriel, les possibilités qu'offre l'apprentissage profond sont immenses : la clé de voûte, c'est une grande quantité de données sur le problème que l'on souhaite résoudre. Vous souhaitez détecter automatiquement la présence de tumeurs sur des images de scanner ? Les photos et les dossiers médicaux associés sont déjà là, dans la base de données de la clinique. Un algorithme pour savoir s'il faut accorder ou non un crédit à un client ? Les livres de compte de votre établissement bancaire feront l'affaire. Optimiser l'organisation de votre magasin pour vendre plus ? C'est facile, il suffit de jeter un œil aux tickets de caisse, facilement identifiés au profil du client grâce à sa carte de fidélité. Laissons de côté les fondements éthiques de telles pratiques (cela mériterait un article entier) et tempérons notre excitation : le deep learning n'a besoin que de données pour apprendre, il est vrai, mais encore faut-il que ces données soient propres et suffisamment nombreuses.

Le nettoyage des bases de données est un travail ennuyeux, ingrat, et pourtant absolument indispensable et terriblement chronophage. Il s'agit principalement de corriger toutes les petites anomalies qui pourraient gêner l'entraînement d'un réseau de neurones. Pour un ordinateur, « Dupon », « dupont » et « DUPONT » sont trois personnes différentes. Il s'agit également d'encoder les données en un format compréhensible pour le réseau, c'est-à-dire sous forme d'un vecteur mathématique. La question est vite réglée pour une image, qui n'est qu'une matrice de nombres, mais l'encodage du sexe d'une personne, par exemple, est déjà plus complexe. Nous avons vu dans l'exemple précédent que la solution classique consiste à passer par un vecteur one-hot rempli de zéro à l'exception de la ligne correspond à la classe. La tâche se complique très largement lorsque l'on souhaite travailler avec des données textuelles (avis sur un site de e-commerce, description des produits, etc.). Il y a trop de mots dans le dictionnaire pour imaginer

passer par un vecteur one-hot. On utilise plutôt des algorithmes appelés word2vec qui associe un vecteur à valeurs réelles et de grande dimension à chaque mot (de sorte que, la somme du vecteur « roi » et « femme » est très proche du vecteur du mot « reine »).

Mais supposons que nos données soient déjà parfaitement propres : nos employés ont respecté à la lettre les règles de formatage, et celles-ci sont déjà stockées au format vectoriel. Reste encore un problème, et celui-ci est de taille : avons-nous assez de données pour entraîner un réseau de neurones ? Le jeu de données doit représenter avec suffisamment de finesse les variations de la fonction que l'on cherche à apprendre. C'est comme essayer de reproduire le plan d'une ville avec quelques photos aériennes : si vous n'avez pas assez de photos, la tâche est impossible. Il existe bien quelques techniques d'augmentation de données (symétrie des images, rotations, ...), mais en définitive, on ne peut pas faire de miracle.

On peut imaginer des variantes du problème de manque de données, dans lesquelles les données seraient riches sur un certain domaine, mais très pauvres ailleurs. L'entreprise Apple en a fait les frais en 2017. Plusieurs plaintes ont mis en évidence le fait que sa nouvelle application de déverrouillage par reconnaissance de visage était moins fiable sur les visages asiatiques que sur les visages européens. Évidemment, les développeurs de la multinationale n'ont pas volontairement codés un comportement discriminatoire dans leur application. Le problème venait en réalité des données d'entraînements, qui contiennent majoritairement des visages de personnes blanches de peau, originaires d'Amérique ou d'Europe. Le réseau de neurones n'ayant vu que très peu de visages asiatiques, celui-ci n'a pas été capable de généraliser ce qu'il avait appris sur les visages européens.

Plus facilement identifiables, les évènements rares nécessitent une adaptation dans la manière de construire la fonction de coût pendant l'entraînement. Par exemple, lorsqu'une banque cherche à détecter les fraudes à la carte bancaire, celle-ci doit exploiter des bases de données contenant des millions de transactions licites, mais un nombre dérisoire de transactions officiellement détectées comme illégales. Ce déséquilibre peut par exemple être pris en compte en pondérant plus fortement les erreurs sur les évènements rares dans la fonction de coût.

En définitive, tout est une question de généralisation : dans quelle mesure un réseau de neurones pourra prédire correctement sur des données différentes de celles vues lors de l'entraînement ? Cette question est centrale en apprentissage automatique.

