

28/11/2023

Le Deep Learning, une introduction

Mattéo Eléouet

matteo.eleouet@gmail.com

YouTube : Mattéo Robino

ABSTRACT

Version 0.4.8

Table des matières

1 Introduction	7
1.1 À propos de l'auteur	7
2 Fondamentaux du Deep Learning	8
2.1 Introduction au Deep Learning	8
2.2 Structure d'un réseau de neurones artificiels (ANN)	10
2.3 Apprentissage supervisé vs non supervisé vs semi-supervisé vs apprentissage par renforcement	11
2.4 Régression et Classification	13
2.5 Les types de données : images, texte, séries temporelles	14
2.6 Comment un modèle de machine learning apprend	17
2.7 Résumé	18
2.8 Questions	20
3 Réseaux de neurones artificiels (Artificial Neural Network)	24
3.1 Le Perceptron	24
3.2 Introduction aux fonctions d'activation	26
3.3 Le Biais dans un Perceptron	32
3.4 Le Perceptron multicouches (MLP)	34
3.5 Le théorème d'approximation universelle	37
3.6 Résumé	38
3.7 Question	39
4 Processus d'apprentissage avec la rétropropagation (Backpropagation)	44
4.1 Composantes clés du processus d'apprentissage	44
4.2 La fonction de perte (loss function)	46
4.3 Descente de gradient	54
4.4 La rétropropagation (Backpropagation)	68
4.5 Résumé	79
4.6 Questions	81
4.7 Réponse	84
5 Projet: Projet avec des MLP et mesurer ses performances	86
5.1 Projet 1: Créer une calculatrice	86
5.2 Projet 2: Reconnaissance des chiffres manuscrits	102
6 Régularisation	105
6.1 Introduction à la régularisation	105
6.2 Techniques de régularisation	107
Bibliographie	110

Table des matières

1 Introduction	7
1.1 À propos de l'auteur	7
2 Fondamentaux du Deep Learning	8
2.1 Introduction au Deep Learning	8
2.1.1 Qu'est-ce que le Deep Learning ?	8
2.1.2 Le lien entre le Deep Learning et le Machine Learning, en quoi le deep learning est une évolution du machine learning	9
2.2 Structure d'un réseau de neurones artificiels (ANN)	10
2.2.1 La couche d'entrée (input layer)	11
2.2.2 Les couches cachées (hidden layer)	11
2.2.3 La couche de sortie (output layer)	11
2.3 Apprentissage supervisé vs non supervisé vs semi-supervisé vs apprentissage par renforcement	11
2.3.1 Apprentissage supervisé (Supervised Learning)	11
2.3.2 Popularité et efficacité de l'apprentissage supervisé	12
2.3.3 Apprentissage non supervisé (Unsupervised Learning)	12
2.3.4 Apprentissage semi-supervisé (Semi-supervised Learning)	13
2.3.5 Apprentissage par renforcement (Reinforcement Learning)	13
2.4 Régression et Classification	13
2.4.1 Régression	13
2.4.2 Classification	14
2.5 Les types de données : images, texte, séries temporelles	14
2.5.1 Les images	14
2.5.2 Le Texte	15
2.5.3 Les Séries Temporelles	16
2.5.4 Les Données Structurées	16
2.6 Comment un modèle de machine learning apprend	17
2.6.1 Apprentissage supervisé (supervised learning)	17
2.6.2 Apprentissage non supervisé (unsupervised learning)	17
2.6.3 Apprentissage semi-supervisé (Semi-supervised learning)	18
2.6.4 L'Apprentissage par Renforcement: Apprentissage Par Essais et Récompenses ..	18
2.7 Résumé	18
2.8 Questions	20
2.8.1 Correction	23
3 Réseaux de neurones artificiels (Artificial Neural Network)	24
3.1 Le Perceptron	24
3.1.1 Un perceptron avec les poids	24
3.2 Introduction aux fonctions d'activation	26
3.2.1 Introduction de Non-linéarités	26
3.2.2 Fonction d'activation sigmoïde	27
3.2.3 Décision sur la Contribution à la Sortie	28
3.2.4 Fonction tangente hyperbolique (tanh)	28
3.2.5 Fonction d'activation de Rectification (ReLU)	29

3.2.6 Fonction d'activation Softmax	30
3.2.7 Normalisation de la Sortie	32
3.3 Le Biais dans un Perceptron	32
3.4 Le Perceptron multicouches (MLP)	34
3.4.1 Propagation avant (feed-forward)	34
3.5 Le théorème d'approximation universelle	37
3.6 Résumé	38
3.7 Question	39
3.7.1 Correction	42
4 Processus d'apprentissage avec la rétropropagation (Backpropagation)	44
4.1 Composantes clés du processus d'apprentissage	44
4.2 La fonction de perte (loss function)	46
4.2.1 Introduction à la fonction de perte (loss function)	46
4.2.2 Les différents types de fonction perte	47
4.3 Descente de gradient	54
4.3.1 Descente de Gradient en 1D	55
4.3.2 Impact de la Taille du Pas (Learning Rate)	55
4.3.3 Descente de gradient en 2D	57
4.3.4 Exemple mathématique de la descente de gradient	58
4.3.5 Qu'est-ce qu'un minimum local ?	63
4.3.6 Les points de selle (saddle points)	65
4.3.7 La descente de gradient stochastique, l'apprentissage par lot	67
4.4 La rétropropagation (Backpropagation)	68
4.4.1 La backpropagation dans le processus d'apprentissage	69
4.4.2 Les Principes Fondamentaux de la Backpropagation	70
4.4.3 L'apprentissage d'un réseau avec la Backpropagation : pas à pas	71
4.4.4 Les problèmes du vanishing gradient et exploding gradient	76
4.4.5 Avenir de la Backpropagation dans le Deep Learning	78
4.5 Résumé	79
4.6 Questions	81
4.7 Réponse	84
5 Projet: Projet avec des MLP et mesurer ses performances	86
5.1 Projet 1: Créer une calculatrice	86
5.1.1 Cahier des charges	86
5.1.2 importer vos bibliothèques python et configurer votre matériel.	87
5.1.3 Le jeu de données	88
5.1.4 Architecture du réseau	88
5.1.5 Fonctions d'entraînement et de préparation	93
5.1.6 Entraînement et visualisation des résultats	95
5.1.7 Test et Visualisation: Révéler l'Âme du Modèle	99
5.2 Projet 2: Reconnaissance des chiffres manuscrits	102
6 Régularisation	105
6.1 Introduction à la régularisation	105
6.1.1 Qu'est-ce que la régularisation ?	106
6.1.2 Pénalités des Normes des Paramètres : Une Introduction	106

6.2 Techniques de régularisation	107
6.2.1 Régularisation L1 et L2 (Weight Decay)	108
Bibliographie	110

1 Introduction

1.1 À propos de l'auteur

 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat voluptatem. Ut enim aequo doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.

2 Fondamentaux du Deep Learning

2.1 Introduction au Deep Learning

2.1.1 Qu'est-ce que le Deep Learning ?

Le deep learning, est une sous-branche du machine learning lui-même étant une sous branche de l'IA, le deep learning se base sur l'imitation du fonctionnement du cerveau humain¹ pour apprendre, prendre des décisions et résoudre des problèmes. Il s'agit d'une approche avancée du machine learning, qui va au-delà des techniques traditionnelles pour atteindre une meilleure précision et une plus grande complexité dans les tâches.

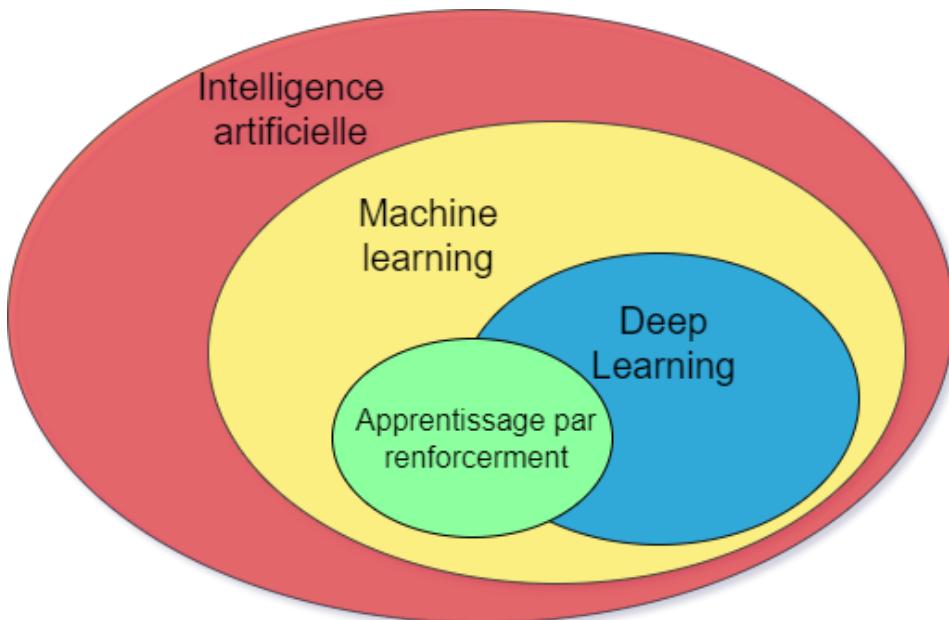


Figure 1 : Relation entre l'Intelligence artificielle, le Machine Learning, l'apprentissage par renforcement et le Deep Learning. l'IA, le Machine Learning, l'Apprentissage par Renforcement, et le Deep Learning comme des ensembles imbriqués.

Ce diagramme illustre comment l'Intelligence Artificielle (IA), le Machine Learning (ML), l'Apprentissage par Renforcement (RL) et le Deep Learning (DL) sont liés entre eux.

L'IA est le domaine le plus vaste qui englobe toutes les techniques permettant à des machines de simuler des comportements « intelligents ». Le Machine Learning est une sous-catégorie de l'IA qui se concentre sur les méthodes permettant aux algorithmes de s'améliorer à partir de données, un « apprentissage ».

L'Apprentissage par Renforcement est une autre sous-catégorie de l'IA qui implique l'apprentissage par essai-erreur, où un agent apprend à réaliser une tâche en maximisant la récompense obtenue à travers ses interactions avec son environnement. Il convient de noter que l'Apprentissage par Renforcement peut être vu à la fois comme une branche du Machine Learning et comme une technique qui peut être combinée avec le Deep Learning, donnant naissance à ce que l'on appelle l'Apprentissage par Renforcement Profond (Deep reinforcement learning).

¹Ouais attention, c'est grossier comme statement

Le deep learning est un sous-ensemble du machine learning qui s'appuie sur les réseaux de neurones artificiels, qui sont des modèles informatiques inspirés des connexions neuronales du cerveau humain. Ces modèles sont composés de différentes couches de neurones artificiels, chacune étant capable de traiter une partie spécifique de l'information. L'information est introduite dans la première couche (la couche d'entrée), puis est traitée et transmise de couche en couche (couche cachée) jusqu'à la dernière (la couche de sortie). Chaque couche « apprend » de l'information de la couche précédente, la modifie et la transmet à la suivante. C'est ce processus d'apprentissage qui permet à un réseau de neurones de faire des prédictions précises ou de prendre des décisions en fonction des données d'entrée.

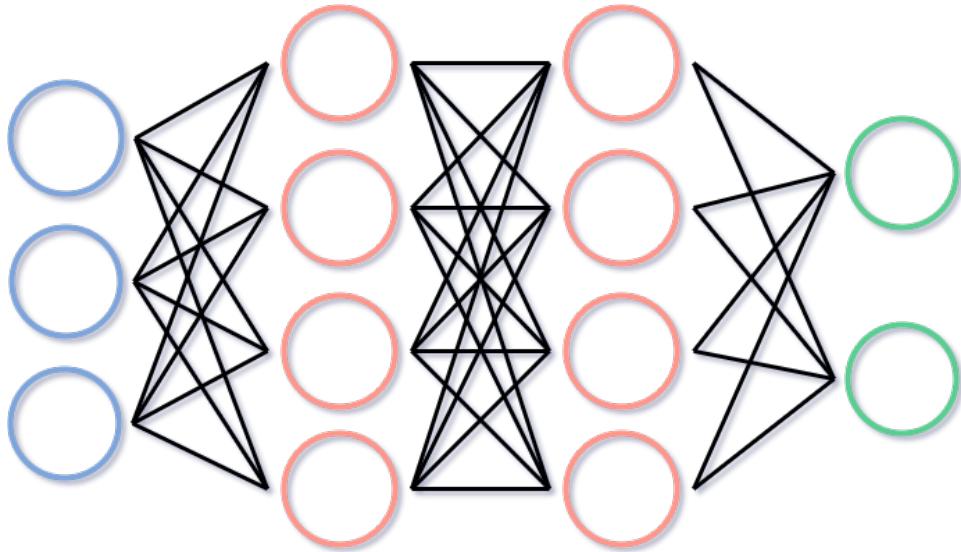


Figure 2 : diagramme représentant un réseau de neurones artificiels

Ce diagramme illustre la structure typique d'un réseau de neurones artificiels. La couche d'entrée, représentée en bleu, reçoit des informations dérivées de données d'entrée, comme les valeurs de pixels d'une image. Ces informations sont ensuite traitées par les couches cachées (en rose), avec chaque neurone (cercle) recevant des informations de la couche précédente, effectuant un calcul simple et transmettant le résultat à la couche suivante via des connexions neuronales (lignes noires). Ce processus se poursuit jusqu'à la couche de sortie. Du point de vue du diagramme, l'information circule de gauche à droite à travers le réseau.

Dans ce livre, le terme « machine learning » sera utilisé pour englober l'ensemble des techniques de machine learning, y compris le deep learning. Cependant, si je souhaite spécifiquement me référer au machine learning traditionnel en excluant le deep learning, j'emploierai le terme de « machine learning classique ».

2.1.2 Le lien entre le Deep Learning et le Machine Learning, en quoi le deep learning est une évolution du machine learning

Le deep learning est une sous-branche du machine learning classique, ils utilisent tous les deux le même paradigme , celui de l'apprentissage à partir de données.

Le machine learning n'est pas explicitement programmé par un humain, c'est-à-dire, qu'il n'y a pas d'humain pour mettre des conditions à toutes étapes pour créer un algorithme comme le font les systèmes experts. Le machine learning classique apprend à partir de données, par

exemple on pourrait entraîner à prédire le prix d'une maison en fonction de ses caractéristiques (comme la superficie, le nombre de chambres, l'emplacement, etc.), ou à classer les courriels comme « spam » ou « non-spam ».

Le deep learning lui aussi apprend à partir de données, il ne sera alors pas programmé, expliqué par un humain, mais utilise un type d'algorithme propre à lui, les réseaux de neurones avec plusieurs couches (avec une seule couche, cela n'est pas considéré comme « deep »). À l'origine créée par G. Hinton, Y. LeCun et Y. Bengio et ses collègues, ces réseaux de neurones, tentent de simuler le comportement du cerveau humain - d'où l'emploi du terme « neurone artificiel ».

Un avantage du deep learning est qu'il peut identifier lui-même les caractéristiques importantes dans les données, une tâche généralement appelée « feature engineering » en machine learning classique, auquel il faut créer les variables qui seront importantes, que l'on a défini à l'aide de méthodes statistiques afin de retirer celles qui le seront moins dans le but d'aider le modèle à apprendre. Grâce à ses couches cachées, le deep learning est capable de modéliser des structures de données complexes sans intervention humaine préalable. Il apprend lui-même quelles variables sont importantes ou non en attribuant une pondération à chacune.

La possibilité d'ajouter plusieurs couches cachées crée de la « profondeur », permettant de réaliser des algorithmes d'une complexité croissante. Chaque couche apprend une représentation de plus en plus complexe à partir des données d'entrée, et les représentations apprises par chaque couche sont ensuite utilisées par la couche suivante pour modéliser un résultat. Cette capacité à gérer des données non structurées, comme des images ou du texte, est l'avantage clé qui distingue le deep learning du machine learning classique. En général, pour des données structurées, le machine learning classique peut généralement être une solution plus appropriée en utilisant une bibliothèque de machine learning classique comme XGBoost.

Des algorithmes ayant une forte complexité nécessitent de grandes quantités de données pour l'entraînement des modèles. Les entraînements sont gourmands en puissance de calculs et en temps . Un exemple d'un cas extrême : l'équivalent « ChatGPT » conçu par Méta, LLaMA[1], a été entraîné sur 2048 cartes graphiques (Nvidia Tesla A100 ayant une mémoire vive de 80 Go et d'une valeur de 20'000 € sur le commerce en 2023), durant cinq mois d'affilée.

Le machine learning classique et le deep learning s'appuient tous les deux sur l'apprentissage à partir de données afin d'identifier des motifs, des schémas, etc, le deep learning n'est pas forcément la meilleure solution à un problème de machine learning. En effet, le choix du modèle dépend de la complexité algorithmique nécessaire, de la quantité de données et de plusieurs autres facteurs tels que la criticité du domaine ciblé, le système sur lequel il sera déployé, ...

2.2 Structure d'un réseau de neurones artificiels (ANN)

Un réseau de neurones artificiels (ANN, pour Artificial Neural Network) est un modèle computationnel qui s'inspire de la manière dont les réseaux neuronaux biologiques du cerveau humain traitent l'information. Composé de neurones interconnectés, un ANN apprend à partir des données d'entrée pour effectuer une variété de tâches, allant de la simple classification à la génération de contenu complexe.

2.2.1 La couche d'entrée (input layer)

La couche d'entrée, comme son nom l'indique, est le point de départ de tout réseau de neurones. Chaque neurone dans cette couche correspond à une caractéristique spécifique de la donnée d'entrée. Si le réseau est dédié au traitement d'images, chaque neurone pourrait représenter la valeur d'un pixel. Dans un contexte de prévision météorologique, chaque neurone pourrait correspondre à des facteurs variés tels que la température, l'humidité ou la vitesse du vent.

2.2.2 Les couches cachées (hidden layer)

Après la couche d'entrée, viennent les couches cachées d'un réseau de neurones. Ces couches sont nommées « cachées » parce qu'elles ne sont pas visibles à l'extérieur du réseau. Les couches cachées sont le lieu où la majorité du traitement se déroule. Chaque noeud dans une couche cachée reçoit des entrées de tous les noeuds de la couche précédente. Ces entrées sont ensuite pondérées, sommées, et transmises à une fonction qui génère la sortie du noeud.