5.2 – Sur et sous- apprentissage

La mise en place d'un réseau de neurones demande d'effectuer de nombreux choix arbitraires pour choisir les hyperparamètres (nombre de couches, forme du réseau, fonctions d'activation, etc.). Ceux-ci peuvent également impacter la capacité de généralisation du réseau. Le sous-apprentissage, par exemple, apparaît lorsque le réseau de neurones ne contient pas assez de paramètres pour apprendre la fonction (voir Fig.3d). On corrige cela en augmentant le nombre de couches et la taille des représentations cachées.

Mais même en apprentissage profond, le mieux est l'ennemi du bien. En augmentant le nombre de paramètres, on peut se retrouver dans la situation contraire du sur-apprentissage (voir Fig.6c). Le réseau de neurones exploite sa grande capacité pour apprendre par cœur les données d'entraînement, et devient incapable de généraliser ce qu'il a appris aux nouvelles mesures. Le sur-apprentissage (ou overfitting) est une épée de Damoclès permanente qui menace (et terrifie) tous ceux travaillant en deep learning. Un algorithme peut fonctionner parfaitement en laboratoire, sur les données d'entraînement, mais être parfaitement inutilisable en situation réelle ! Heureusement, il est possible de limiter les risques en respectant quelques règles simples. La

première, et la plus importante de ces règles, nécessite de découper le dataset en trois ensembles de mesures totalement disjoints.

L'entraînement du réseau de neurones (la descente de gradient) à proprement parler s'effectue sur la totalité des données de l'ensemble d'entraînement. Il représente entre 60 et 80% des données. L'évaluation des performances du modèle entraîné est réalisée sur un deuxième ensemble, appelé ensemble de validation. C'est notamment ce dernier qui servira à régler les hyperparamètres du modèle. Insistons encore un peu sur le mot « disjoint » : comme les exemples de l'ensemble de validation n'ont jamais été vus par le réseau de neurones pendant l'entraînement, on pourra détecter le sur-apprentissage lorsque l'écart entre l'erreur d'entraînement et l'erreur de validation dépassera un seuil raisonnable. Cette séparation a valeur de loi en apprentissage profond : évaluer son modèle sur des données vues en entraînement est considéré comme de la triche et de la malhonnêteté !

Dans le cas idéal, on ajoute un troisième ensemble de test permettant d'évaluer la version finale du modèle, c'est-à-dire après avoir fixé tous les hyper-paramètres. En effet, on peut également sur-apprendre les hyper-paramètres du modèle, d'où cette nouvelle étape de sécurité.