Bien que les couches cachées soient nommées ainsi, elles ne sont pas secrètes ni inaccessibles. Le terme « caché » est plutôt utilisé pour indiquer qu'elles ne sont pas directement visibles dans l'interface utilisateur ou dans les sorties finales du modèle, c'est le côté « black box » du deep learning.

Un réseau de neurones peut avoir n'importe quel nombre de couches cachées, et chaque couche peut avoir n'importe quel nombre de noeuds. Le choix de la structure exacte dépend de la complexité du problème à résoudre. En règle générale, plus le problème est complexe, plus il y aura de couches cachées et de noeuds.

2.2.3 La couche de sortie (output layer)

La dernière couche d'un réseau de neurones est la couche de sortie. Cette couche a pour mission de produire les résultats ou prédictions finales du réseau. Comme dans les couches cachées, chaque noeud de la couche de sortie reçoit des entrées de tous les noeuds de la couche précédente. Cependant, le nombre de noeuds dans la couche de sortie dépend généralement du type de problème que le réseau est destiné à résoudre. Par exemple, pour une tâche de classification binaire, il n'y aurait qu'un seul noeud dans la couche de sortie, tandis que pour une tâche de classification multiclasse, il y aurait un noeud pour chaque classe possible.

2.3 Apprentissage supervisé vs non supervisé vs semi-supervisé vs apprentissage par renforcement

2.3.1 Apprentissage supervisé (Supervised Learning)

L'apprentissage supervisé est un paradigme du machine learning dans lequel un modèle est entraîné à partir d'un ensemble de données étiquetées. Dans cet ensemble de données, chaque exemple (échantillon) comporte des entrées jumelées à la réponse attendue, dénommée « étiquette ». Deux types principaux de problèmes sont résolus par l'apprentissage supervisé : la classification et la régression.

La classification consiste à prédire une catégorie ou une classe à partir d'un ensemble défini (par exemple, déterminer si un email est un spam ou non), tandis que la régression consiste à

prédir une valeur continue (par exemple, prédire le prix d'une maison à partir de ses caractéristiques).

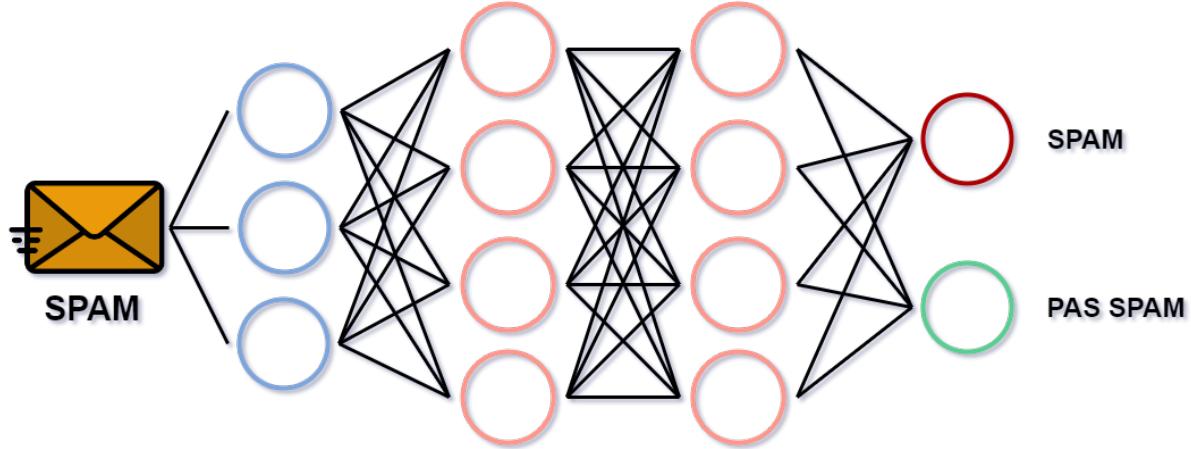


Figure 3 : diagramme représentant un réseau de neurones artificiels faisant de la classification en supervised learning

Description du diagramme: Ce diagramme représente le processus d'apprentissage supervisé. Les données d'entrée sont associées à des étiquettes spécifiques, et ces paires d'entrées et d'étiquettes sont utilisées pour entraîner le modèle. Le modèle apprend alors à prédire l'étiquette correcte à partir des données d'entrée.

2.3.2 Popularité et efficacité de l'apprentissage supervisé

L'apprentissage supervisé est souvent considéré comme le plus populaire parmi les différentes méthodes d'apprentissage du machine learning, principalement en raison de sa performance. Cette méthode a démontré sa capacité à résoudre une grande variété de problèmes complexes dans divers domaines, allant de la reconnaissance d'images à la prédition financière.

La nature des données étiquetées utilisées en apprentissage supervisé offre un avantage considérable : la possibilité de fournir au modèle des informations claires et précises sur les résultats attendus, ce qui améliore considérablement son habileté à réaliser des prédictions précises et fiables. Par conséquent, bien que d'autres formes d'apprentissage machine continuent d'évoluer et d'apporter des contributions significatives, l'apprentissage supervisé reste le pilier du domaine en raison de sa puissance et de son efficacité démontrées.

2.3.3 Apprentissage non supervisé (Unsupervised Learning)

L'apprentissage non supervisé est un autre paradigme de machine learning où, contrairement à l'apprentissage supervisé, les modèles sont entraînés sur des données non étiquetées. Le but est de découvrir des structures ou des relations cachées dans les données. Les problèmes d'apprentissage non supervisé incluent généralement la réduction de dimensionnalité, la détection d'anomalies, et le regroupement (clustering).

Le clustering, par exemple, vise à diviser l'ensemble de données en groupes distincts de manière que les données dans chaque groupe soient similaires entre elles et différentes des données des autres groupes. Par exemple, vous pourrez faire du clustering pour regrouper un groupe d'utilisateur pour un système de recommandation.

2.3.4 Apprentissage semi-supervisé (Semi-supervised Learning)

L'apprentissage semi-supervisé est un paradigme de machine learning qui combine les éléments des méthodes supervisées et non supervisées. Dans cette approche, un modèle est entraîné sur un ensemble de données partiellement étiquetées - un grand ensemble de données d'entrée non étiquetées, avec un petit ensemble de données d'entrée étiquetées.

L'idée est que le modèle peut apprendre de la structure globale des données non étiquetées pour aider à prédire les étiquettes pour ces données, tout en utilisant les données étiquetées pour guider le processus d'apprentissage. C'est une approche efficace lorsque l'étiquetage des données est coûteux ou chronophage, la difficulté de trouvé des données étiquetées sera le nerf de la guerre dans l'industrie.

2.3.5 Apprentissage par renforcement (Reinforcement Learning)

L'apprentissage par renforcement est un paradigme de machine learning où un agent apprend à prendre des décisions en interagissant avec son environnement. Dans cette approche, un agent effectue certaines actions dans un environnement et reçoit des récompenses ou des punitions (feedback) en retour. L'objectif de l'agent est d'apprendre une politique, qui est une stratégie pour choisir des actions qui maximisent la récompense cumulative à long terme.

Contrairement à l'apprentissage supervisé ou non supervisé, l'apprentissage par renforcement est basé sur l'idée d'apprendre par l'expérience et par l'interaction. Il est souvent utilisé pour les problèmes où un agent doit prendre une série de décisions et où la récompense pour une action peut être retardée. Les exemples d'applications comprennent les jeux (comme les échecs ou le Go), la navigation de robots, et le trading algorithmique.

2.4 Régression et Classification

2.4.1 Régression

La régression, est utilisée lorsque la variable de sortie est une quantité continue. Dans ce cas, l'objectif est de prédire une valeur numérique précise. Les exemples courants de problèmes de régression comprennent la prédition des prix des maisons, la prévision des températures, ou l'estimation de la durée d'un événement.

L'un des modèles de régression les plus couramment utilisés dans le deep learning est le réseau de neurones à régression (Regression Neural Network). La dernière couche de ce réseau de neurones est un seul neurone avec une fonction d'activation linéaire, produisant une sortie continue.

L'erreur entre les prédictions et les valeurs réelles est mesurée par une fonction de coût, comme l'erreur quadratique moyenne (Mean Squared Error - MSE) ou l'erreur absolue moyenne (Mean Absolute Error - MAE).

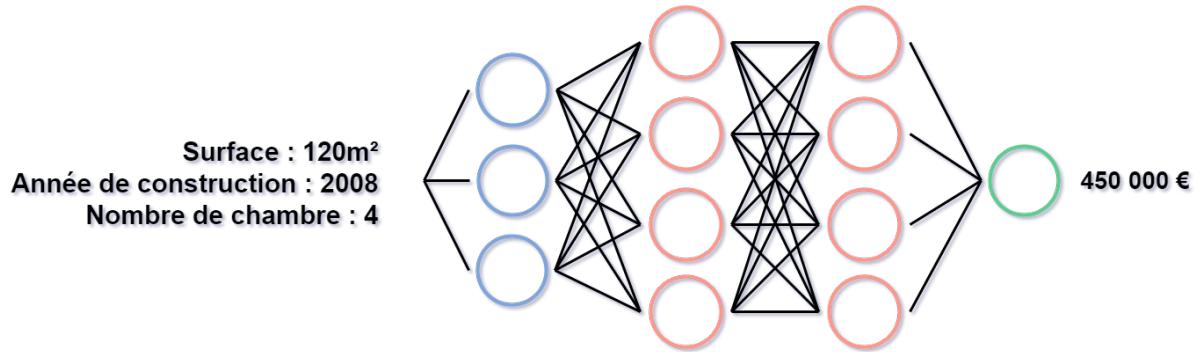


Figure 4 : diagramme représentant un réseau de neurones artificiels faisant de la régression.

Description du diagramme: Ce diagramme illustre un réseau de neurones à régression. Il prend plusieurs entrées, les fait passer par des couches cachées (où l'apprentissage a lieu), puis produit une sortie continue par le biais d'un seul neurone dans la couche de sortie.

2.4.2 Classification

La classification en deep learning, en revanche, est utilisée lorsque la variable de sortie est une catégorie. Dans ce cas, l'objectif est de prédire la classe ou la catégorie à laquelle appartient une entrée donnée. Les exemples courants de problèmes de classification comprennent la détection de spam, la reconnaissance d'images, ou la prédiction de la survie des passagers du Titanic.

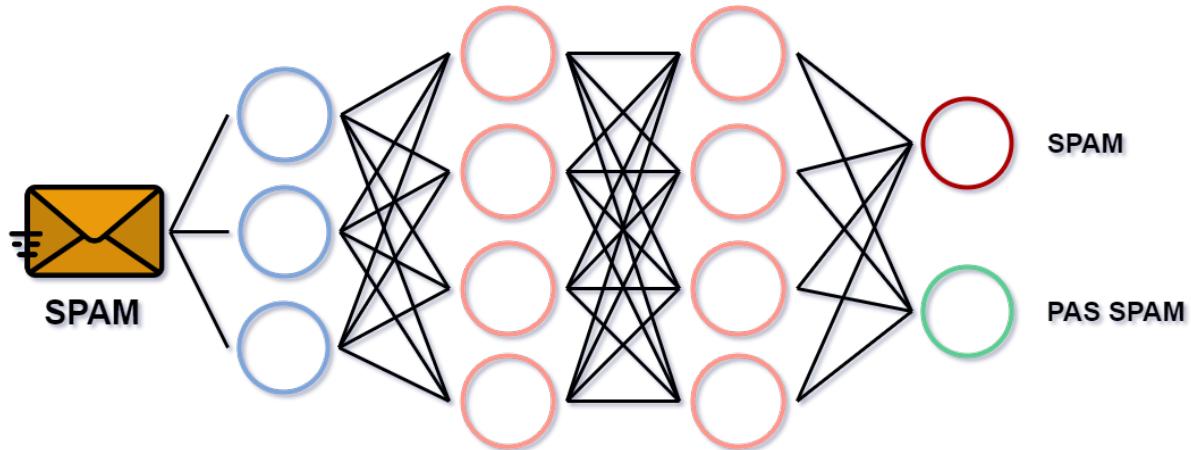


Figure 5 : diagramme représentant un réseau de neurones artificiels faisant de la classification en supervised learning

Description du diagramme: Ce diagramme représente le processus d'apprentissage supervisé. Les données d'entrée sont associées à des étiquettes spécifiques, et ces paires d'entrées et d'étiquettes sont utilisées pour entraîner le modèle. Le modèle apprend alors à prédire l'étiquette correcte à partir des données d'entrée.

2.5 Les types de données : images, texte, séries temporelles

2.5.1 Les images

Les images sont l'un des types de données les plus couramment utilisés en deep learning. Elles sont généralement représentées sous forme de matrices, où chaque cellule de la matrice correspond à un pixel de l'image. Les images en niveaux de gris sont représentées par une matrice

2D, où chaque cellule contient une valeur entre 0 et 255 représentant l'intensité de gris. Les images en couleur sont généralement représentées en format RGB (Red, Green, Blue), qui est une matrice 3D où chaque pixel est représenté par trois valeurs correspondant aux intensités des couleurs rouge, verte et bleue.

Dans le cas des images, les packages Python comme PIL ou OpenCV sont souvent utilisés pour le prétraitement des images. Pour PyTorch, une image est transformée en tenseur, avec des dimensions correspondant à (Channels, Height, Width), où Channels correspond aux canaux de couleur (3 pour RGB, 1 pour les images en niveaux de gris), et Height et Width correspondant à la hauteur et à la largeur de l'image en pixels.

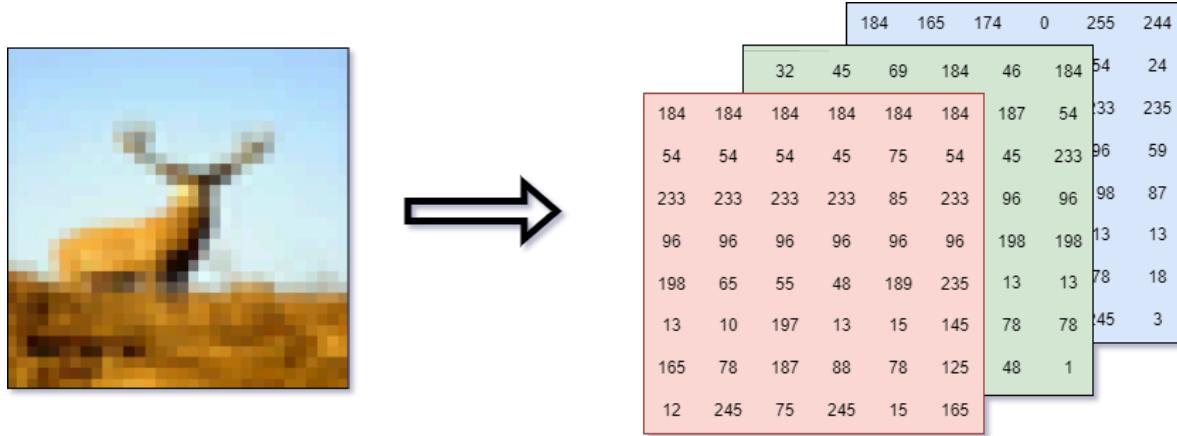


Figure 6 : Image du jeu de données CIFAR-10, les valeurs sont des valeurs à aléatoire.

Description du diagramme: Le diagramme représente une image en couleur sous forme de matrice 3D. Chaque pixel est présenté par trois valeurs, chacune correspondant à l'intensité des couleurs rouge, verte et bleue.

2.5.2 Le Texte

Le texte est un autre type de données largement utilisé en deep learning. Les données textuelles sont généralement représentées sous forme de séquences de mots ou de caractères. Pour l'encodage, des techniques comme l'encodage one-hot, l'embedding de mots (Word Embedding) ou l'encodage par des vecteurs de mots (Word2Vec, GloVe) sont souvent utilisées.



Figure 7 : Illustration de l'encodage d'une séquence de texte

Description du diagramme: Le diagramme illustre comment une séquence de texte est encodée. Chaque mot est transformé en un vecteur à travers une opération d'embedding.

2.5.3 Les Séries Temporelles

Les séries temporelles sont des données collectées à intervalles réguliers sur une période de temps. Elles sont largement utilisées dans de nombreux domaines tels que la finance, la météorologie et la santé. En deep learning, les séries temporelles sont majoritairement traitées avec des modèles de type séquence tels que les réseaux de neurones récurrents (RNN) ou les Transformers, nous verrons ces architectures de deep learning plus loin dans le livre.

2.5.4 Les Données Structurées

Les données structurées sont un type de données qui suit un modèle ou un format prédéfini, facilitant ainsi leur analyse et leur traitement. Elles peuvent être stockées dans des bases de données relationnelles ou des fichiers csv, xlsx et sont généralement organisées en colonnes et en lignes. Chaque colonne représente une caractéristique (ou un « attribut ») et chaque ligne représente un exemple (ou une « observation »).

Un exemple classique de données structurées est un tableau de données contenant des informations sur des individus, où chaque colonne représente une caractéristique telle que l'âge, le sexe, le revenu, etc., et chaque ligne représente une personne spécifique.

Les données structurées peuvent être traitées avec réseaux de neurones, qui modélise des relations entre les caractéristiques. Pour préparer ces données pour l'apprentissage, les caractéristiques numériques sont souvent normalisées (c'est-à-dire mises à l'échelle pour avoir une moyenne de 0 et un écart-type de 1), tandis que les caractéristiques catégorielles sont généralement encodées en utilisant des techniques comme l'encodage one-hot.

ID	Age	Sexe	Profession	Revenu annuel (K€)
1	25	F	Ingénieur	50
2	32	M	Médecin	70
3	45	F	Professeur	45
4	23	M	Étudiant	13
5	55	F	Retraité	40

Figure 8 : Illustration d'un jeu de données structurées

Description du diagramme: Le diagramme illustre une table de données structurées, avec chaque colonne représentant une caractéristique différente et chaque ligne représentant une observation distincte. Certaines des caractéristiques sont numériques, tandis que d'autres sont catégorielles.

Les bibliothèques Python comme Pandas sont souvent utilisées pour le prétraitement des données structurées. Pour PyTorch, une table de données structurées est transformée en tenseur, avec des dimensions correspondant à (Nombre d'exemples, Nombre de caractéristiques).

2.6 Comment un modèle de machine learning apprend

2.6.1 Apprentissage supervisé (supervised learning)

Imaginez que vous essayez d'apprendre à jouer au golf pour la première fois. Au début, vos coups sont aléatoires et imprécis. Vous pouvez comparer cela à un modèle de machine learning avant son entraînement : il fait des prédictions, mais elles sont souvent très éloignées de la vérité.

L'essentiel de l'apprentissage, que ce soit pour vous en tant que golfeur ou pour un modèle de machine learning, réside dans la correction des erreurs.

Quand vous jouez au golf, vous obtenez immédiatement un feedback. Si la balle a dévié à gauche, vous savez que vous devez ajuster votre swing pour la prochaine fois. Dans un modèle de machine learning, ce feedback prend la forme de ce que nous appelons une « fonction de perte » (pour une compréhension intuitive, vous pouvez simplement l'appeler « score d'erreur »). C'est une mesure qui permet au modèle de savoir à quel point il s'est trompé. Plus ce score est élevé, plus le modèle s'est trompé dans sa prédiction. Nous parlerons plus en détails de la fonction de perte au chapitre du processus d'apprentissage.

2.6.2 Apprentissage non supervisé (unsupervised learning)

Bien que l'apprentissage non supervisé puisse sembler mystérieux au premier abord, il peut être illustré de manière simple. Imaginez que vous donnez à un enfant un ensemble de blocs de différentes formes et couleurs sans lui fournir d'instructions précises. Avec le temps, vous remarquerez que l'enfant commence à classer les blocs par couleur ou par forme, à les empiler selon leur taille, ou encore à les aligner par ordre de taille croissante ou décroissante. L'enfant, en observant et en découvrant des motifs, apprend de ses propres observations. C'est exactement le processus que suit un modèle de machine learning dans le cadre de l'apprentissage non supervisé.

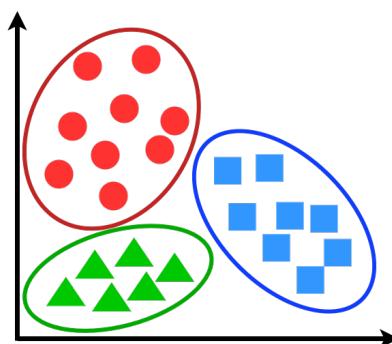


Figure 9 : Diagramme de clustering à 3 classe

Description du diagramme: Le diagramme montre la classification d'un cluster d'un apprentissage non supervisé via les formes.

Lorsqu'un modèle de machine learning « apprend », il ne fait pas vraiment ce que nous, en tant qu'êtres humains, faisons lorsque nous apprenons quelque chose. Au lieu de cela, il optimise une fonction - une formule mathématique, si vous voulez - qui décrit la « performance » du modèle sur les données d'entraînement. Cette fonction est communément appelée « loss function », ou fonction de coût.

La fonction de coût est un peu comme un GPS pour le modèle. Si vous pensez à votre position actuelle comme étant l'état actuel du modèle, alors la fonction de coût est la distance entre votre position actuelle et votre destination - l'objectif d'apprentissage du modèle.

2.6.3 Apprentissage semi-supervisé (Semi-supervised learning)

L'apprentissage semi-supervisé est un concept intéressant qui se situe quelque part entre l'apprentissage supervisé et l'apprentissage non supervisé. Pour comprendre ce concept, imaginez un enfant jouant avec des blocs de différentes formes et couleurs. Cette fois, cependant, un adulte est présent et donne occasionnellement à l'enfant des conseils sur comment jouer avec les blocs. Par exemple, l'adulte pourrait montrer à l'enfant comment empiler les blocs de la même taille ou les aligner par couleur. Toutefois, la majeure partie du temps, l'enfant est libre d'explorer et d'apprendre par lui-même. C'est essentiellement ce que fait un modèle de machine learning dans un contexte d'apprentissage semi-supervisé.

Dans le contexte de l'apprentissage semi-supervisé, une partie des données d'entraînement est étiquetée, tandis que l'autre partie ne l'est pas. Le modèle apprend à partir des deux : il utilise les données étiquetées pour comprendre certaines structures des données, et il utilise les données non étiquetées pour explorer et découvrir d'autres structures par lui-même.

2.6.4 L'Apprentissage par Renforcement: Apprentissage Par Essais et Récompenses

Imaginez un enfant qui apprend à jouer à un jeu vidéo pour la première fois. Au début, l'enfant ne connaît pas les règles du jeu et fait beaucoup d'erreurs. Mais avec le temps, il commence à comprendre comment le jeu fonctionne. Il apprend que certaines actions entraînent des points ou des récompenses, tandis que d'autres peuvent entraîner des pénalités ou des pertes de vie. Ce processus d'apprentissage par essais et erreurs, guidé par des récompenses et des pénalités, est l'essence de l'apprentissage par renforcement.

Dans le cadre de l'apprentissage par renforcement, un agent d'apprentissage (le modèle) interagit avec un environnement (comme le jeu vidéo), fait des actions, reçoit des feedbacks sous forme de récompenses ou de pénalités, et apprend à travers ce processus. L'objectif de l'agent est de maximiser la somme totale des récompenses qu'il reçoit au fil du temps. Pour ce faire, l'agent doit apprendre quelle est la meilleure action à faire dans chaque situation ou état de l'environnement.

Contrairement à l'apprentissage supervisé et semi-supervisé, l'apprentissage par renforcement n'a pas de paires d'entrées/sorties étiquetées à partir desquelles apprendre directement. Au lieu de cela, l'agent apprend de l'expérience - il fait des actions, voit les résultats, et ajuste son comportement en conséquence.

Un concept clé dans l'apprentissage par renforcement est la politique. Une politique est essentiellement une stratégie que l'agent utilise pour décider quelle action faire dans chaque état. L'agent commence souvent par une politique aléatoire, puis l'ajuste au fil du temps pour favoriser les actions qui ont tendance à donner de meilleures récompenses.

2.7 Résumé

Dans ce chapitre, nous avons vu qu'est-ce que c'était le deep learning, qu'il était une sous-division de l'IA et du machine learning. Les modèles de deep learning ont une profondeur et une complexité surpassant largement celles de leurs homologues du machine learning. De plus,

le deep learning permet une extraction de caractéristiques de manière automatique et hiérarchique, contrairement au machine learning traditionnel qui nécessite une extraction manuelle des caractéristiques.

Le chapitre explore également les quatre types d'apprentissage en machine learning : supervisé, non supervisé, semi-supervisé et par renforcement. L'apprentissage supervisé et non supervisé fonctionne avec des données étiquetées et non étiquetées respectivement. L'apprentissage semi-supervisé combine les deux, tandis que l'apprentissage par renforcement fonctionne par interaction avec l'environnement.

La régression et la classification sont expliquées comme deux types de tâches courantes en apprentissage supervisé, la régression prévoyant des valeurs continues et la classification prévoyant des catégories.

Enfin, le chapitre met l'accent sur les différents types de données traitées en deep learning : les images, le texte et les séries temporelles. Les images sont généralement représentées sous forme de matrices, où chaque cellule correspond à un pixel. Le texte, quant à lui, est généralement représenté sous forme de séquences de mots ou de caractères

L'apprentissage supervisé, non supervisé, semi-supervisé et par renforcement sont les quatre grandes catégories d'apprentissage dans le domaine du machine learning. Chacun d'eux se réfère à un ensemble unique de stratégies et de techniques d'apprentissage, illustrées par des analogies telles que jouer au golf, jouer avec des blocs de construction ou jouer à un jeu vidéo.

Dans l'apprentissage supervisé, un modèle apprend à partir de données étiquetées, tout comme un golfeur apprend en recevant un feedback après chaque coup. L'apprentissage non supervisé, en revanche, ressemble plus à un enfant qui joue avec des blocs sans instructions explicites, découvrant les structures et les motifs par lui-même. L'apprentissage semi-supervisé est un équilibre entre les deux, comme un enfant jouant avec des blocs, mais recevant occasionnellement des conseils d'un adulte. Ces trois types d'apprentissage s'appuient sur l'idée d'une fonction de coût, qui guide l'apprentissage du modèle en lui fournissant un score d'erreur pour chaque prédiction.

Enfin, l'apprentissage par renforcement est un processus d'apprentissage dynamique guidé par des récompenses et des pénalités. C'est comme un enfant apprenant à jouer à un jeu vidéo, ajustant constamment son comportement pour maximiser le score total. Dans cette approche, l'agent d'apprentissage (le modèle) interagit activement avec son environnement, apprend de l'expérience et ajuste progressivement sa politique - une stratégie pour décider quelle action entreprendre dans chaque état.

2.8 Questions

1. **Quelle est la description la plus précise du Deep Learning ?**
 - a. C'est une approche avancée du Machine Learning qui vise à imiter le fonctionnement du cerveau humain pour apprendre et résoudre des problèmes.
 - b. C'est un enchaînement de condition if et des else vrai ou faux pour résoudre des problématiques.
 - c. C'est une méthode de statistique qui utilise des algorithmes pour modéliser les données.
 - d. C'est un sous-domaine de l'IA qui se concentre uniquement sur l'analyse du langage naturel de type ChatGPT.
2. **Le diagramme de la figure 1 des relations entre l'IA, le ML et le DL illustre que :**
 - a. Le Machine Learning et le Deep Learning sont complètement séparés.
 - b. Le Machine Learning est un sous-ensemble du Deep Learning.
 - c. L'IA, le Machine Learning et le Deep Learning sont des sous-ensembles les uns des autres.
 - d. Le Deep Learning et l'IA ne sont pas liés.
3. **Quelle est la différence principale entre le Machine Learning et le Deep Learning ?**
 - a. Le Deep Learning nécessite des données manuellement étiquetées tandis que le Machine Learning ne le fait pas.
 - b. Le Deep Learning utilise des réseaux de neurones artificiels plus complexes et peut effectuer une extraction de caractéristiques de manière automatique et hiérarchique.
 - c. Le Machine Learning est une approche plus moderne que le Deep Learning.
 - d. Le Machine Learning peut traiter des données de grande dimension et non structurées, tandis que le Deep Learning ne le peut pas.
4. **Qu'est-ce que l'apprentissage supervisé ?**
 - a. C'est une méthode où un modèle apprend à prendre des décisions en interagissant avec son environnement.
 - b. C'est une méthode qui combine les éléments des méthodes supervisées et non supervisées.
 - c. C'est une méthode où un modèle est entraîné sur des données non étiquetées pour découvrir des structures ou des relations cachées dans les données.
 - d. C'est une méthode où un modèle est entraîné à partir d'un ensemble de données étiquetées.
5. **En quoi consiste la classification dans le contexte de l'apprentissage supervisé ?**
 - a. Prédire une catégorie ou une classe à partir d'un ensemble défini.
 - b. Prédire une valeur continue.
 - c. Diviser l'ensemble de données en groupes distincts.
 - d. Maximiser la récompense cumulative à long terme.
6. **Qu'est-ce qui décrit le mieux l'apprentissage non supervisé ?**
 - a. Les modèles sont entraînés sur des données étiquetées pour résoudre des problèmes spécifiques.

- b. Les modèles sont entraînés pour prendre des décisions en interagissant avec leur environnement.
- c. Les modèles sont entraînés sur des données non étiquetées pour découvrir des structures cachées.
- d. Les modèles sont entraînés à partir d'un ensemble de données partiellement étiquetées.

7. Quel type de tâche en deep learning utiliserait-on pour prédire le prix d'une maison?

- a. Classification
- b. Régression
- c. Les deux
- d. Aucun des deux

8. Comment les images en couleur sont-elles représentées pour le deep learning

- a. Par une matrice 3D où chaque pixel est représenté par trois valeurs correspondant aux intensités des couleurs rouge, verte et bleue
- b. Par une matrice 2D où chaque cellule contient une valeur entre 0 et 255
- c. Par une seule valeur correspondant à l'intensité de la couleur la plus dominante
- d. Elles ne sont pas représentées, elles sont utilisées telles quelles

9. Qu'est-ce que l'apprentissage supervisé ?

- a. Un modèle apprend sans aucune guidance ou label
- b. Un modèle apprend en recevant des feedbacks sur ses erreurs et en ajustant ses prédictions
- c. Un modèle apprend par essai et erreur dans un environnement avec récompenses et pénalités
- d. Un modèle apprend en recevant des feedbacks occasionnels tout en explorant et découvrant par lui-même

10. Qu'est-ce que l'apprentissage non supervisé ?

- a. Un modèle apprend sans aucune guidance ou label, découvrant les patterns dans les données par lui-même
- b. Un modèle apprend par essai et erreur dans un environnement avec récompenses et pénalités
- c. Un modèle apprend en recevant des feedbacks sur ses erreurs et en ajustant ses prédictions
- d. Un modèle apprend en recevant des feedbacks occasionnels tout en explorant et découvrant par lui-même

11. Qu'est-ce que l'apprentissage par renforcement ?

- a. Un modèle apprend en recevant des feedbacks sur ses erreurs et en ajustant ses prédictions
- b. Un modèle apprend par essai et erreur dans un environnement avec récompenses et pénalités
- c. Un modèle apprend sans aucune guidance ou label, découvrant les patterns dans les données par lui-même

- d. Un modèle apprend en recevant des feedbacks occasionnels tout en explorant et découvrant par lui-même

2.8.1 Correction

1. Quelle est la description la plus précise du Deep Learning ?

Réponse: a C'est une approche avancée du Machine Learning qui vise à imiter le fonctionnement du cerveau humain pour apprendre et résoudre des problèmes

2. Le diagramme avec l'image « Relation_IA_ML_DL.png » illustre que :

Réponse: c L'IA, le Machine Learning et le Deep Learning sont des sous-ensembles les uns des autres.

3. Quelle est la différence principale entre le Machine Learning et le Deep Learning ?

Réponse: b Le Deep Learning utilise des réseaux de neurones artificiels plus complexes et peut effectuer une extraction de caractéristiques de manière automatique et hiérarchique.

4. Qu'est-ce que l'apprentissage supervisé ?

Réponse: d C'est une méthode où un modèle est entraîné à partir d'un ensemble de données étiquetées.

5. En quoi consiste la classification dans le contexte de l'apprentissage supervisé ?

Réponse: a Prédire une catégorie ou une classe à partir d'un ensemble défini.

6. Qu'est-ce qui décrit le mieux l'apprentissage non supervisé ?

Réponse: c Les modèles sont entraînés sur des données non étiquetées pour découvrir des structures cachées.

7. Quel type de tâche en deep learning utiliserait-on pour prédire le prix d'une maison?

Réponse: b Régression

8. Comment les images en couleur sont-elles généralement représentées pour le deep learning

Réponse: a Par une matrice 3D où chaque pixel est représenté par trois valeurs correspondant aux intensités des couleurs rouge, verte et bleue

9. Qu'est-ce que l'apprentissage supervisé ?

Réponse: b Un modèle apprend en recevant des feedbacks sur ses erreurs et en ajustant ses prédictions

10. Qu'est-ce que l'apprentissage non supervisé ?

Réponse: a Un modèle apprend sans aucune guidance ou label, découvrant les patterns dans les données par lui-même

11. Qu'est-ce que l'apprentissage par renforcement ?

Réponse: b Un modèle apprend par essai et erreur dans un environnement avec récompenses et pénalités

3 Réseaux de neurones artificiels (Artificial Neural Network)

Dans ce chapitre nous rentrons plus en détails dans les réseaux de neurones. Nous parlerons du perceptron qui est un neurone artificiel, des différentes fonctions d'activations utilisées qui permettent de casser la linéarité des perceptrons et d'avoir des relations complexes dans les données, nous parlerons de la structure d'un réseau de neurones, du fonctionnement des perceptrons entre eux et du théorème d'approximation universelle, qui démontre la capacité des réseaux de neurones à modéliser une vaste gamme de fonctions continues.

3.1 Le Perceptron

Un perceptron est un neurone artificiel ou une unité de traitement dans un réseau neuronal. Les perceptrons sont utilisés comme un bloc de base dans les réseaux de neurones profonds formant la base des modèles de deep learning actuels.

3.1.1 Un perceptron avec les poids

La structure d'un perceptron est relativement simple. Il comprend plusieurs entrées, chacune étant pondérée par un poids spécifique. Ces poids déterminent l'importance relative de chaque entrée pour la sortie. Un biais est également ajouté pour ajuster la sortie indépendamment des entrées. La sortie d'un perceptron est obtenue en sommant le produit de chaque entrée pondérée par son poids. Nous aborderons le rôle du biais en détail dans la suite, commençons sans pour facilier les calculs.

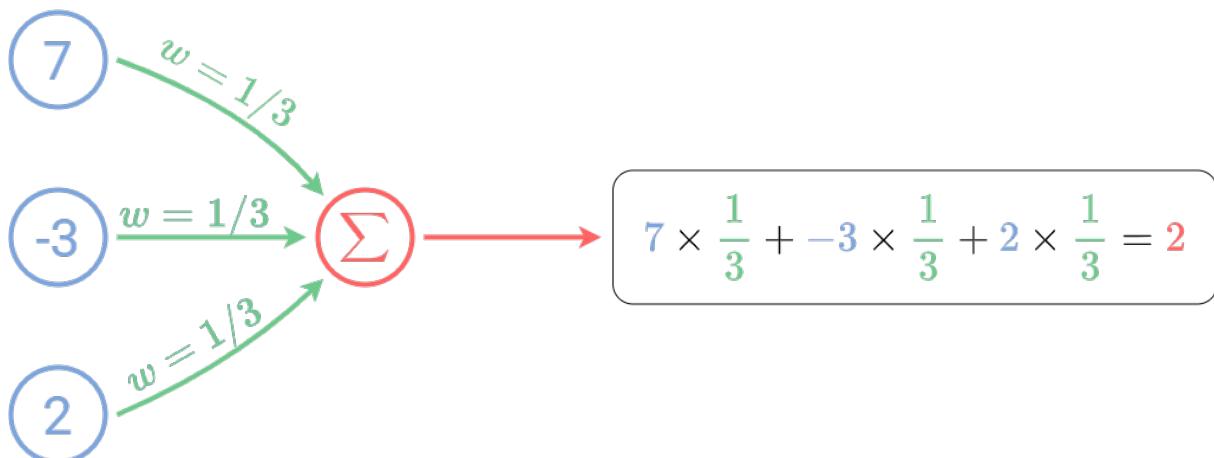


Figure 10 : Perceptron sans biais qui calcule la moyenne des entrées.

Description du diagramme: Ici les valeurs d'entrées sont : 7, -3 et 2 elles ont chacun un poids de $\frac{1}{3}$ ce qui transforme ce perceptron en machine... à calculer la moyenne ! $7 \times \frac{1}{3} + -3 \times \frac{1}{3} + 2 \times \frac{1}{3} = 2$ ici la sortie de ce perceptron est 2.

```
x = [7, -3, 2] # Valeurs d'entrée x
w = [1/3, 1/3, 1/3] # Poids w
output = sum(x[i] * w[i] for i in range(len(x)))
print(output) # L'output du perceptron est égal à 2
```

Voici un deuxième exemple où les valeurs sont différentes.