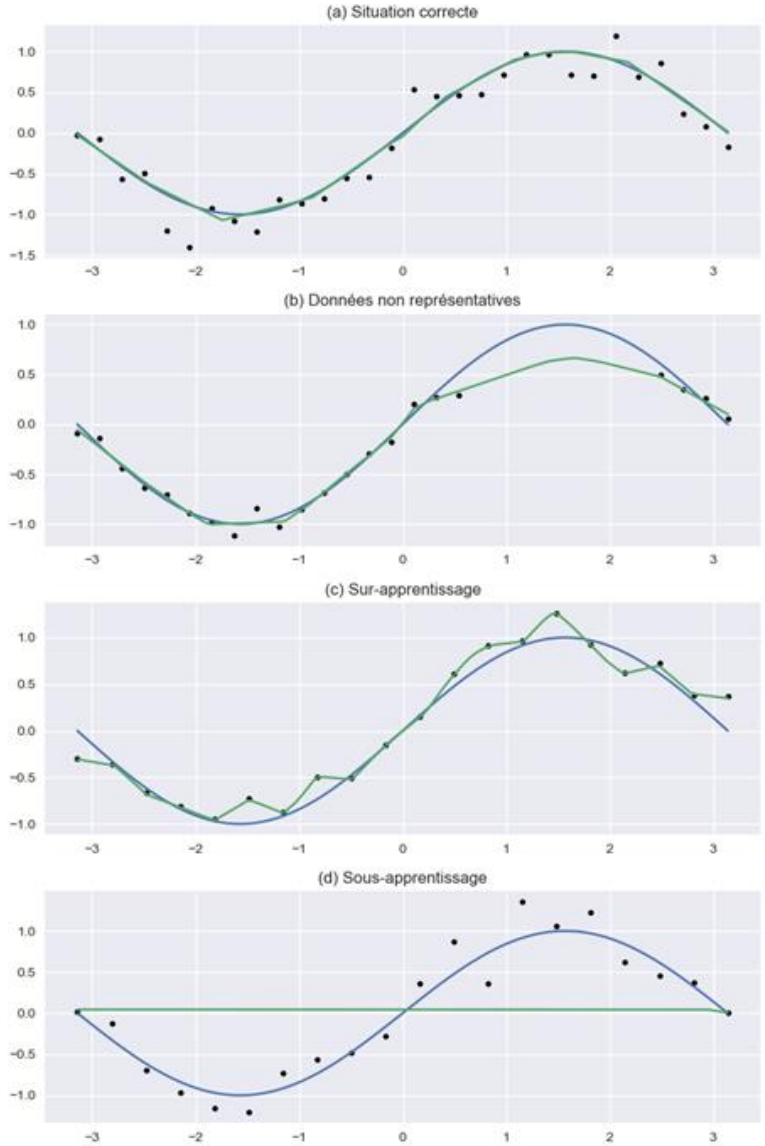

Figure 6 : Résultat d'entraînement d'un MLP à une couche cachée à partir des mesures (points noirs) issues d'une fonction sinus. (a) Les données représentent bien la fonction, et le MLP est bien choisi. (b) Certaines données ont été supprimées, le réseau y est moins performant. (c) Le MLP est sur-dimensionné, celui-ci ne généralise pas. (d) Cette fois, le réseau est sous-dimensionné et ne parvient pas à apprendre quoi que ce soit.

5.3 – Mesures de performances

Félicitations ! Vous avez surmonté toutes les difficultés liées à vos données après plusieurs heures à les nettoyer. Vous avez soigneusement fabriqué vos trois ensembles train/validation/test et l'algorithme de descente de gradient semble finalement avoir convergé après plusieurs heures à monopoliser votre GPU. Voici venu le moment crucial : l'évaluation des performances du modèle. Cette étape vient (bien sûr) s'effectuer sur les données de test, mais la métrique utilisée va varier selon le type de problème. Certaines tâches ont d'ailleurs des métriques qui leur sont propres, comme la détection d'objets dans une image, les problèmes de suivi de mouvement, où encore les tâches de robotique. Loin de nous l'idée d'en dresser une liste exhaustive, étudions plutôt celles que vous serez le plus probablement amené à utiliser.

Pour les problèmes de régression, il n'y pas grand-chose de nouveau : la solution simple utilise la distance euclidienne moyenne entre votre prédiction $\hat{y}$ et valeur attendue y. On la retrouve généralement sous l'appellation RMSE (pour root mean squared error) :

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2}$$

Pour les problèmes de classification, les choses se corsent. Pour l'instant, concentrons-nous sur un problème à deux classes (ce qui correspond à une classification binaire, comme un algorithme qui détecte la présence de défauts sur une chaîne de fabrication à partir des photos d'un composant : y a-t-il des défauts, oui ou non ?). La première idée, qui vient d'ailleurs assez naturellement, serait de compter le nombre de bonnes prédictions par rapport au nombre de prédictions total, autrement dit, la précision du modèle en langage courant. C'est effectivement une bonne mesure des performances, que l'on retrouve d'ailleurs quasi systématiquement. Mais cela ne suffit pas !

A priori, la plupart des composants de la chaîne de production sont de bonne facture (disons, 90%). Les données de test traduisent également de cette répartition et contiennent 90% de photos de « bon composant ». Comment dans ce cas interpréter les performances d'un réseau de neurones qui aurait appris à prédire « Bon » pour absolument tous les composants ? Sa précision serait de 90%, ce qui semble plutôt bon. Pourtant, il est évident que celui-ci est parfaitement inutile ! Il faut donc une autre métrique, et pour cela, il faut différencier les vrais positifs (les composants correctement identifiés comme défectueux) des faux positifs (les composants défectueux identifiés comme bons). On fait de même avec les vrais négatifs et les faux négatifs. Pour mesurer précisément les performances d'un algorithme de classification binaire, on introduit une matrice de confusion dans laquelle chaque ligne correspond à une classe réelle et chaque colonne, à une classe prédite. Ainsi, habituellement, la première ligne de la première colonne correspond au nombre de composants défectueux (positifs) identifiés comme tels, c'est-à-dire le nombre de vrais positifs.

Cette matrice peut se lire de plusieurs façons (cf. figure 7). La diagonale par exemple correspond à la précision dont nous discutions dans le paragraphe précédent. En lisant la dernière ligne, on évalue la capacité du modèle à détecter les pièces valides. La première colonne nous donne une indication sur la fiabilité du réseau lorsque celui-ci prédit qu'un composant est défectueux. La matrice de confusion permet de repérer immédiatement le problème décrit précédemment : si le réseau donne toujours la même réponse, alors l'une des colonnes de la matrice sera entièrement nulle.

Le terme précision en français prête à confusion dans son emploi dans un problème de classification. On y préfère d'ailleurs le mot anglais accuracy ou encore exactitude en français, car la précision définit une autre métrique qui n'a pas tout à fait la même signification. La précision est en réalité le rapport entre le nombre de vrais positifs, divisé par le nombre d'éléments détectés comme positifs. Par exemple, si la précision de votre détecteur de défaut est de 70%, alors sur 100 pièces détectées comme à jeter, 30 d'entre elles étaient parfaitement fonctionnelles.

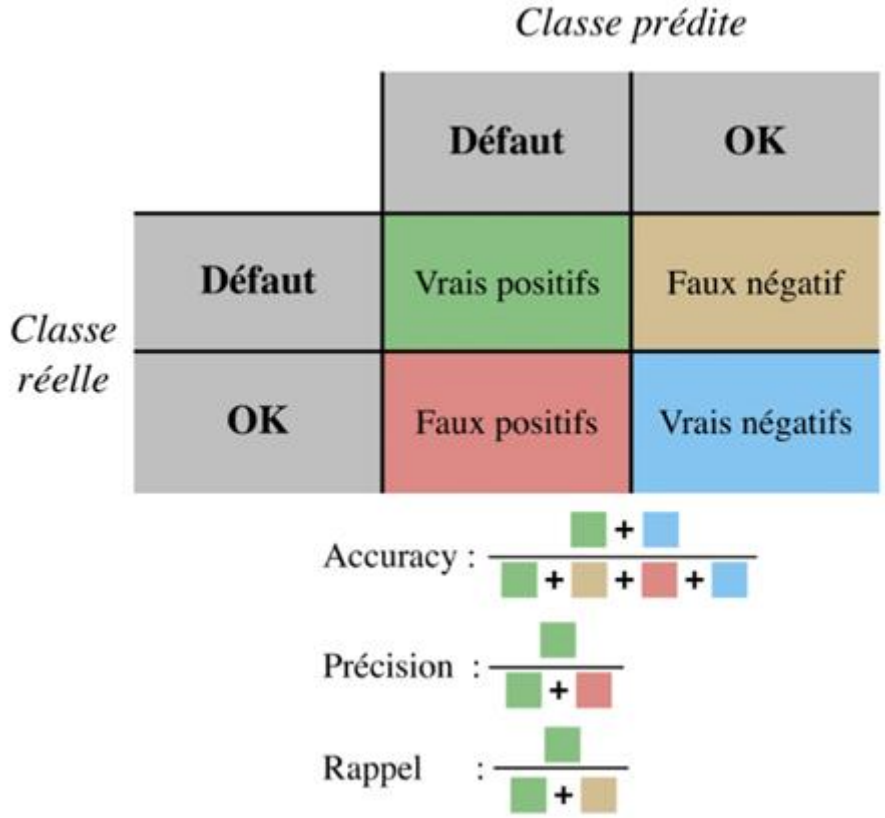

Figure 7 : Matrice de confusion et formules des différentes métriques pour les problèmes de classification

Une troisième métrique s'intéresse aux pièces défectueuses qui ne sont pas détectées par le réseau. Il s'agit du rappel. Ainsi, un rappel de 80% indique que sur 100 composants défectueux, seulement 80 ont été correctement identifiés. Le jonglage entre précision et rappel est une gymnastique qui demande un peu de pratique pour être parfaitement à l'aise. En résumé, la précision juge le nombre de composants jetés pour rien, et le rappel le nombre de mauvais composants qui ont été ignorés. Ces métriques s'étendent aux problèmes de classification dans plusieurs catégories. On calcule dans ce cas la précision, l'accuracy et le rappel pour chaque classe.

6 – Le Deep Learning en pratique

6.1 - Structures des réseaux

Théoriquement, le MLP est à lui seul suffisamment puissant pour approcher n'importe quelle fonction que l'on souhaite apprendre. En pratique, il y a plusieurs prérogatives à cela : nous avons déjà évoqué l'importance des données, et l'influence des hyperparamètres. Quand bien même toutes ces conditions seraient réunies, il n'y a, en fait, aucune garantie pour que la descente de gradient parvienne à trouver les meilleures valeurs des poids possibles, et si possible dans un délai raisonnable. On doit donc se résoudre à inventer des structures de réseau de neurones qui facilitent l'apprentissage sur certaines tâches. C'est ce qu'on appelle la bitter lesson en apprentissage profond : tous les efforts que nous mettons en œuvre aujourd'hui pour inventer de nouvelles

structures de modèles profond est vain, car l'évolution de domaine et l'augmentation des puissances de calculs des ordinateurs auront permis l'émergence de nouvelles méthodes plus simples qui résoudront le problème avec au moins autant de succès.