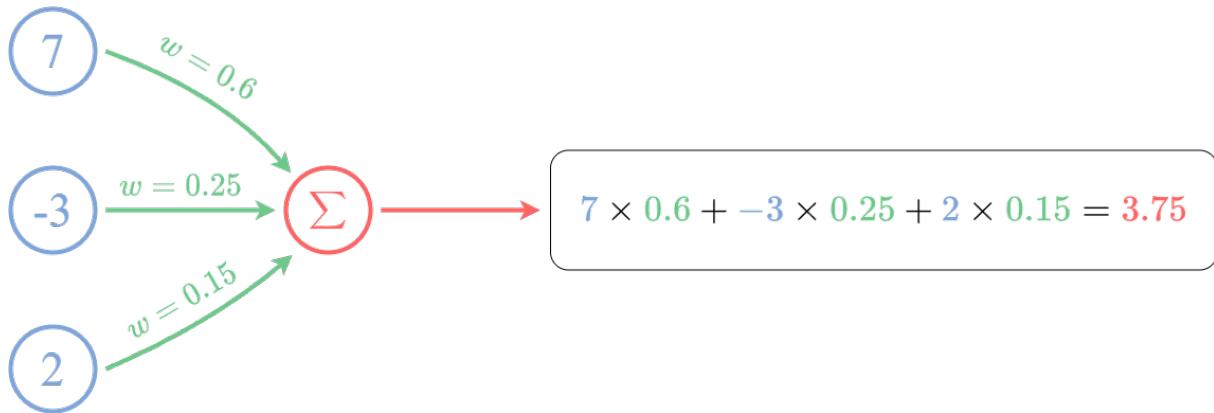


Figure 11 : Perceptron sans biais qui calcule la sortie avec des poids différents.

Description du diagramme: Ici les valeurs d'entrées sont les mêmes, mais ils ont des poids différents. Voici le calcul est $7 \times 0.6 + -3 \times 0.25 + 2 \times 0.15 = 3.75$ ici la sortie de ce perceptron est 3.75.

Ici les entrées sont $x = \begin{pmatrix} 7 \\ -3 \\ 2 \end{pmatrix}$ les poids sont $w = \begin{pmatrix} 0.6 \\ 0.25 \\ 0.15 \end{pmatrix}$ on remplace les inconnus par les valeurs de x et $w \rightarrow \hat{y} = \begin{pmatrix} 7 \\ -3 \\ 2 \end{pmatrix}^T \times \begin{pmatrix} 0.6 \\ 0.25 \\ 0.15 \end{pmatrix} = 3.75$ Nous venons d'effectuer un produit scalaire.

```
import numpy as np
x = np.array([7, -3, 2]) # Vecteur d'entrée x
w = np.array([0.6, 0.25, 0.15]) # Vecteur de poids w

output = np.dot(x.T, w) # sur numpy cela s'appelle dot car un produit scalaire
# en anglais se nomme "dot product"
print(output) # L'output du perceptron est égal à 3.75
```

La formule mathématique de ces calculs serait $\hat{y} = x^T w$ où \hat{y} la sortie du perceptron et x^T est la transposée de x

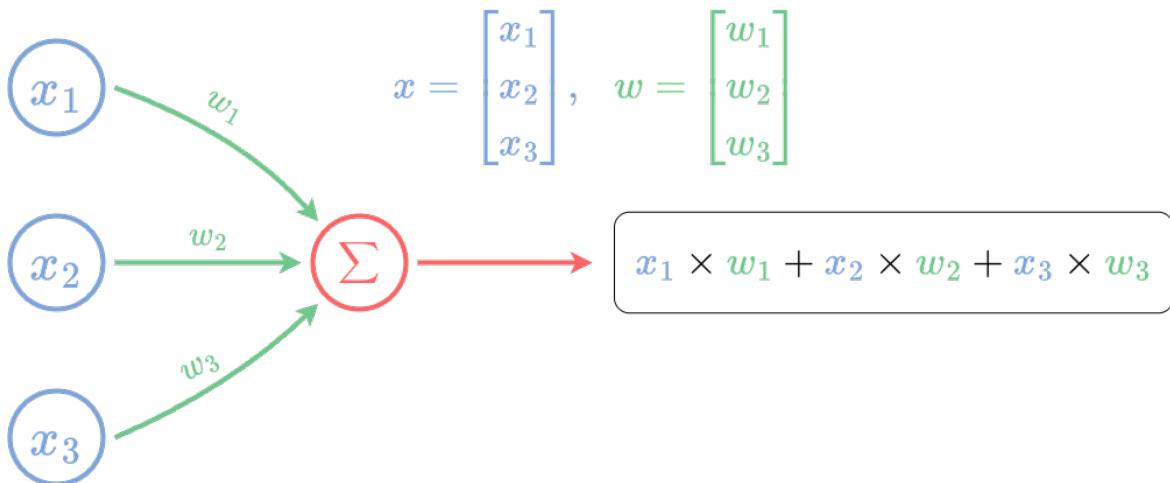


Figure 12 : Calcule mathématique d'un perceptron sans biais.

Ce qu'il faut comprendre c'est plus la valeur d'un poids est élevée plus son influence sur le perceptron sera grande, c'est de cette manière-là que dans le deep learning s'effectue le *feature engineering* qui consiste à choisir les variables importantes. Par exemple, si vous deviez définir le prix d'un bien immobilier à partir de plusieurs variables, vous pourriez penser aux nombres de pièces, la couleur des murs, mais la variable du nombre du m^2 serait probablement très importante pour définir le prix d'un bien immobilier, elle aurait alors plus d'importance que celle de la couleur des murs.

3.2 Introduction aux fonctions d'activation

Une fonction d'activation est un élément essentiel des réseaux de neurones artificiels, notamment des perceptrons. Elle est utilisée pour transformer la sortie pondérée de la somme des entrées d'un neurone. Cette transformation peut aider à introduire des non-linéarités dans le modèle, permettre à chaque neurone de prendre une décision sur sa contribution à la sortie globale et aider à normaliser la sortie d'un neurone.

3.2.1 Introduction de Non-linéarités

Jusque-là nos perceptrons étaient linéaires, pour faire simple notre perceptron ce sont des additions et des multiplications, ce qui est mathématiquement linéaire, graphiquement, la linéarité, c'est une droite.

Les fonctions d'activation sont non-linéaires. C'est une caractéristique clé qui permet aux réseaux de neurones d'apprendre à partir de données complexes et non linéaires. Sans les non-linéarités introduites par les fonctions d'activation, un réseau de neurones serait essentiellement un modèle linéaire, limité dans sa capacité à traiter des relations plus complexes entre les entrées et les sorties.

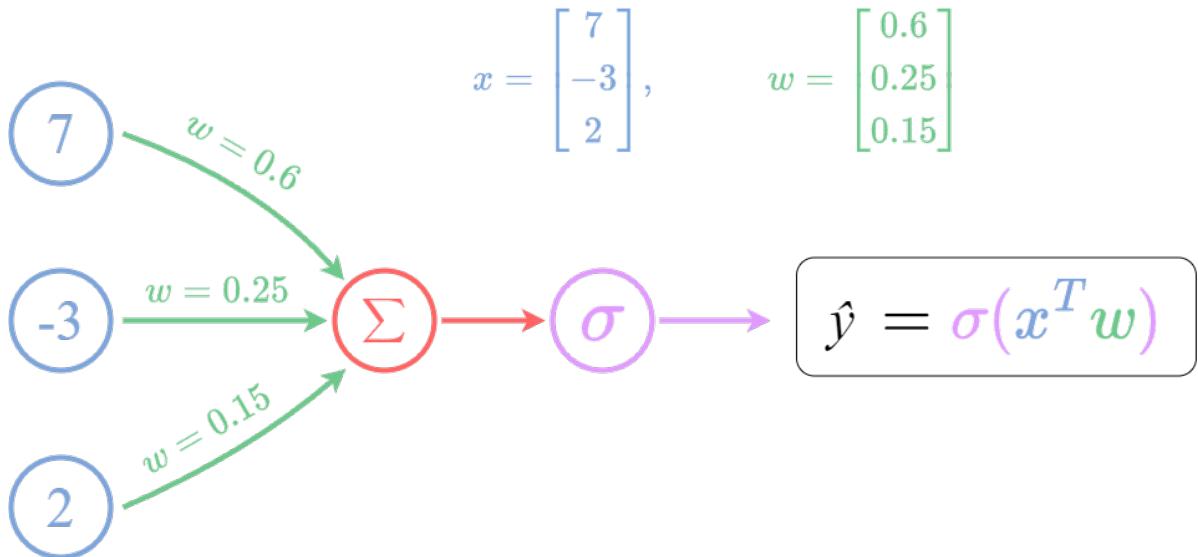


Figure 13 : Perceptron sans biais avec une fonction d'activation en sortie représentée par σ .

3.2.2 Fonction d'activation sigmoïde

La fonction d'activation sigmoïde, également appelée fonction logistique, est une fonction couramment utilisée en deep learning, elle introduit une non-linéarité et produit des sorties dans une plage spécifique entre 0 et 1, graphiquement, la sigmoïde a une forme en « S » caractéristique.

La formule mathématique de la sigmoïde est la suivante :

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

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

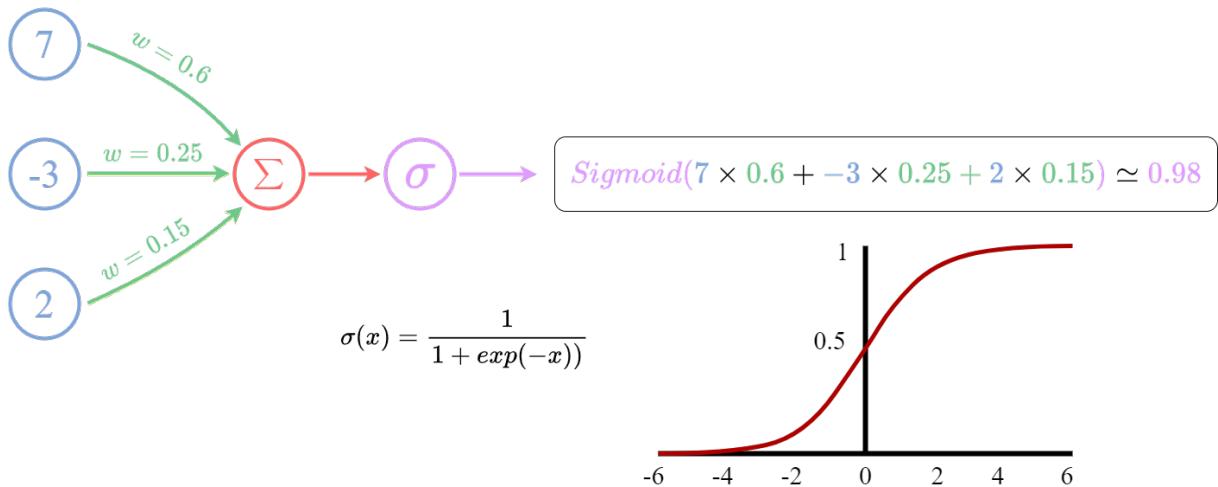


Figure 14 : Perceptron sans biais qui calcule la sortie avec la fonction d'activation sigmoïde.

Description du diagramme: Ici la valeur de la sortie est 3.75 et devient 0.98 avec la fonction d'activation Dans cette formule, σ représente la valeur de sortie de la sigmoïde pour une valeur

d'entrée x . La fonction $\exp(x)$ est la fonction exponentielle, qui élève le nombre e (environ 2,71828) à la puissance x . L'opérateur « $-$ » devant x indique que nous prenons l'opposé de x .

La fonction sigmoïde était largement utilisée dans les anciennes architectures de réseaux de neurones. Cependant, elle a été supplantée par d'autres fonctions d'activation plus performantes dans de nombreux scénarios. La sigmoïde peut entraîner des problèmes de saturation lorsque les valeurs d'entrée sont très élevées ou très basses, ce qui peut conduire à une convergence lente et à une atténuation du gradient lors de l'entraînement. Nous aborderons l'apprentissage en détail dans le chapitre suivant dédié à ce sujet.

3.2.3 Décision sur la Contribution à la Sortie

Les fonctions d'activation permettent également à chaque neurone de prendre une décision concernant sa contribution à la sortie du réseau. Par exemple, dans le cas de la fonction d'activation sigmoïde, qui transforme chaque entrée en une valeur entre 0 et 1, un neurone peut décider d'activer ou non sa sortie en fonction de la valeur de la sortie pondérée.

Si la sortie pondérée est très positive, la fonction sigmoïde produira une valeur proche de 1, ce qui signifie que le neurone est « activé » et contribue pleinement à la sortie du réseau. Si la sortie pondérée est très négative, la fonction sigmoïde produira une valeur proche de 0, ce qui signifie que le neurone est « désactivé » et ne contribue pas vraiment à la sortie du réseau.

3.2.4 Fonction tangente hyperbolique (tanh)

La fonction d'activation Tangente Hyperbolique, abrégée en Tanh. Elle est utilisée pour transformer la somme pondérée des entrées d'un neurone en une sortie qui peut être ensuite transmise aux autres neurones du réseau.

La principale caractéristique de la tanh est sa plage sortie, entre -1 et 1 contrairement à la fonction d'activation sigmoïde, qui produit une sortie entre 0 et 1 . Cette plage, centrée autour de zéro, est bénéfique lors de l'apprentissage de notre modèle, si les sorties sont toutes positives ou négatives, ce qui produit un phénomène appelé le « biais de notification » (reporting bias), ralentissant l'apprentissage du réseau.

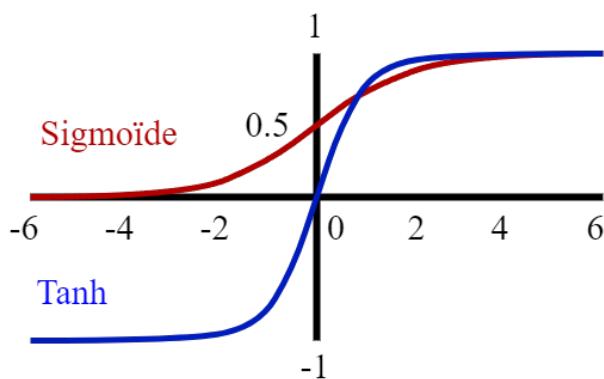


Figure 15 : Courbe de la tanh et de la sigmoïde.

Une bonne fonction d'activation a une dérivée facile à calculer, celle de la tanh est simple à calculer pour un ordinateur ce qui sera très utile lors de l'algorithme de la rétropropagation (backpropagation) qui utilise les dérivées des fonctions d'activations pour ajuster les paramètres du modèle lors de l'apprentissage, nous verrons cela plus en détail plus tard.

Cependant, Tanh n'est pas sans défauts. En particulier face au problème du gradient qui disparaît (vanishing gradient), qui arrive souvent sur des réseaux ayant de nombreuses couches cachées, surtout quand la dérivée de la fonction d'activation est proche de 0. Ce qui arrive à la tanh lorsque les sorties sont proches de 1 ou de -1. Vous comprendrez mieux cela au chapitre prochain lorsque nous verrons le processus d'apprentissage.

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

```
# x est la sortie du perceptron
def tanh(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
```

3.2.5 Fonction d'activation de Rectification (ReLU)

La fonction Rectified Linear Unit, appelée ReLU, est probablement la fonction d'activation la plus utilisée dans le deep learning. La ReLU est définie comme la fonction qui renvoie l'entrée si elle est positive et zéro dans le cas contraire. En d'autres termes, elle « rectifie » les valeurs négatives en les mettant à zéro.

La formule mathématique est très simple :

$$\sigma(x) = \max(0, x)$$

```
def relu(x):
    return np.maximum(0, x)
```

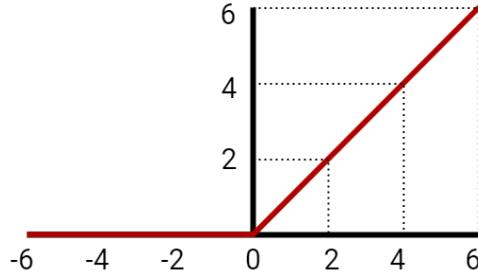


Figure 16 : Courbe de la ReLU.

Cette simplicité a plusieurs avantages qui font de la ReLU une fonction d'activation extrêmement populaire dans le deep learning. Premièrement, ReLU est très facile à calculer, ce qui peut aider à accélérer le processus d'apprentissage. Deuxièmement, sa dérivée est également simple : elle est de 1 pour les entrées positives et de 0 pour les entrées négatives, ce qui rend la rétropropagation plus efficace.

La ReLU est robuste au vanishing gradient, ce problème se produit lorsque les gradients deviennent très petits au fur et à mesure qu'ils sont rétropropagés à travers le réseau, de sorte que les poids de certaines couches du réseau ne sont pratiquement pas mis à jour pendant l'apprentissage. Comme la dérivée de ReLU est toujours 1 pour les entrées positives, elle permet de propager efficacement ces gradients et d'éviter ce problème.

Cependant, il faut noter que ReLU n'est pas sans défauts. L'un des problèmes majeurs est le « dying ReLU » : si un neurone donne une sortie négative, la dérivée de ReLU pour ce neurone sera 0, ce qui signifie que ce neurone n'apprendra plus rien pendant la rétropropagation. En

d'autres termes, le neurone « meurt ». Plusieurs variantes de ReLU ont été proposées pour résoudre ce problème, comme Leaky ReLU, Parametric ReLU et GELU existent, mais il y en a plein d'autre.

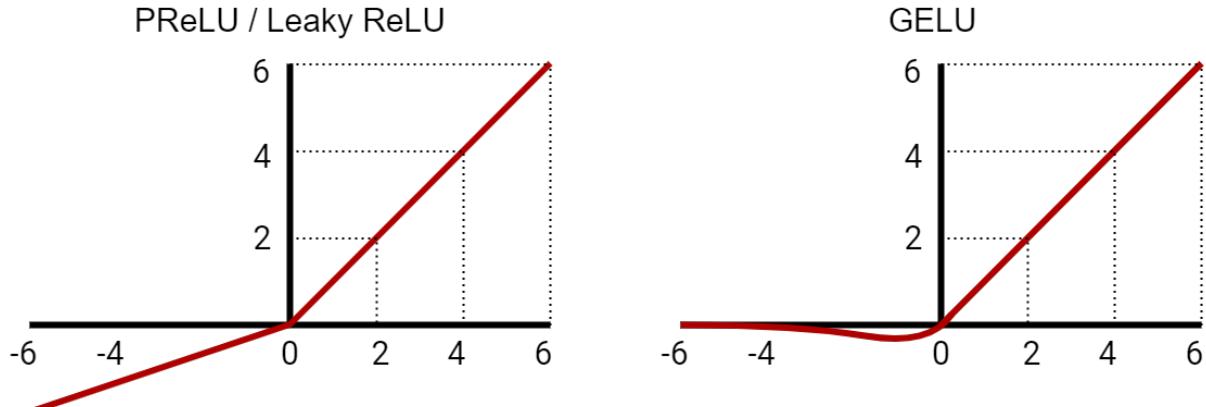


Figure 17 : Courbe de la PReLU / Leaky ReLU et de la GELU.

Description du diagramme: La PReLU et la Leaky ReLU ont la même forme selon le paramètre qu'on leur inclus en code

Exemple d'implémentation de ces fonctions d'activations en PyTorch :

```
prelu = nn.PReLU(num_parameters=1, init=0.25)
leaky_relu = nn.LeakyReLU(negative_slope=0.01)
gelu = nn.GELU()
```

3.2.6 Fonction d'activation Softmax

La fonction d'activation softmax est fondamentale en deep learning, particulièrement dans la couche de sortie des tâches de classification multiclass. Elle représente une généralisation de la fonction logistique. Le softmax « compresse » un vecteur z de dimension K , composé de valeurs réelles, en un autre vecteur de même dimension dont les valeurs, comprises entre 0 et 1, s'additionnent pour donner 1. Autrement dit, la fonction softmax permet de convertir des scores bruts en probabilités, facilitant ainsi l'interprétation des prédictions du modèle.

La formule mathématique de la fonction softmax est la suivante

$$s(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{pour } j = 1, \dots, K$$

```
def softmax(z):
    z_exp = np.exp(z)
    z_sum = np.sum(z_exp, axis=0, keepdims=True)
    return z_exp / z_sum
```

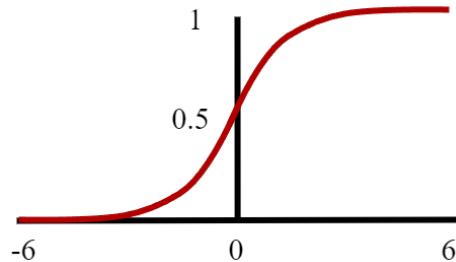


Figure 18 : Courbe de la fonction d'activation softmax.

Dans cette formule plus sophistiquée que les précédentes, z est un vecteur de scores de dimension K . Pour chaque score z_j du vecteur z (j est un itérateur allant de 1 à K), $s(z)_j$ est la probabilité correspondante obtenue par la fonction softmax. La partie supérieure de la fraction, e^{z_j} est l'exponentiel du score z_j . La partie inférieure de la fraction, $\sum_{k=1}^K e^{z_k}$, est la somme des exponentielles de tous les scores dans le vecteur z . Tous les scores sont normalisés par la somme des exponentielles pour garantir que la somme des probabilités soit égale à 1.

Imaginons que nous avons un réseau neuronal qui est utilisé pour classer une image dans l'une de ces quatre catégories : chat, chien, oiseau ou poisson. Après avoir passé l'image par le réseau neuronal

$$z = \begin{pmatrix} 2.0 \\ 1.0 \\ -1.0 \\ 3.0 \end{pmatrix} \text{ où chaque score correspond à chaque catégorie dans cet ordre}$$

La fonction softmax transforme ces scores en probabilités. Pour le premier score (pour « chat »), la probabilité correspondante serait :

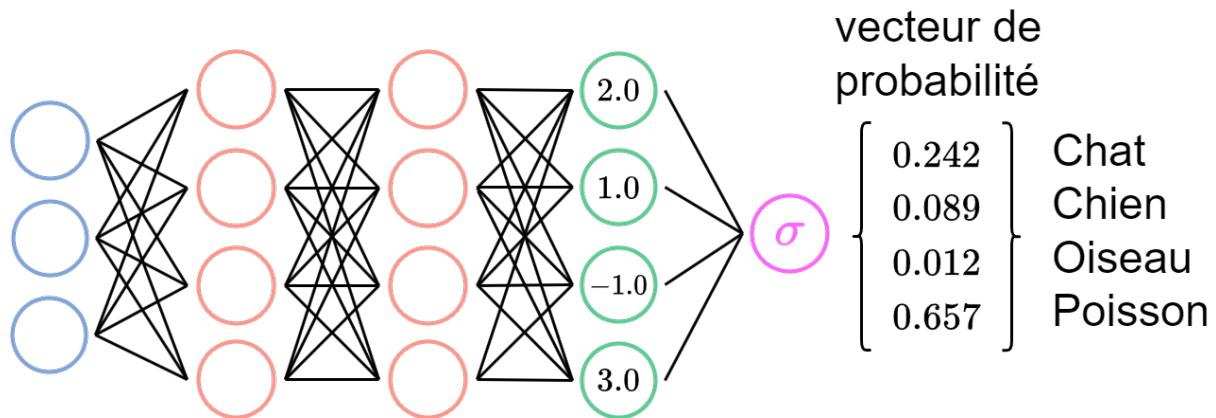


Figure 19 : Réseau de neurones qui donne sa confiance avec une softmax comme fonction d'activation.

Description du diagramme: Le réseau de neurone donne un indicateur de sa confiance envers plusieurs catégories la somme des probabilités est de 1, ici le réseau de neurone pense à 65,7% que l'image est une image de poisson.

```
import math
# Notre vecteur de scores (logits)
z = [2.0, 1.0, -1.0, 3.0]
# Calculons les exponentielles de tous les scores
z_exp = [math.exp(i) for i in z]
```

```

# Somme de toutes les exponentielles
sum_z_exp = sum(z_exp)
# Application de la fonction softmax pour obtenir les probabilités
softmax = [round(i / sum_z_exp, 3) for i in z_exp]
print(softmax)

```

L'utilisation de la fonction softmax est recommandée principalement pour les problèmes de classification multiclasse où chaque instance peut appartenir une seule classe parmi plusieurs. Elle est utile dans ces cas, car elle donne une mesure de la confiance du modèle pour chaque classe possible pour une instance donnée.

$$s(z)_j = \frac{e^{z_{\text{chat}}}}{\sum_{k=1}^K e^{z_k}}$$

Où $e^{z_{\text{chat}}}$ est l'exponentielle du score pour « chat », et $\sum_{k=1}^K e^{z_k}$ est la somme des exponentielles de tous les scores. En subsistant les valeurs concrètes dans cette formule, nous obtenons :

$$s(z)_{\text{chat}} = \frac{e^{2.0}}{e^{2.0} + e^{1.0} + e^{-1.0} + e^{3.0}}$$

$$s(z)_{\text{chat}} = \frac{7.389}{7.389 + 2.718 + 0.368 + 20.086}$$

$$s(z)_{\text{chat}} = \frac{7.389}{30.561}$$

$$s(z)_{\text{chat}} = 0.242$$

Cela signifie que, par exemple, la probabilité associée à la première valeur qui serait un chat de notre vecteur z est de 24,2%. De même, la probabilité associée à la deuxième valeur est 0.089 ou 8.9% et ainsi de suite. Toutes ces probabilités s'additionnent pour donner 1, ce qui est une propriété des probabilités.

3.2.7 Normalisation de la Sortie

Enfin, les fonctions d'activation peuvent aider à normaliser la sortie d'un neurone. Par exemple, la fonction d'activation softmax, souvent utilisée dans la couche de sortie des réseaux de neurones pour la classification multi-classes, transforme la sortie pondérée de chaque neurone en une probabilité entre 0 et 1. La somme de toutes ces probabilités est 1, ce qui permet d'interpréter chaque probabilité comme la confiance du réseau dans chaque classe possible.

3.3 Le Biais dans un Perceptron

Le biais est un terme ajouté dans une équation linéaire qui permet de décaler la sortie du modèle par une constante (une simple addition). En termes simples, le biais est intercepté dans une équation linéaire, c'est-à-dire le point où la ligne touche l'axe des ordonnées lorsque toutes les entrées sont à zéro.

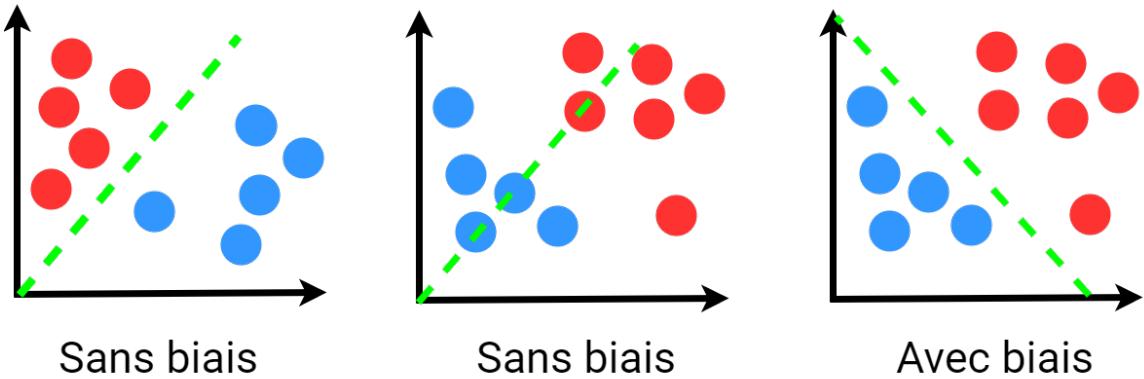


Figure 20 : Diagramme de classification d'un perceptron avec et sans biais.

Description du diagramme: Dans le premier plot sans biais, le perceptron démarre à l'origine (le point où $x = 0$ et $y = 0$) arrive à bien séparer la classe rouge et la classe bleue. Dans le deuxième, il n'y arrive plus du tout, car la distribution ne lui permet pas à partir de l'origine, sur le troisième, il y arrive grâce au biais.

Le biais permet de donner de la flexibilité à notre perceptron et donc à notre modèle pour s'adapter aux données. Il permet de décaler la fonction d'activation. C'est essentiel parce qu'il donne au perceptron la capacité de produire une variété de sorties, même pour les mêmes entrées. Il permet à la fonction d'activation, par exemple une fonction sigmoïde, de se déplacer vers la gauche ou la droite, ce qui aide le modèle à s'adapter à différentes gammes de valeurs d'entrée. Sachez-que ce paramètre est activé par défaut sur PyTorch, vous n'aurez pas à le dire explicitement de l'inclure.

Cependant, il est intéressant de noter que dans certaines rares situations, les ingénieurs d'architecture de réseaux de neurones peuvent choisir de ne pas inclure de biais dans certaines couches cachées. Généralement des architectures de type CNN, les architectures qui servent à lire des images. Cela se produit souvent dans des situations où les données d'entrée sont centrées autour de zéro, et donc le biais n'apporte pas beaucoup de bénéfice.

Jusqu'à maintenant, nous calculons la sortie d'un perceptron $w^T x$ si nous ajoutons le biais cela donne $w^T x + b$ ce qui transforme notre perceptron de fonction linéaire à une fonction affine.

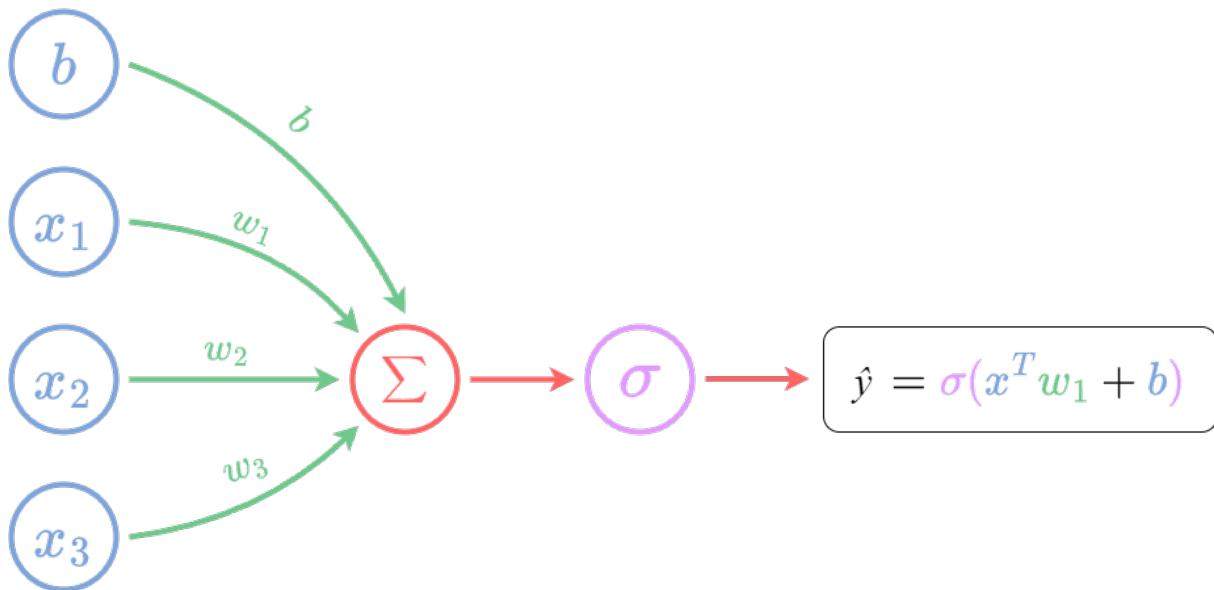


Figure 21 : Ajout d'un biais au perceptron

3.4 Le Perceptron multicouches (MLP)

Un Perceptron multicouche, ou MLP en anglais pour « Multi-Layer Perceptron », correspond à ce que j'ai désigné par « réseau de neurones » précédemment. Il est composé d'une couche d'entrée, de couches cachées et d'une couche de sortie. En somme, il s'agit d'une architecture de réseau de neurones constituée de nombreux perceptrons. Le terme « MLP » est parfois utilisé pour évoquer une ou plusieurs couches de perceptrons.

3.4.1 Propagation avant (feed-forward)

Un réseau de neurones feed-forward, est un réseau où l'information (les sorties des perceptrons) se déplace uniquement dans une seule direction, de l'entrée vers la sortie sans boucle ni cycle, il existe des architectures de réseaux de neurones où l'information peut circuler dans les deux sens comme une architecture de type RNN pour (Recurrent Neural Network), ce genre d'architecture est utilisée pour traiter des données textuelles.

Une caractéristique des réseaux de neurones feed-forward est l'absence de connexion entre les neurones au sein de la même couche. Chaque neurone d'une couche est connecté à tous les neurones de la couche suivante.

Chaque neurone reçoit des valeurs de la couche précédente et somme ces valeurs, à cela est ajouté le biais. La fonction d'activation est ensuite appliquée à cette somme pour produire la sortie du neurone. Cette sortie est transmise à tous les neurones de la couche suivante. Cela est répété pour chaque couche jusqu'à ce que la couche de sortie soit atteinte.

Nous allons analyser le fonctionnement d'un réseau feed-forward qui souhaiterait prédire la taille d'une personne à partir de deux variables, son poids et son âge, calculer à la main serait fastidieux alors faisons le en python avec numpy, pour simplifier les calculs les fonctions d'activation seront des ReLU.

```
import numpy as np
```

```

def relu(x):
    return np.maximum(0, x)

# Initialiser les poids et les biais du réseau
weights = {
    'hidden_1': np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]]),
    'hidden_2': np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]]),
    'output': np.array([[0.1], [0.2], [0.3]])
}
biases = {
    'hidden_1': np.array([0.1, 0.2, 0.3]),
    'hidden_2': np.array([0.1, 0.2, 0.3]),
    'output': np.array([0.1])
}

def feed_forward(inputs):
    # Calculer l'activation de la première couche cachée
    hidden_sum_1 = np.dot(inputs, weights['hidden_1']) + biases['hidden_1']
    hidden_activation_1 = relu(hidden_sum_1)
    print(f"Valeur de la somme pondérée dans la première couche cachée: {hidden_sum_1}")
    print(f"Valeur de l'activation dans la première couche cachée: {hidden_activation_1}")

    # Calculer l'activation de la deuxième couche cachée
    hidden_sum_2 = np.dot(hidden_activation_1, weights['hidden_2']) + biases['hidden_2']
    hidden_activation_2 = relu(hidden_sum_2)
    print(f"Valeur de la somme pondérée dans la deuxième couche cachée: {hidden_sum_2}")
    print(f"Valeur de l'activation dans la deuxième couche cachée: {hidden_activation_2}")

    # Calculer l'activation de la couche de sortie
    output_sum = np.dot(hidden_activation_2, weights['output']) + biases['output']
    output_activation = relu(output_sum)
    print(f"Valeur de la somme pondérée dans la couche de sortie: {output_sum}")
    print(f"Valeur de l'activation dans la couche de sortie: {output_activation}")

# Les données d'entrée pour l'exemple
# Poids = 70 kg, Âge = 25 ans
inputs = np.array([70, 25])
feed_forward(inputs)

# output :
>>> Valeur de la somme pondérée dans la première couche cachée: [17.1 26.7 36.3]
>>> Valeur de l'activation dans la première couche cachée: [17.1 26.7 36.3]
>>> Valeur de la somme pondérée dans la deuxième couche cachée: [37.9 46.01

```

54.12]

>>> Valeur de l`activation dans la deuxième couche cachée: [37.9 46.01 54.12]

>>> Valeur de la somme pondérée dans la couche de sortie: [29.328]

>>> Valeur de l`activation dans la couche de sortie: [29.328]

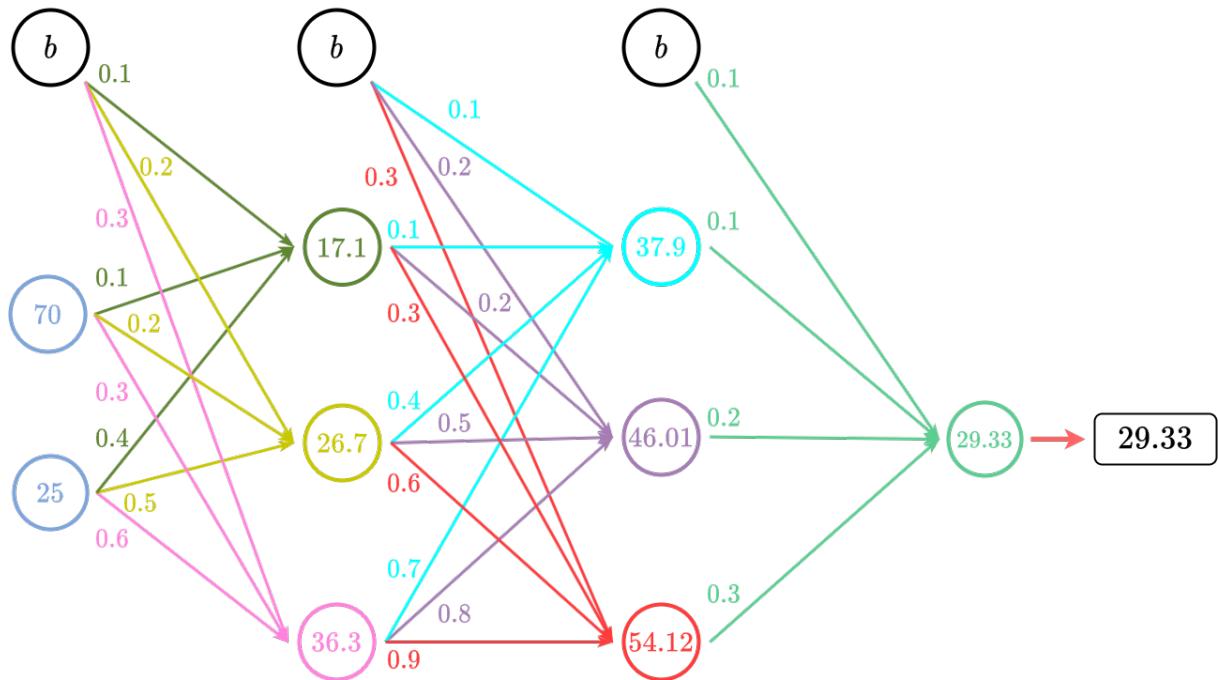


Figure 22 : Réseau à perceptron multicouche feed-forward

Ça fait beaucoup de valeur, j'ai essayé de jouer sur les couleurs pour vous aider à y voir plus clair. Je ne vais pas faire à la main tous les calculs, mais quelques un pour que vous y voyiez quand même la logique sous-jacente. Les biais sont représentés du même design que les perceptrons, mais ils n'ont pas de valeurs d'entrée, le biais n'est qu'une addition, je vous rappelle la formule mathématique du calcul d'un perceptron $w^T x + b$ où w et x sont des vecteurs, cette formule multiplie les entrées par leur poids et additionne leur biais propre à chaque perceptron. Chaque perceptron contient une fonction d'activation ReLU, si la valeur de sortie du perceptron est négative alors la valeur sera à nulle (0).