Mais aujourd'hui, le MLP ne suffit pas ! Il est cependant toujours utilisé comme brique élémentaire de construction pour des modèles plus complexes. Éloignons-nous donc maintenant des principes introductifs pour nous intéresser aux pratiques plus modernes du deep learning. Voici un petit bestiaire des modèles classiques :

Les **réseaux convolutionnels** (CNN) (ou réseau de neurones convolutifs) sont bien plus adaptés au traitement de l'image que ne l'est le MLP. Pour cela, les transformations linéaires du MLP sont remplacées par des convolutions 2D, qui ont l'avantage d'utiliser moins de paramètres, et surtout d'être invariants aux translations (déplacer un objet dans l'image ne change pas drastiquement les calculs du réseau de neurone).

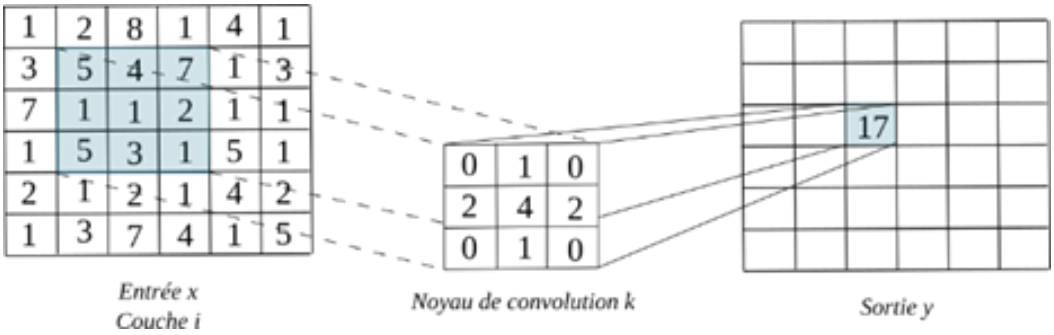

Figure 8 : Exemple d'une convolution 2D entre une matrice d'entrée x et un noyau de convolution k. Les seuls paramètres à calculer sont les poids du noyau de convolution

Les réseaux convolutionnels sont à l'origine du regain d'intérêt pour les réseaux de neurones au début des années 2010 en surpassant de très loin les performances des méthodes de vision par ordinateur plus classiques. La pratique courante consiste à appliquer une série de couches de convolution (avec leurs fonctions d'activation) sur une image, puis d'aplatir le résultat final en un long vecteur qui passe ensuite dans un MLP qui prédit alors la grandeur d'intérêt. Il existe également quelques utilisations de réseaux convolutionnels en 1D et 3D (pour l'audio et la vidéo par exemple).

Les réseaux récurrents (RNN) sont adaptés à l'analyse de séries temporelles, en ajoutant une mémoire au réseau. Chaque donnée x_t de la séquence est utilisée pour mettre à jour un vecteur caché h_t.

$$h_{t+1} = \sigma(W_h h_t + W_x x_t + b)$$

Ce vecteur caché peut être utilisé de plusieurs façons : pour une tâche de classification (par exemple, prédire si une action va prendre de la valeur ou non), on peut faire passer la dernière valeur de h_t dans un MLP. Ce principe est particulièrement utilisé par le traitement de flux vidéo et le traitement de l'écriture manuscrite. L'équation précédente est un peu passée de mode, et présente un mauvais comportement pendant la descente de gradient (le gradient à tendance à s'évanouir, c'est-à-dire valoir zéro, lorsque la séquence est longue). Il existe des structures plus modernes comme les Long Short-Term Memory (LSTM) ou plus récemment les réseaux Gated Recurrent Unit (GRU).

Les **auto-encodeurs** (AE) sont très pratiques dans beaucoup d'applications. Une donnée de grande dimension (généralement une image) est encodée par un réseau de neurones dans un vecteur contenant largement moins de valeurs, qui est ensuite décodé pour reconstruire la donnée d'origine. Il s'agit d'un algorithme de compression très efficace, qui peut être détourné pour, par

exemple, encoder une image et décoder l'image en profondeur correspondante. Cette structure en entonnoir est présente dans beaucoup de modèles profonds.

Figure 9 : Aucun de ces visages n'existent dans la vraie vie, ils ont été générés par un GAN, un modèle à deux réseaux : un générateur d'images, et un discriminateur chargé de distinguer les vrais visages des faux

Les **réseaux adversariaux génératif** (GAN) sont une idée astucieuse lorsque la tâche consiste à générer des données, comme des visages humains ou de la musique. Le principe consiste à entraîner deux réseaux de neurones antagonistes : le générateur fabrique des images à partir d'un vecteur d'entrée aléatoire et le discriminateur est entraîné à différencier une vraie donnée d'une donnée fabriquée par le générateur. Au début, la tâche est facile pour le discriminateur, mais se complique lorsque le générateur s'améliore. Les deux réseaux sont entraînés en simultané. Certains auto-encodeurs peuvent d'ailleurs être entraînés comme des GAN.