Faisons le calcul pour un perceptron de la première couche de neurone cachée:

On multiplie chaque entrée par son poids : $70 \times 0.1 + 25 \times 0.4 = 7 + 10 = 17$ Ensuite, on ajoute le biais (0.1) pour obtenir la somme pondérée : $17 + 0.1 = 17.1$

Faisons le calcul pour 37.9 de la deuxième couche cachée.

Les valeurs obtenues de la première couche cachées sont $\begin{pmatrix} 17.1 \\ 26.7 \\ 36.3 \end{pmatrix}$ les poids pour la deuxième couche sont $\begin{pmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{pmatrix}$ et les biais sont $\begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$.

On multiplie chaque entrée par son poids correspondant : $17.1 \times 0.1 + 26.7 \times 0.4 + 36.3 \times 0.7 = 1.71 + 10.68 + 25.41 = 37.8$. Ensuite, on ajoute le biais (0.1) pour obtenir la somme pondérée : $37.8 + 0.1 = 37.9$ après avoir obtenu la somme pondérée de 37.9 si la

somme pondérée obtenue est supérieure à zéro, alors la sortie est égale à la somme pondérée elle-même, sinon, la sortie est fixée à zéro. Dans notre cas, puisque la somme pondérée est égale à 37.9, qui est supérieure à zéro, la sortie de la fonction ReLU sera également égale à 37.9.

Ensuite pour obtenir la valeur de la couche de sortie 29.328 nous refaisons pareil, on utilise les valeurs des couches précédentes, $\begin{pmatrix} 37.9 \\ 46.01 \\ 54.12 \end{pmatrix}$ ainsi que les poids $\begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$ et le biais pour la couche de sortie qui sera de 0.1

On multiplie chaque entrée par son poids correspondant : $37.9 \times 0.1 + 46.01 \times 0.2 + 54.12 \times 0.3 = 3.79 + 9.202 + 16.236 = 29.228$. Ensuite on ajoute le biais (0.1) pour obtenir la somme pondérée : $29.228 + 0.1 = 29.328$ puis nous utilisons toujours la fonction d'activation ReLU, la somme pondérée est positive alors la valeur ne change pas.

Maintenant, cette sortie doit-elle signifier quelque chose ? Pas vraiment, c'était la première feed-forward pass; les poids et les biais ont été définis arbitrairement sans se baser sur les données réelles. Dans une situation réelle, ces paramètres seraient ajustés en utilisant une méthode d'apprentissage comme la descente de gradient, que nous allons voir dans le prochain chapitre.

3.5 Le théorème d'approximation universelle

Le théorème d'approximation universelle est un théorème mathématique, nous ne nous attarderons pas sur sa preuve détaillée ici. Si vous êtes curieux de la preuve mathématique, je vous suggère de consulter le livre Deep Math [2] rédigé par Arnaud Bodin et François Recher.

En termes simples, ce théorème affirme qu'un réseau de neurones feed-forward, à condition d'avoir un nombre suffisant de couches cachées et de largeur adéquate, a la capacité d'approximer n'importe quelle fonction continue sur des ensembles fermés et bornés avec une précision quasi parfaite.

Si vous voulez modéliser une fonction mathématique complexe. Il pourrait s'agir de la modélisation de la trajectoire d'une fusée, de la prédiction de la météo, de la prédiction du prix des actions sur le marché boursier, ou de tout autre problème complexe que vous pouvez imaginer. Le théorème d'approximation universelle vous assure qu'il existe un réseau de neurones capable d'apprendre cette fonction à une précision satisfaisante.

Néanmoins, bien que le théorème d'approximation universelle offre une garantie théorique de la capacité des réseaux de neurones à apprendre une variété de fonctions, il ne donne pas de garanties pratiques. Il ne spécifie pas combien de neurones pourraient être nécessaires, comment ils doivent être organisés, ou comment les paramètres du réseau (les poids) peuvent être appris. De plus, le théorème ne traite pas de l'optimisation des paramètres du réseau, qui est une question majeure dans la pratique de l'apprentissage profond.

En dépit de ces limites, le théorème d'approximation universelle reste une pierre angulaire de la théorie du deep learning. Il fournit une justification théorique pour l'usage généralisé des réseaux de neurones pour une large gamme de tâches d'approximation de fonctions, et

il donne une indication que l'augmentation de la taille ou de la profondeur d'un réseau de neurones peut permettre de mieux approximer certaines fonctions.

3.6 Résumé

Nous avons vu ce qu'était un perceptron, qu'il est doté d'un ensemble d'entrée x et que chaque entrée a une pondération par des poids spécifiques, qui déterminent l'importance relative de chaque entrée. Un biais est également ajouté pour ajuster la sortie indépendamment des entrées.

L'importance des poids est illustrée par des exemples de calculs. Une entrée avec un poids élevé influence davantage le perceptron, ce qui est crucial pour le « feature engineering » en deep learning, où il est important de sélectionner les variables les plus pertinentes pour le problème à résoudre.

Ensuite, l'importance du biais dans un perceptron est discutée. Le biais est un terme constant qui permet de décaler la sortie du modèle, ce qui donne au perceptron la flexibilité nécessaire pour s'adapter à différentes données. Il permet de déplacer la fonction d'activation vers la gauche ou la droite, aidant ainsi le modèle à s'adapter à différentes gammes de valeurs d'entrée.

Nous avons exploré l'importance des fonctions d'activation pour les perceptrons. Ces fonctions transforment la sortie pondérée de la somme des entrées d'un perceptron introduisant des non-linéarités dans le modèle, ce qui permet au réseau d'apprendre des relations complexes.

Nous avons vu 4 fonctions d'activation :

- La fonction sigmoïde, une fonction couramment utilisée qui produit une sortie entre 0 et 1. Cependant, elle peut causer des problèmes de saturation pour les valeurs d'entrée extrêmes, ralentissant l'apprentissage.
- La fonction tangente hyperbolique (tanh), qui produit une sortie entre -1 et 1. Cette fonction est également sujette au problème de vanishing gradient lors de l'apprentissage.
- La fonction d'activation de Rectification (ReLU), qui est largement utilisée en raison de sa simplicité de calcul et de sa robustesse au vanishing gradient. Néanmoins, elle a le défaut du « dying ReLU » qui désactive certains neurones.
- La fonction d'activation Softmax, qui est principalement utilisée dans la couche de sortie pour les tâches de classification multiclasse. Elle transforme les scores bruts en probabilités, facilitant l'interprétation des résultats.

Ensuite, nous avons vu le processus de propagation avant (feed-forward) dans les réseaux de neurones. Dans un réseau feed-forward, l'information (les sorties des perceptrons) se déplace uniquement dans une seule direction, de l'entrée vers la sortie, sans boucle ni cycle. Chaque neurone d'une couche est connecté à tous les neurones de la couche suivante, et il n'y a pas de connexion entre les neurones au sein de la même couche.

Le chapitre se conclut par une discussion sur le théorème d'approximation universelle, qui stipule que, théoriquement, un réseau de neurones feed-forward peut approximer n'importe quelle fonction continue à une précision arbitrairement petite, tant qu'il est suffisamment large et a suffisamment de couches cachées. Cependant, bien que ce théorème offre une garantie théorique de la capacité des réseaux de neurones, il ne donne pas de garanties pratiques.

3.7 Question

1. Qu'est-ce qu'un perceptron multicouche (MLP) ?

- a. Un type d'algorithmes de machine learning utilisé pour la classification binaire.
- b. Un type de réseau de neurones constitué d'une couche d'entrée, d'une ou plusieurs couches cachées, et d'une couche de sortie.
- c. Une fonction d'activation utilisée dans les réseaux de neurones.
- d. Un type de réseau de neurones où l'information se déplace dans les deux sens.

2. Qu'est-ce qu'un réseau de neurones feed-forward ?

- a. Un réseau de neurones où l'information peut circuler dans les deux sens.
- b. Un réseau de neurones où l'information se déplace uniquement dans une seule direction, de l'entrée à la sortie.
- c. Un type de fonction d'activation utilisée dans les réseaux de neurones.
- d. Un type de réseau de neurones utilisé pour le traitement du texte.

3. Comment sont connectés les neurones dans un réseau de neurones feed-forward ?

- a. Les neurones de la même couche sont connectés entre eux.
- b. Les neurones de chaque couche sont connectés à tous les neurones de la couche suivante.
- c. Les neurones de chaque couche sont connectés à tous les neurones de la couche précédente.
- d. Les neurones de chaque couche sont connectés à tous les neurones des autres couches.

4. Pourquoi avons-nous besoin de biais dans un réseau de neurones?

- A. Pour garantir que chaque neurone ait une sortie différente de zéro
- B. Pour ajuster la sortie du neurone indépendamment de ses entrées
- C. Pour garantir que les poids soient toujours positifs
- D. Pour augmenter la capacité du réseau à mémoriser les données d'entrée

5. Quelle est l'expression pour le calcul de la somme pondérée dans une couche cachée d'un réseau de neurones ?

- a. $\text{hidden_sum} = X \cdot W_{\text{hidden}} + b_{\text{hidden}}$
- b. $\text{hidden_sum} = W_{\text{hidden}} \cdot X + b_{\text{hidden}}$
- c. $\text{hidden_sum} = X + W_{\text{hidden}} + b_{\text{hidden}}$
- d. $\text{hidden_sum} = \frac{X}{W_{\text{hidden}}} + b_{\text{hidden}}$

6. A quoi sert une fonction d'activation ?

- a. Elle sert à introduire de la non-linéarité dans le modèle, permettant ainsi d'apprendre des relations complexes entre les entrées et les sorties.
- b. Elle sert à normaliser les entrées du modèle, en les transformant en valeurs entre 0 et 1.
- c. Elle sert à augmenter la vitesse de l'apprentissage du modèle, en accélérant la convergence de l'algorithme d'optimisation.
- d. Elle sert à réduire le nombre de paramètres du modèle, en introduisant des contraintes de parcimonie.

7. A quoi sert la non-linéarité en deep learning ?

- a. Elle permet de gérer les problèmes de vanishing et exploding gradients.

- b. Elle permet au modèle de capturer des relations complexes et non-linéaires entre les entrées et les sorties.
- c. Elle est utile pour accélérer la convergence de l'algorithme d'optimisation.
- d. Elle aide à réduire le surapprentissage en introduisant de l'irrégularité dans le modèle.

8. Quel est l'un des principaux avantages de la fonction d'activation ReLU ?

- a. Elle est complexe à calculer
- b. Elle est facile à calculer et sa dérivée est simple
- c. Elle produit une sortie constante
- d. Elle produit une sortie négative

Réponse : b

9. Quelle est la formule mathématique de la fonction d'activation ReLU ?

- a. $\sigma(x) = \min(0,x)$
- b. $\sigma(x) = \max(0,x)$
- c. $\sigma(x) = x^2$
- d. $\sigma(x) = x/2$

10. Quelle est la fonction principale de la fonction d'activation softmax ?

- a. Elle produit une sortie négative
- b. Elle produit une sortie constante
- c. Elle transforme les scores bruts en probabilités
- d. Elle produit une sortie binaire

11. Qu'est-ce que la « normalisation de sortie » d'une fonction d'activation ?

- a. Transformer la sortie d'un neurone en une probabilité entre 0 et 1
- b. Transformer la sortie d'un neurone en une valeur constante
- c. Transformer la sortie d'un neurone en une valeur négative
- d. Transformer la sortie d'un neurone en une valeur positive

12. Pourquoi la fonction d'activation Tanh est-elle parfois préférée à la fonction Sigmoid dans les couches cachées d'un réseau de neurones ?

- a. Parce que la Tanh a une sortie plus grande
- b. Parce que la Tanh est plus simple à calculer
- c. Parce que la Tanh produit des sorties centrées à zéro

13. Imaginez que vous avez un perceptron simple avec les poids [0.2, -0.5] et le biais est 0.1. Les entrées sont [2, 3]. La fonction d'activation est ReLU. Comment calculeriez-vous la sortie de ce perceptron ?

- a. $(0.2 \times 2 + -0.5 \times 3) + 0.1 = -0.9$, puis appliquez ReLU, donc la sortie est 0
- b. $(0.2 \times 2 + -0.5 \times 3) \times 0.1 = -0.09$, puis appliquez ReLU, donc la sortie est 0
- c. $(0.2 \times 2 - 0.5 \times 3) \times 0.1 = -0.09$, puis appliquez ReLU, donc la sortie est 0.09
- d. $(0.2 \times 2 + -0.5 \times 3) + 0.1 = -0.9$, puis appliquez ReLU, donc la sortie est -0.9

14. Lors l'exemple d'un réseau feed-forward de perceptron du chapitre 5.4 la tâche était de définir la taille d'une personne à partir de son poids et de son âge, ceci était une tâche de

- a. classification

- b. regression
 - c. les deux
15. **À quoi sert la fonction softmax dans la couche de sortie d'un réseau neuronal ?**
- a. Pour la classification binaire
 - b. Pour la classification multiclassée
 - c. Pour la régression
 - d. Pour l'apprentissage non supervisé

3.7.1 Correction

1. Qu'est-ce qu'un perceptron multicouche (MLP) ?

Réponse: b Un type de réseau de neurones constitué d'une couche d'entrée, d'une ou plusieurs couches cachées, et d'une couche de sortie.

2. Qu'est-ce qu'un réseau de neurones feed-forward ?

Réponse: b Un réseau de neurones où l'information se déplace uniquement dans une seule direction, de l'entrée à la sortie.

3. Comment sont connectés les neurones dans un réseau de neurones feed-forward ?

Réponse: b Un réseau de neurones où l'information se déplace uniquement dans une seule direction, de l'entrée à la sortie.

4. Pourquoi avons-nous besoin d'un biais dans un réseau de neurones?

Réponse: b Pour ajuster la sortie du neurone indépendamment de ses entrées

5. Quelle est l'expression pour le calcul de la somme pondérée dans une couche cachée d'un réseau de neurones ?

Réponse: a $\text{hidden_sum} = X \cdot W_{\text{hidden}} + b_{\text{hidden}}$

6. A quoi sert une fonction d'activation ?

Réponse: a Elle sert à introduire de la non-linéarité dans le modèle, permettant ainsi d'apprendre des relations complexes entre les entrées et les sorties.

7. A quoi sert la non-linéarité en deep learning ?

Réponse: b Elle permet au modèle de capturer des relations complexes et non-linéaires entre les entrées et les sorties

8. Quel est l'un des principaux avantages de la fonction d'activation ReLU ?

Réponse: b Elle est facile à calculer et sa dérivée est simple

9. Quelle est la formule mathématique de la fonction d'activation ReLU ?

Réponse: b $\sigma(x) = \max(0, x)$

10. Quelle est la fonction principale de la fonction d'activation softmax ?

Réponse: c Elle transforme les scores bruts en probabilités

11. Qu'est-ce que la « normalisation de sortie » d'une fonction d'activation ?

Réponse: a Transformer la sortie d'un neurone en une probabilité entre 0 et 1

12. Pourquoi la fonction d'activation Tanh est-elle parfois préférée à la fonction Sigmoid dans les couches cachées d'un réseau de neurones ?

Réponse: c Parce que la Tanh produit des sorties centrées à zéro

13. Imaginez que vous avez un perceptron simple avec les poids [0.2, -0.5] et le biais est 0.1. Les entrées sont [2, 3]. La fonction d'activation est ReLU. Comment calculeriez-vous la sortie de ce perceptron ?

Réponse: a $(0.2 \times 2 + -0.5 \times 3) + 0.1 = -0.9$, puis appliquez ReLU, donc la sortie est 0

14. Lors l'exemple d'un réseau feed-forward de perceptron du chapitre 5.4 la tâche était de définir la taille d'une personne à partir de son poids et de son âge, ceci

étaient une tâche de

Réponse: b Regression

15. **À quoi sert la fonction softmax dans la couche de sortie d'un réseau neuronal ?**

Réponse: b Pour la classification multiclassse

4 Processus d'apprentissage avec la rétropropagation (Back-propagation)

Dans le chapitre précédent, nous avons appris ce qu'est un réseau de neurones, et surtout comment il fait ses prédictions à partir de données. Cela soulève une question, comment le modèle ajuste un ensemble de paramètres pour faire des prédictions exact ? La réponse à cette question est l'objet principal de ce chapitre, l'apprentissage.

Les meilleurs algorithmes de deep learning utilisent l'apprentissage supervisé (supervised learning) comme paradigme où le modèle apprend à prédire une sortie à partir d'exemples d'entrée et de sortie labellisée. Dans ce chapitre, nous ne parlerons pas des paradigmes de l'apprentissage par renforcement (reinforcement learning) et de l'apprentissage non supervisé (unsupervised learning).

Comprendre le processus d'apprentissage d'un modèle de deep learning vous enlèvera une grosse partie « blackbox » sur le fonctionnement du deep learning. Ce chapitre a pour but de vous donner une bonne intuition de l'apprentissage et de ses problématiques. Nous allons explorer en détail les composantes clés de ce processus, notamment la fonction de perte, la descente de gradient et la backpropagation. Ces composantes permettent l'ajustement des paramètres de notre modèle et améliorent sa capacité à faire des prédictions précises à partir des données d'entraînement.

4.1 Composantes clés du processus d'apprentissage

Imaginons l'apprentissage d'un modèle de deep learning comme une traversée en voiture dans une ville inconnue à la recherche d'une destination précise – disons, un parking. Dans ce contexte, quatre éléments essentiels nous aideront à naviguer : les données, la fonction de coût, la descente de gradient et la rétropropagation.

Pour commencer, les données constituent le paysage urbain que nous explorons. Elles décrivent les routes, les immeubles, les sens de circulation, tous ces détails qui façonnent notre environnement de navigation. Les données alimentent notre véhicule, en fournissant à la fois le point de départ pour notre voyage et les repères nous permettant d'évaluer notre progression vers la destination.

C'est ici que la fonction de coût entre en jeu, agissant comme un senseur de distance qui nous indique à quel point nous sommes proches ou loin de notre parking. Chaque fois que nous faisons un mouvement – ou, dans le contexte du modèle, une prédition – la fonction de coût évalue la précision de ce mouvement en comparant notre position actuelle à la position du parking.

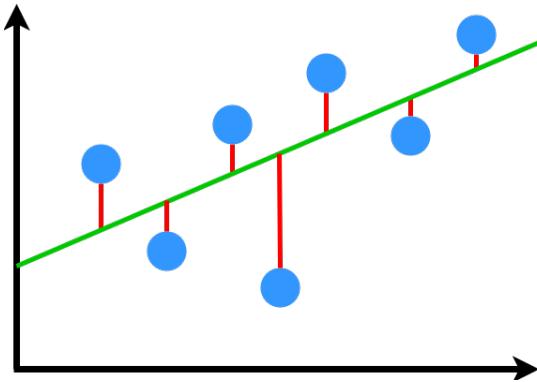


Figure 23 : « Fonction de coût mesurant la « distance » (représentée en rouge) de chaque point par rapport à la prédition du modèle. »

Figure num_figure: Ce diagramme illustre la façon dont la fonction de coût mesure la « distance » (indiquée en rouge) entre notre position actuelle et la position du parking. Plus nous sommes loin du parking, plus la « distance » est grande.

Maintenant, alors que la fonction de coût nous indique si nous nous rapprochons ou nous éloignons du parking, elle ne nous dit pas quel chemin prendre. C'est là qu'interviennent la descente de gradient et la rétropropagation, qui travaillent ensemble comme un GPS sophistiqué.

La rétropropagation est la technologie sous-jacente de notre GPS, capable de retracer nos mouvements pour comprendre quelles décisions ont influencé notre distance au parking. Une fois que la fonction de coût a évalué notre « distance », la rétropropagation analyse cette « distance » et la distribue en arrière à travers notre itinéraire, pour comprendre comment chaque décision a contribué à l'écart actuel.

La descente de gradient, quant à elle, utilise les informations de la rétropropagation pour déterminer notre prochain mouvement. Elle examine tous les chemins possibles à chaque carrefour et choisit la direction qui semble réduire le plus notre « distance » au parking.

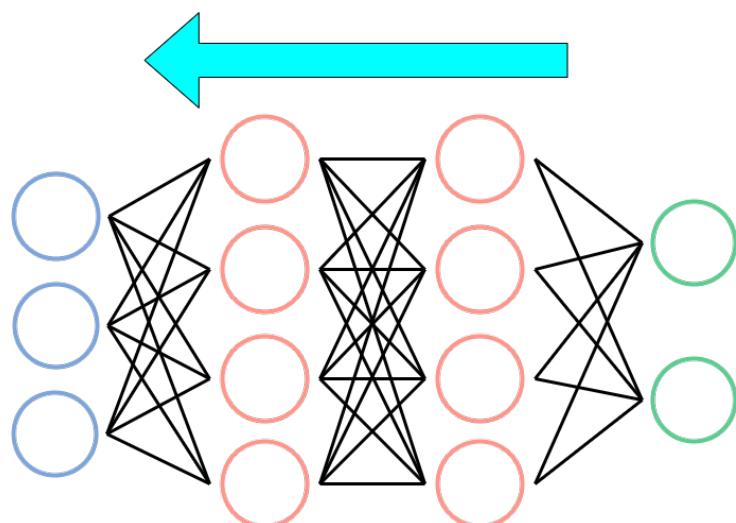


Figure 24 : « Rétropropagation. Elle retrace le chemin de notre voyage, de notre position actuelle jusqu'à notre point de départ. »

En résumé, nos quatre composantes clés sont les données qui décrivent notre environnement, la fonction de coût qui mesure notre « distance » au parking, la rétropropagation qui nous aide à comprendre nos erreurs passées, et la descente de gradient qui nous guide vers notre destination. Tous ces éléments travaillent en symbiose pour minimiser notre « distance » au parking, nous rapprochant ainsi de notre objectif. Dans les sections suivantes, nous approfondirons chacune de ces composantes et leur rôle dans l'apprentissage d'un modèle de deep learning.

4.2 La fonction de perte (loss function)

4.2.1 Introduction à la fonction de perte (loss function)

La fonction de perte (loss function), également appelée fonction d'erreur (error function), est un élément indispensable du machine learning. Vous entendrez souvent les gens utiliser ces deux noms de fonctions de manières interchangeables. Elle est essentielle pour entraîner un modèle, car elle fournit une mesure quantitative de la performance du modèle. En termes simples, elle quantifie la différence entre la prédiction d'un modèle et la valeur réelle. La fonction de perte calcule donc l'erreur pour une seule instance d'apprentissage.

La fonction de coût est la moyenne des pertes pour les instances d'apprentissage. Elle résume en quelque sorte la performance globale du modèle sur l'ensemble des données d'entraînement. Cependant, le nom de fonction coût, fonction perte et fonction d'erreur sont fréquemment utilisés de manière interchangeable, le terme de fonction objectif apparaît parfois.

Au cœur de l'apprentissage en machine, l'objectif fondamental est de minimiser la fonction de perte. Cette minimisation est réalisée en trouvant un ensemble de poids (ou paramètres) w qui réduit au maximum l'erreur calculée par la fonction de perte.

Cette fonction de perte est essentielle, car elle compare les prédictions de l'algorithme aux valeurs réelles que nous cherchons à prédire. Autrement dit, elle mesure la divergence entre la sortie du modèle et la vérité terrain.

Pour illustrer, supposons que vous entraînez un algorithme pour prédire les températures futures à partir de données météorologiques. Pour chaque jour, l'algorithme produit une prédiction de la température, et la fonction de perte compare cette prédiction à la température réelle enregistrée ce jour-là. Plus la prédiction est précise, plus la valeur de la perte est faible, et vice versa.

Ce qu'il faut comprendre, c'est que la fonction de perte n'est pas une mesure absolue de l'exactitude ou de la précision. Elle est plutôt relative : sa valeur est utilisée pour comparer différentes prédictions et différents modèles entre eux. Une valeur de perte plus faible signifie simplement qu'un modèle ou une prédiction est meilleur(e) que d'autres selon le critère spécifique défini par la fonction de perte.

Le but de l'entraînement est de trouver un ensemble de paramètres qui minimise la valeur de la fonction de coût sur l'ensemble des données d'entraînement, en espérant que cela se généralisera à d'autres données inconnues. L'importance de la fonction de perte sert à guider notre algorithme d'apprentissage. Nous ne saurions pas quels paramètres produisent de bonnes prédictions et quels paramètres produisent de mauvaises prédictions.

4.2.2 Les différents types de fonction perte

Trouver la bonne fonction de perte se choisit selon si notre problématique est une régression ou une classification, ici sera présenté deux fonctions de perte pour des problématiques de régression et deux pour de la classification. Il en existe bien d'autre, choisir la bonne fonction perte est crucial pour le bon apprentissage de votre algorithme et peut-être souvent une fonction composée de plusieurs fonctions perte avec chacune une pondération.

4.2.2.1 L'Erreur Absolue Moyenne (MAE)

L'Erreur Absolue Moyenne, Mean Absolute Error (MAE) en anglais, ou encore L1 loss, est une métrique d'erreur. Cette mesure permet de quantifier l'erreur générée par un modèle de prédiction en calculant la moyenne des valeurs absolues des différences entre les prédictions du modèle et les valeurs réelles.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Où :

- n est le nombre total d'obersations ou de points de données
- y_i est la vérité terrain pour l'observation i
- \hat{y}_i est la valeur prédite par le modèle pour l'observation i
- $|y_i - \hat{y}_i|$ est l'erreur absolue pour l'observation i

```
def MAE(y_true, y_pred):  
    return np.mean(np.abs(y_true - y_pred))
```

Chaque différence entre la valeur réelle y_i et la valeur prédite \hat{y}_i est prise en valeur absolue, ce qui garantit que toutes les erreurs sont traitées de manière égale, qu'elles soient positives ou négatives. Ensuite, nous prenons la moyenne de ces erreurs absolues.

La MAE est particulièrement utile car elle peut être interprétée directement dans les unités de la variable que vous essayez de prédire. Par exemple, si vous prédisez les températures en degrés Celsius et que votre MAE est de 2, cela signifie que vos prédictions sont en moyenne à 2 degrés de la véritable température. Exemple le modèle prédit 16 degrés quand il en fait 18 en réalité.

La MAE traite toutes les valeurs de manière égale, qu'elles soient petites ou grandes à la différence de la MSE qui est au carré qui est sensible aux erreurs extrêmes.

4.2.2.2 La Fonction de perte Quadratique Moyenne (MSE)

L'erreur Quadratique Moyenne, Mean Squared Error (MSE) en anglais, parfois appelé L2 loss), est une autre métrique d'erreur, elle est une alternative à la MAE, elle s'applique, elle aussi, à des problématiques de régression, c'est-à-dire des tâches qui ne classifient pas un chien d'un chat par exemple. Elle mesure la moyenne des carrés des erreurs, c'est-à-dire la moyenne des différences au carré entre les valeurs prédites et les valeurs réelles.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Où :

- n est le nombre total d'obersations ou de points de données
- y_i est la vérité terrain pour l'observation i
- \hat{y}_i est la valeur prédictée par le modèle pour l'observation i
- $(y_i - \hat{y}_i)^2$ est l'erreur au carré pour l'observation i

```
def MSE(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)
```

Dans cette formule, chaque erreur est élevée au carré. Cela signifie que la MSE pénalise plus lourdement les grandes erreurs que les petites. Par conséquent, un modèle avec une MSE plus faible est un modèle qui a réussi à minimiser les grandes erreurs.

La MSE est employée comme fonction de perte en régression, grâce à sa propriété de différentiabilité. Pour comprendre ce que cela signifie, il faut d'abord comprendre ce qu'est une fonction différentiable. Une fonction est dite différentiable lorsqu'elle est lisse, sans cassures ni pointes, ce qui signifie qu'à tout point de cette fonction, nous pouvons tracer une tangente². En mathématiques, cette tangente correspond au taux de variation de la fonction à ce point précis et est représentée par la dérivée de la fonction.

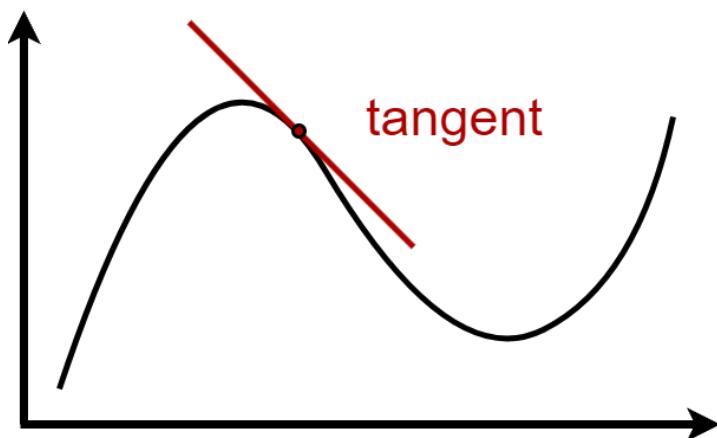


Figure 25 : fonction différentiable en tout point avec une tangent.

Cette propriété est cruciale dans le contexte du deep learning. Les algorithmes d'optimisation tels que la descente de gradient stochastique³ utilisent cette dérivée pour déterminer la direction à suivre afin de minimiser l'erreur de prédiction. Ils utilisent le principe que la dérivée est positive quand la fonction augmente, et négative quand elle diminue. En suivant cette direction, ils sont capables d'ajuster les paramètres du modèle pour réduire l'erreur, ce qui améliore la précision des prédictions.

La MSE a une particularité importante : elle donne un poids plus important aux erreurs importantes. Cela signifie que la MSE est plus sensible aux valeurs aberrantes (ou « outliers ») que d'autres métriques d'erreur comme la MAE.

Exemple : prenons le cas où nous avons un modèle de prédiction de température, qui a fait les prédictions suivantes pour cinq jours donnés :

- Prédiction : [15, 18, 20, 22, 25]

²J'ajouterais un schéma avec un exemple de fonction différentiable et un exemple de fonction non différentiable. Ex:fonction x^2 et $|x|$

³Attention !! Tu parles de la SGD trop tôt

Et voici les températures réelles qui ont été enregistrées ces jours-là :

- Vraies valeurs : [17, 20, 18, 21, 20]

Nous allons calculer la MAE et la MSE pour ces prédictions.

Calculons les erreurs absolues :

- Jour 1 : $|15 - 17| = 2$
- Jour 2 : $|18 - 20| = 2$
- Jour 3 : $|20 - 18| = 2$
- Jour 4 : $|22 - 21| = 1$
- Jour 5 : $|25 - 20| = 5$

Maintenant, prenons la moyenne de ces valeurs : $\text{MAE} = \frac{2+2+2+1+5}{5} = 2.4$

Calculons les erreurs au carré :

- Jour 1 : $(15 - 17)^2 = 4$
- Jour 2 : $(18 - 20)^2 = 4$
- Jour 3 : $(20 - 18)^2 = 4$
- Jour 4 : $(22 - 21)^2 = 1$
- Jour 5 : $(25 - 20)^2 = 25$

Maintenant, prenons la moyenne de ces valeurs $\text{MSE} = \frac{4+4+4+1+25}{5} = 7.6$

Si nous calculons la MAE et la MSE pour ces prédictions, nous constatons que :

- La MAE est de 2.4, ce qui signifie que les prédictions du modèle s'écartent en moyenne de 2.4 degrés de la réalité
- La MSE est de 7.6, une valeur supérieure à celle de la MAE, ce qui reflète la pénalisation plus sévère des grandes erreurs par la MSE par rapport à la MAE.

Ces fonctions de perte permettent d'évaluer la performance d'un modèle de prédition et de quantifier l'importance des erreurs qu'il génère, chacune avec ses spécificités et ses avantages.⁴

4.2.2.3 Choisir entre la MSE et la MAE

Le choix entre ces deux fonctions de perte est important selon ce que l'on souhaite pénaliser. La MSE donne un poids plus important aux erreurs plus grandes, en les élevant au carré, tandis que la MAE traite toutes les erreurs de manière uniforme, en prenant simplement leur valeur absolue.

Par exemple, si notre modèle prédit que le temps de trajet d'un taxi, disons 15 minutes alors que trajet réel prend 45 minutes ! Cette erreur pourrait entraîner un client très mécontent. Il est adapté d'utiliser une MSE dans ce cas pour aider l'algorithme à ne pas faire de grosse erreur, et focaliser les efforts de changement des paramètres de notre modèle pour réduire la fréquence et l'ampleur de ces grandes erreurs.

4.2.2.4 Cross-Entropy Loss

La Cross-Entropy Loss, également connue sous le nom de Log Loss, est une fonction de perte utilisée pour les problèmes de classification. Ces termes sont souvent utilisés de manière in-

⁴Remettre une couche sur le fait que les métriques ont le même minimum mais compare juste deux prédictions de manière différente

terchangeable, bien que le terme Log Loss soit généralement employé pour les problèmes de classification binaire, tandis que la Cross-Entropy Loss englobe à la fois la classification binaire et multiclasses. Dans le contexte de la Cross-Entropy Loss, on utilise les termes « Binary Cross-Entropy » pour spécifier une classification binaire et « Categorical Cross-Entropy » pour une classification multiclasse.

Originaire de la théorie de l'information, la Cross-Entropy est une mesure essentielle utilisée pour entraîner les modèles de deep learning. Elle mesure la dissimilarité entre la distribution de probabilité prédite par le modèle et la vérité terrain. La Cross-Entropy punit sévèrement les prédictions confiantes mais incorrectes. Elle est donc préférée pour les problèmes de classification où une mauvaise classification confiante peut avoir un coût élevé comme la MSE face à la MAE pour les problèmes de régression. La Cross-Entropy a deux formules mathématiques dépendant du type de classification utilisé, à savoir la classification binaire et la classification multi-classes.

4.2.2.4.1 Binary Cross-Entropy Loss

Pour la classification binaire, où nous avons deux classes possibles (0 et 1), la formule est la suivante :

$$L_{\text{BCE}} = -\frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

5

Dans cette formule, L_{BCE} est la loss de la la Binary Cross-Entropy, N est le nombre total d'observations, y_i est la « target » soit le vrai label à prédire pour chaque observation i , et \hat{y}_i est la probabilité prédite pour la classe 1 pour l'observation i ⁵. La somme est caculée sur toutes les observations.