Tout récemment, une nouvelle structure a fait son apparition en révolutionnant les méthodes de traitement du langage : il s'agit des **transformers** (nous vous éviterons le jeu de mot avec les films de Michael Bay). Sans rentrer dans les détails, ces modèles reposent sur un mécanisme d'attention entre les mots d'une phrase. Ils sont à l'origine des immenses progrès des services de traduction en ligne, mais aussi de certains assistants virtuels.

La liste pourrait s'allonger indéfiniment, car c'est là l'une des libertés de l'apprentissage profond : toutes les structures sont bonnes à être testées. Le gros de la recherche se concentre donc sur la fabrication de modèles spécifiquement dédiés à une tâche. Nous pourrions mentionner par exemple les réseaux en graphe (GNN) qui excellent dans tous les problèmes pouvant se mettre sous cette forme particulière.

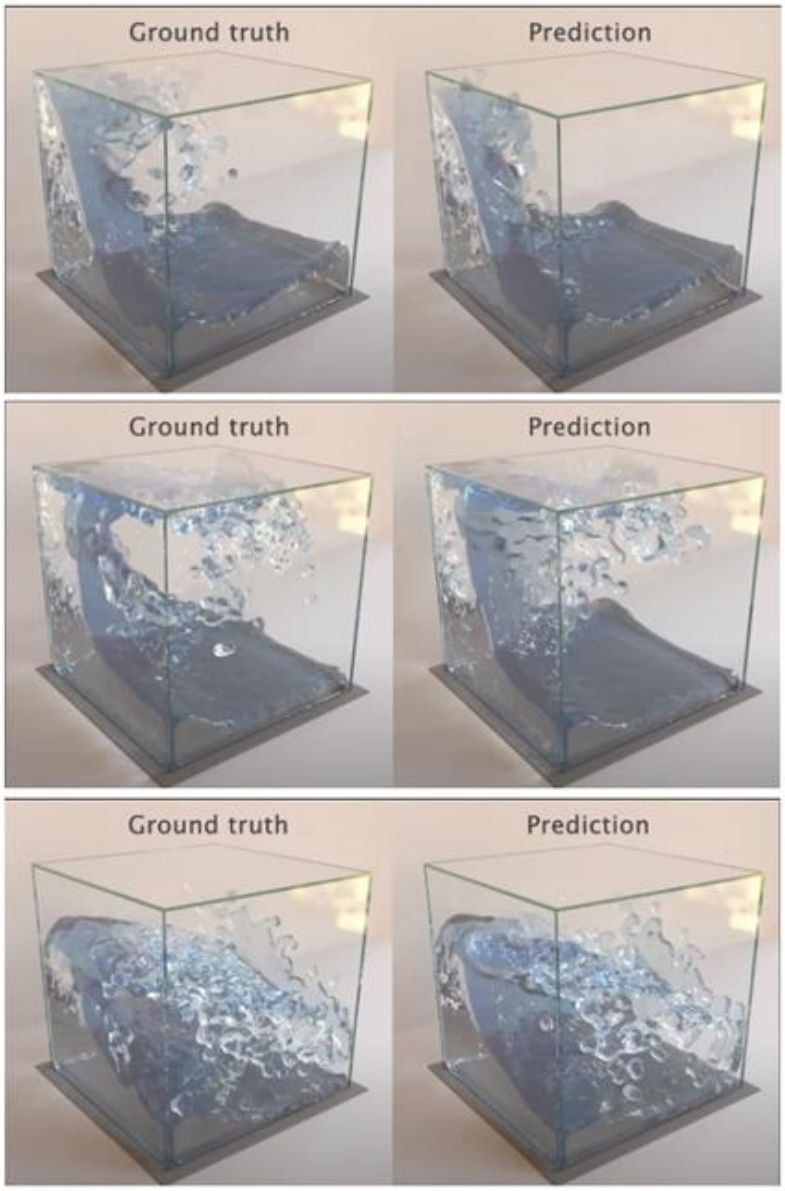

Figure 10 : Les mouvements du fluide prédit par un réseau de neurones en graphe, qui modélise le fluide par un ensemble de particules reliés entre eux, source [1]

6.2 – Implémentation pratique

Il faut différencier la phase d'apprentissage d'un réseau de neurones de la phase d'inférence, c'est-à-dire le moment où celui-ci est réellement utilisé. L'entraînement est une étape très longue, en grande partie à cause de la descente de gradient, qui nécessite de calculer la dérivée du critère d'optimisation par rapport à un très grand nombre de paramètres. L'avantage, c'est qu'il s'agit d'une étape qui se parallélise très bien, d'où l'utilisation massive des cartes graphiques. Architecturalement, une carte graphique est composée de milliers de cœurs très simples, là où un processeur n'est composé que d'une dizaine de cœurs en moyenne, mais bien plus puissants. Le GPU permet donc d'effectuer beaucoup plus d'opérations en parallèle qu'un CPU. L'inférence, par contre, est très rapide, puisqu'il s'agit majoritairement de multiplications matricielles, ce qui rend possible l'utilisation de réseaux de neurones en temps réel, dans des véhicules autonomes par exemple.

Dans cette partie, nous allons voir comment implémenter un MLP. Nous nous limiterons ici au langage Python, qui dispose de nombreuses bibliothèques facilitant la mise en œuvre de réseaux de neurones. À vrai dire, la quasi-totalité des implémentations de réseaux de neurones se fait avec

ce langage. Les bibliothèques les plus couramment utilisées pour l'implémentation de réseaux de neurones sont PyTorch (maintenu par Meta), Tensorflow (maintenu par Google). Ces deux bibliothèques diffèrent quelque peu dans leur manière de définir un réseau de neurones, mais s'appuient sur la même fonction de base : autograd. Il s'agit d'une bibliothèque permettant de dériver n'importe quelle fonction codée en Python (pour peu que l'on respecte certaines règles). Sans autograd, le deep learning ne connaîtrait pas le succès qu'on lui connaît. Ces deux bibliothèques sont gratuites et libres, et extrêmement bien documentées en ligne.

Pour cet exemple, nous allons plutôt utiliser Scikit- learn, une bibliothèque qui regroupe des centaines de fonctions liées au machine learning. On y retrouve bien sûr le MLP, que nous allons utiliser, mais aussi l'algorithme des k-plus-proches voisins, random forest, etc... Son utilisation est extrêmement simplifiée et elle est donc idéale pour comprendre le fonctionnement des algorithmes avec de petits modèles. Cependant, pour des problèmes plus complexes, l'utilisation de librairies spécialisées et optimisées comme PyTorch ou Tensorflow est obligatoire.

Dans un premier temps, on cherche à importer les données qui serviront à entraîner le réseau de neurones. Nous allons garder les chiffres manuscrits de la base de données MNIST mentionnées précédemment.

```
from sklearn.datasets import fetch_openml
import numpy as np
X, y = fetch_openml('mnist_784', version=1,return_X_y=True)
```

Ces quelques lignes téléchargent automatiquement les données de MNIST et stockent les entrées X et les sorties y dans les variables correspondantes. On peut ensuite visualiser une image. Notons que les images des chiffres manuscrits sont redimensionnées en 28×28 pixels :

```
import matplotlib.pyplot as plt
plt.gray()
plt.imshow(np.reshape(np.array(X)[0,:],(28,28)))
plt.show()
```

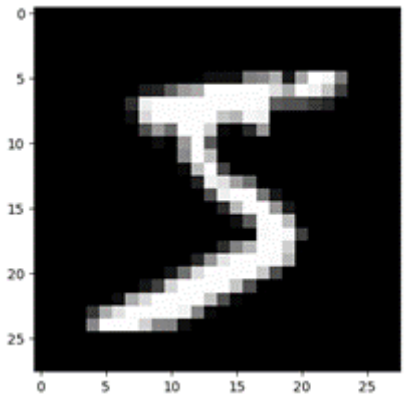

*Figure 11 : Image extraite de la base de données MNIST. Il s'agit d'une image 28*28 pixels, pouvant également être représentée comme un vecteur "plat" de 784 éléments*

On doit maintenant séparer les données de notre dataset en deux sous-ensembles : test et train, pour éviter tout risque de sur-apprentissage. Scikit-learn propose une fonction pour automatiser ce découpage. Notez que nous n'utilisons pas d'ensemble de validation dans cet exemple, ce n'est pas nécessaire dans la mesure où le problème est très simple. On sélectionne ici 20% des données pour constituer l'ensemble de test.

L'argument « shuffle » permet de mélanger les données afin de bien répartir équitablement chaque classe dans les deux ensembles. Cela permet de limiter le biais lié à la structure de la base de

données (Par exemple, tous les 9 pourraient être à la suite de tous les 8, eux-mêmes à la suite de tous les 7 etc.).

```
from sklearn.model_selection import train_test_split
X = X / 255.
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True,
test_size=0.2)
```

Une bonne pratique consiste à normaliser les entrées du réseau de neurones, cela a tendance à accélérer un peu l'apprentissage. On divise donc les images par 255. Les données sont maintenant prêtes, on peut créer le MLP et l'entraîner en lui fournissant les données connues. Plusieurs hyperparamètres sont précisés ici :

- Il y a 1 couche cachée de 50 neurones. On observe expérimentalement que cela suffit pour la reconnaissance de chiffres manuscrits ;
- On réalise au maximum 10 passages sur l'ensemble de la base de données (epoch) d'entraînement. Cela est suffisant pour obtenir de bons résultats dans notre cas mais peut être bien plus élevé pour des applications plus complexes ou si plus de paramètres sont à déterminer ;
- L'algorithme d'apprentissage utilisé est une descente de gradient stochastique.

```
from sklearn.neural_network import MLPClassifier

# Définition du modèle
mlp = MLPClassifier(hidden_layer_sizes=(50, ), max_iter=10, solver='sgd',
verbose=10, learning_rate_init=.1)

# Entrainement
model_final = mlp.fit(X_train, y_train)

# Calcul des précisions
print(f"Précision sur les données d'entraînement : {mlp.score(X_train,
y_train):.3f}")
print(f"Précision sur les données de test : {mlp.score(X_test, y_test):.3f}")
```

Enfin, on peut visualiser une des prédictions :

```
exemple_test_x = np.array(X_test[:1])
exemple_test_y = np.array(y_test[:1])
plt.gray()
plt.imshow(exemple_test_x.reshape((28,28)))
plt.show()
print(f"Prediction : {mlp.predict(exemple_test_x)}")
print(f"Sortie : {exemple_test_y}")
```

Le code de cet exemple de classification des chiffres manuscrits avec la bibliothèque Scikit-learn, ainsi que le code complet de l'apprentissage sans bibliothèque, sont disponibles sur le dépôt : https://github.com/LudovicDeMatteis/Revue3EI_IA

7 – Conclusion

Le deep learning est un domaine en pleine expansion qui permet d'obtenir des performances très élevées dans de nombreuses tâches, notamment la reconnaissance d'images et de parole. Il repose sur des architectures de réseaux de neurones particulièrement complexes, appelés réseaux profonds. Bien qu'il soit extrêmement performant, le deep learning n'est pas exempt de difficultés

et il est important de bien comprendre les principes de base avant d'en faire un usage aveugle. En particulier, il est important de disposer d'un jeu de données suffisamment important et représentatif pour pouvoir entraîner un réseau de neurones de manière efficace. Son apparition dans les programmes de l'éducation nationale permettra sans doute aux élèves et aux étudiants de mieux comprendre les principes fondamentaux du deep learning et plus généralement du machine learning, un sujet porteur qui interviendra nécessairement dans la carrière des futurs ingénieurs et techniciens. Il s'agit néanmoins d'une discipline jeune et en plein développement, il est important de suivre les évolutions en cours pour pouvoir tirer le meilleur parti des outils et des méthodes qui sont mis à disposition.

Cette ressource est une brève introduction aux réseaux de neurones, mais leur champ d'application grandit un peu plus chaque jour, offrant son lot de nouvelles applications et de nouveaux challenges à relever ! Pour être honnête, ces derniers mots ne sont pas les nôtres, mais en réalité ceux d'une IA, qui a proposé ce paragraphe à partir de quelques extraits de l'article que vous venez de lire. Il s'agit de GPT3, un modèle basé sur les transformers pour le traitement du langage naturel.

Références :

[1]: Figure 10 issue de "Learning to simulate complex physics with graph networks", de Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec et Peter Battaglia.

[2]: "L'apprentissage profond", Cours de Yann Le Cun au Collège de France, https://www.college-de-france.fr/site/yann-lecun/course-2015-2016.htm

[3]: "The bitter lesson", de Richard Sutton, http://www.incompleteideas.net/IncIdeas/BitterLesson.html

[4]: "Deep Learning" de Ian Goodfellow, Yoshua Bengio et Aaron Courville, éditions MIT press

[5]: TensorFlow Playground : https://playground.tensorflow.org/

[6]: Personalized Human-Swarm Interaction Through Hand Motion, M. Macchini, L. de Matteis, F. Schiano, D. Floreano, mars 2021, https://arxiv.org/abs/2103.07731

[7]: Dossier Intelligence Artificielle, juin 2022, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/dossier-intelligence-artificielle

[8]: Introduction à l'apprentissage automatique, L. de Mateïs, S. Nathan, juin 2022, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/introduction-lapprentissage-automatique