Analysons chaque partie de la formule avant de l'utiliser :

- L_{BCE} : C'est la Binary cross entropy loss que nous essayons de calculer. C'est une mesure de la différence entre les probabilités prédites par le modèle \hat{y}_i et les étiquettes réelles y_i . Une valeur plus faible indique les prédictions du modèle sont proches des valeurs réelles.
- N est le nombre d'observations utilisé pour calculer la loss, appelé « batch size » comme les jeux de données sont trop volumineux, la loss calcule sur plusieurs fragments du jeu de données étape par étape. Dans la formule, elle est le diviseur pour permettre d'obtenir la moyenne de la loss sur tous les échantillons.
- $\sum_{i=1}^N$: c'est une somme sur tous les échantillons du « batch size » (une petite partie du jeu de données). Pour chaque échantillon i , nous calculons la contribution de cet échantillon à la loss totale et nous ajoutons toutes ces contributions pour obtenir la loss totale.
- y_i : C'est l'étiquette réelle de l'échantillon i . Pour une classification binaire, y_i est soit 0, soit 1.
- \hat{y}_i : C'est la probabilité prédite par le modèle que l'échantillon i appartienne à la classe positive (par exemple, que l'image soit celle d'un chat).

⁵Séparer la somme en deux (une somme sur les $y_{i=0}$ et une somme sur les $y_{i=1}$) Cela permet de visualiser plus facilement comment la fonction coût réagit aux erreurs de classification

⁶A intégrer que c'est une valeur comprise entre 0 et 1

- $\log(\hat{y}_i)$ et $\log(1 - \hat{y}_i)$: Le logarithme est utilisé pour amplifier l'effet des différences entre les probabilités prédictes et les étiquettes réelles. Si la probabilité prédictée est proche de l'étiquette réelle, la valeur du logarithme est proche de 0, ce qui contribue peu à la loss totale. En revanche, si la probabilité prédictée est loin de l'étiquette réelle, la valeur du logarithme est un grand nombre négatif, ce qui contribue beaucoup à la loss totale.
- $y_i \cdot \log(\hat{y}_i)$: Ce terme mesure la contribution à la loss de la prédiction pour la classe négative. Si $y_i = 0$ (c'est-à-dire, si l'échantillon i n'appartient pas à la classe positive), alors cette contribution est $\log(1 - \hat{y}_i)$, qui est faible si \hat{y}_i est proche de 0 et grand si \hat{y}_i est proche de 1. Si $y_i = 1$ (autrement dit, si l'échantillon i appartient à la classe positive), alors cette contribution est 0. Car $1 - 1 = 0$

En résumé, la formule de la cross entropy loss mesure à quel point les probabilités prédictes par le modèle sont proches des étiquettes réelles. Elle pénalise fortement les prédictions qui sont loin des étiquettes réelles, ce qui encourage le modèle à faire des prédictions précises. Le logarithme est utilisé pour amplifier l'effet des prédictions incorrectes, et les termes $y_i \cdot \log(\hat{y}_i)$ et $(1 - y_i) \cdot \log(1 - \hat{y}_i)$ permettent de calculer séparément la contribution à la loss des prédictions pour les classes positive et négative.

Maintenant, avec un exemple, prenons une situation où vous avez un modèle qui prédit si une image montre un chat (1) ou un chien (0). Supposons que nous avons trois images. Les étiquettes réelles (la réalité) et les probabilités prédictes par le modèle sont les suivantes :

Image	Étiquette réelle y_i	Probabilité prédictée d'être un chat \hat{y}_i
1	1 (c'est un chat)	0.9
2	0 (c'est un chien)	0.2
3	1 (c'est un chat)	0.6

Tableau 1 : Valeur du jeu de données avec prédiction du modèle

Reprendons notre formule de notre Binary Cross-Entropy loss plus haut :

Où :

- N est le nombre total d'échantillons (dans ce cas, $N = 3$),
- y_i est l'étiquette réelle de l'échantillon i (1 si c'est un chat, 0 si ce n'est pas un chat),
- \hat{y}_i est la probabilité prédictée que l'échantillon i soit un chat.

Ici nous avons trois échantillons, donc nous allons calculer la cross entropy loss pour chaque échantillon et ensuite prendre la moyenne :

1. Pour l'échantillon 1:
 - $y_1 = 1$ (c'est un chat)
 - $\hat{y}_1 = 0.9$ (la probabilité prédictée d'être un chat)
 - Donc, la loss pour cet échantillon est : $1 \cdot \log(0.9) + (1 - 1) \cdot \log(1 - 0.9) = -0.105$
2. Pour l'échantillon 2:
 - $y_2 = 0$ (ce n'est pas un chat, autrement dit c'est un chien)
 - $\hat{y}_2 = 0.2$ (la probabilité prédictée d'être un chat)
 - Donc, la loss pour cet échantillon est : $0 \cdot \log(0.2) + (1 - 0) \cdot \log(1 - 0.2) = -0.223$
3. Pour l'échantillon 3 :

- $y_3 = 1$ (c'est un chat)
- $\hat{y}_3 = 0.6$ (la probabilité prédictive d'être un chat)
- Donc, la loss pour cet échantillon est : $1 \cdot \log(0.6) + (1 - 1) \cdot \log(1 - 0.6) = -0.511$

Ensuite, nous additionnons ces trois valeurs et nous les divisons par 3 (le nombre total d'échantillons) pour obtenir la cross entropy loss moyenne. C'est-à-dire nous allons calculer :

$$L_{\text{BCE}} = -\frac{1}{3} \times (-0.105 + -0.233 + -0.511)$$

$$L_{\text{BCE}} = 0.280$$

```
import numpy as np
# Les étiquettes réelles
y = np.array([1, 0, 1])
# Les probabilités prédictives
y_hat = np.array([0.9, 0.2, 0.6])

# Calcul de la cross entropy loss pour chaque échantillon
loss = -1 * (y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat))

print(f"Loss pour l'échantillon 1 : {loss[0]:.3f}")
print(f"Loss pour l'échantillon 2 : {loss[1]:.3f}")
print(f"Loss pour l'échantillon 3 : {loss[2]:.3f}")

# Calcul de la cross entropy loss moyenne
mean_loss = np.mean(loss)

print(f"Loss moyenne : {mean_loss:.3f}")

# output
>>> Loss pour l'échantillon 1 : 0.105
>>> Loss pour l'échantillon 2 : 0.223
>>> Loss pour l'échantillon 3 : 0.511
>>> Loss moyenne : 0.280
```

Donc après avoir calculé la moyenne nous obtenons la binary cross entropy loss L_{BCE} pour ces données est d'environ 0.280. Plus la valeur est faible, plus le modèle est proche des valeurs réelles.

Le calcul peut être effectué en un seul calcul qui donnerait :

$$L_{\text{BCE}} = -\frac{1}{3} \times [(1 \times \log(0.9) + (1 - 1) \times \log(1 - 0.9)) + (0 \times \log(0.2) + (1 - 0) \times \log(1 - 0.2)) + (1 \times \log(0.6) + (1 - 1) \times \log(1 - 0.6))]$$

4.2.2.4.2 Categorical Cross-Entropy Loss

Pour la classification multi-classes:

$$L_{\text{CCE}} = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^C y_{ik} \times \log(\hat{y}_{ik})$$

Comme pour la Binary Cross-Entropy je vais expliquer cette formule avant de l'utiliser.

- L_{CCE} : C'est la perte de l'entropie croisée catégorielle que nous essayons de minimiser
- $-\frac{1}{N}$: Nous prenons la moyenne des pertes calculées pour chaque échantillon dans le batch. N est le nombre d'échantillons dans le batch.
- $\sum_{i=1}^N$: Nous sommes sur tous les échantillons dans le lot
- C est le nombre de classes Pour chaque classe k , nous calculons la contribution de cette classe à la loss pour chaque échantillon.
- $\sum_{k=1}^C$ Pour chaque échantillon, nous sommes sur toutes les classes.
- y_{ik} : C'est l'étiquette réelle de l'échantillon i pour la classe k . Pour une classification multi-classe, y_{ik} est 1 si l'échantillon i appartient à la classe k et 0 sinon.
- \hat{y}_{ik} : C'est la probabilité prédictive par le modèle que l'échantillon i appartienne à la classe k .
- $\log(\hat{y}_{ik})$: Le logarithme est utilisé pour amplifier l'effet des différences entre les probabilités prédictives et les étiquettes réelles. Si la probabilité prédictive est proche de l'étiquette réelle, la valeur du logarithme est proche de 0, ce qui contribue peu à la loss totale. En revanche, si la probabilité prédictive est loin de l'étiquette réelle, la valeur du logarithme est un grand nombre négatif, ce qui contribue beaucoup à la loss totale.
- $y_{ik} \times \log(\hat{y}_{ik})$: Ce terme mesure la contribution à la loss de la prédiction pour la classe k . Si $y_{ik} = 1$ (c'est-à-dire, si l'échantillon i appartient à la classe k), alors cette contribution est $\log(\hat{y}_{ik})$, qui est faible si \hat{y}_{ik} est proche de 1 et grande si \hat{y}_{ik} est proche de 0. Si $y_{ik} = 0$ (c'est-à-dire, si l'échantillon i n'appartient pas à la classe k), alors cette contribution est 0.

Voici un exemple d'utilisation de la CCE from scratch.

```
import numpy as np
# Définir les étiquettes réelles avec un one-hot-encoder
y = np.array([
    [1, 0, 0, 0], # Le premier échantillon est un chien
    [0, 1, 0, 0], # Le deuxième échantillon est un chat
    [0, 0, 0, 1] # Le troisième échantillon est un poisson
])

# Définir les probabilités prédictives par le modèle
y_hat = np.array([
    [0.7, 0.1, 0.1, 0.1], # Probabilités prédictives pour le premier échantillon
    [0.1, 0.7, 0.1, 0.1], # Probabilités prédictives pour le deuxième échantillon
    [0.1, 0.1, 0.1, 0.7] # Probabilités prédictives pour le troisième échantillon
])

# Calculer la categorical cross entropy loss avec une boucle
loss = 0.0
N = y.shape[0]
for i in range(N):
    for k in range(y.shape[1]):
        loss += y[i, k] * np.log(y_hat[i, k])

loss = -loss / N
print(f"Categorical Cross Entropy Loss: {loss:.3f}")
```

```
# output  
>>> Categorical Cross Entropy Loss : 0.357
```

Dans ce code, y contient les étiquettes réelles pour chaque échantillon et chaque classe sous forme de vecteurs one-hot. Le one-hot encoding est une représentation où chaque étiquette est un vecteur dont tous les éléments sont 0, sauf pour l'indice correspondant à la classe de l'échantillon, où la valeur est 1. Ainsi, chaque ligne de y correspond à un échantillon et est un vecteur one-hot, et chaque colonne correspond à une classe. Si un échantillon appartient à une classe, alors la valeur correspondante dans y est 1, sinon elle est 0.

y_{hat} contient les probabilités prédites par le modèle pour chaque échantillon et chaque classe. Chaque ligne de y_{hat} correspond à un échantillon, et chaque colonne correspond à une classe. Les valeurs dans y_{hat} sont des probabilités, donc elles sont comprises entre 0 et 1, et la somme des probabilités pour chaque échantillon est 1.

J'ai utilisé des boucles pour imiter une somme, puisque qu'une somme n'est en fait qu'une boucle qui parcourt chaque élément pour faire une grosse addition.

4.3 Descente de gradient

La descente de gradient est un algorithme d'optimisation utilisé pour le machine learning. Son objectif est de minimiser une fonction de coût, petit à petit, en modifiant les paramètres du modèle dans le but que notre fonction coût tende vers 0. Sans la descente de gradient, il serait difficile, voire impossible, de trouver ces valeurs de manière efficace. La descente de gradient n'est pas utilisée sous la forme que vous allez voir, mais la comprendre vous est indispensable pour comprendre la Stochastic Gradient Descent (SGD) et ses évolutions qui sont utilisées en pratique.

Prenons l'exemple où l'on se retrouve en voiture, à la recherche d'un parking dans une ville inconnue. On se retrouve à tourner en rond, à la recherche d'un emplacement pour stationner. Le défi réside dans le fait que l'on ne sait pas où se situent les parkings, ni lequel est le plus proche ou le moins cher.

C'est là que votre « GPS » entre en jeu, représentant ici la fonction de perte. Votre GPS ne peut pas vous dire directement où se trouve le meilleur parking, mais il peut vous donner une indication de la distance entre vous et le parking le plus proche. Plus vous êtes loin d'un parking, plus la valeur de cette fonction de perte est élevée.

À chaque carrefour, vous avez le choix entre plusieurs directions. Pour choisir, vous consultez votre GPS. Vous allez dans la direction qui semble réduire la distance au parking le plus proche. Parfois, vous pourriez vous tromper et vous éloigner de l'endroit où vous vouliez aller, augmentant ainsi votre « fonction de perte ». Mais grâce au feedback de votre GPS, vous pouvez corriger votre trajectoire et essayer une nouvelle direction.

Avec le temps, en utilisant cette approche de « descente de gradient », vous finirez par trouver un parking. De la même manière, dans un problème de machine learning, vous ajustez vos paramètres pas à pas pour minimiser votre fonction de perte, ce qui vous rapproche de la solution optimale.

Au vu l'importance de cet algorithme, heureusement qu'il n'est pas compliqué, pour les notions mathématiques, nous allons utiliser les dérivées et un peu d'algèbre.

4.3.1 Descente de Gradient en 1D

La descente de gradient en une dimension est la forme la plus simple de cette méthode d'optimisation. Dans ce contexte, nous disposons d'une seule variable — que nous appellerons « poids » (représenté par W pour « weights » en anglais) - à ajuster pour minimiser notre fonction de perte.

Sur le graphique suivant, l'axe des ordonnées représente le coût ou la perte. L'axe des abscisses, quant à lui, représente notre poids W . Notre objectif est d'atteindre le point minimal de la fonction de perte, représenté ici en vert. Les flèches rouges indiquent la direction de la descente de gradient — elles pointent vers l'opposé du gradient, ce qui est logique puisque nous cherchons à minimiser la fonction de perte, pas à la maximiser.

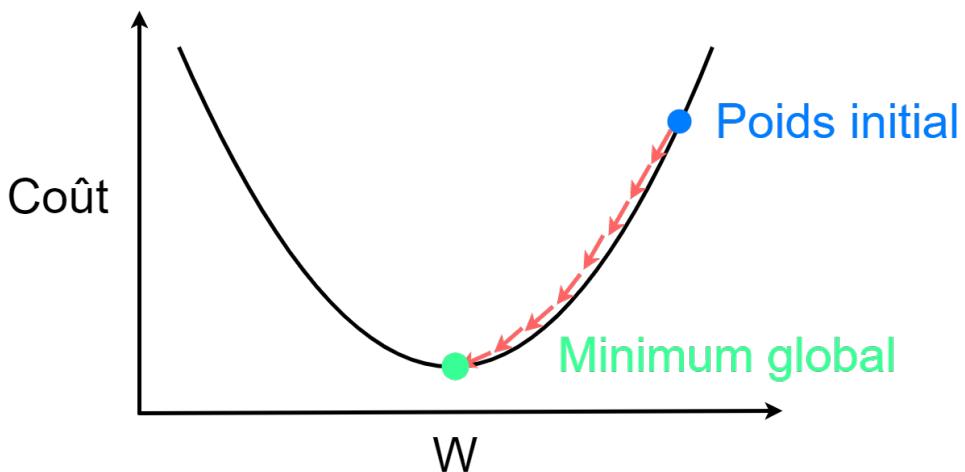


Figure 26 : « Descente de gradient: le point bleu représente le poids initial (choisi aléatoirement). Les flèches rouges indiquent les étapes de descente de gradient (aussi appelées « pas », ou « learning rate » en anglais) qui nous rapprochent progressivement du minimum global, représenté par le point vert. »

4.3.2 Impact de la Taille du Pas (Learning Rate)

Le « pas » ou « learning rate » est un « hyperparamètre », un nouveau terme important à connaître, un paramètre est appris par le modèle et est optimisé par l'algorithme, par exemple les biais et les poids s'adapte lors de l'entraînement, eux sont des paramètres, le learning rate ne sera pas optimisé par l'algorithme lui-même, mais doit être défini par le développeur, c'est un « hyperparamètre ».

Lors d'une descente de gradient, le pas (learning rate) détermine la taille de la flèche sur les diagrammes, la taille de la progression afin que notre modèle converge vers le minimum. Plus précisément, il correspond au facteur multiplicatif appliqué à la dérivée de la fonction de perte dans la formule de mise à jour des poids, il est représenté par α :

$$W := W - \alpha \cdot \nabla J(W)$$

Où :

- W représente les paramètres (weights, les poids) du modèle, elle est généralement représentée par la lettre θ (theta) par convention, mais je trouve plus approprié d'utiliser la lettre W qui est aussi parfois utilisé.

- L'opérateur `:=` est une mise à jour d'une valeur, en programmation, on utilise `=`.
- α est le taux d'apprentissage (learning rate) qui contrôle la taille des pas effectués lors de la mise à jour des paramètres, elle est présentée comme une flèche rouge dans les diagrammes.
- $\nabla J(W)$ le triangle à l'envers est « nabla » en grec mais c'est surtout le « gradient » dans le contexte du machine learning. Ici $\nabla J(W)$ est le gradient de la fonction coût J (une MSE par exemple). Le gradient représente la direction de la plus forte augmentation de la fonction coût, c'est pour cela que α est au négatif, afin de non pas de maximiser la fonction coût, mais de la minimiser, à cette étape le gradient dérivera la fonction coût, vous verrez ça en pratique plus loin.
- La fonction coût est représentée par la lettre J en référence à la matrice Jacobienne utilisée. Nous l'utiliserons plus tard pour simplifier les calculs.

```
def descente_gradient(theta, alpha, gradient):
    theta = theta - alpha * gradient
    return theta
```

Un taux d'apprentissage (learning rate) élevé entraîner peut aider à notre modèle à rapidement converger vers le minimum de la fonction de perte. Mais un taux d'apprentissage trop élevé ne convergera jamais, les mises à jour de poids deviennent si importantes que l'algorithme risque de « sauter » par-dessus le minimum recherché et peut même diverger. Sur une simple parabole cela peut aller, mais il faut imaginer que les problèmes d'optimisations sont très complexes.

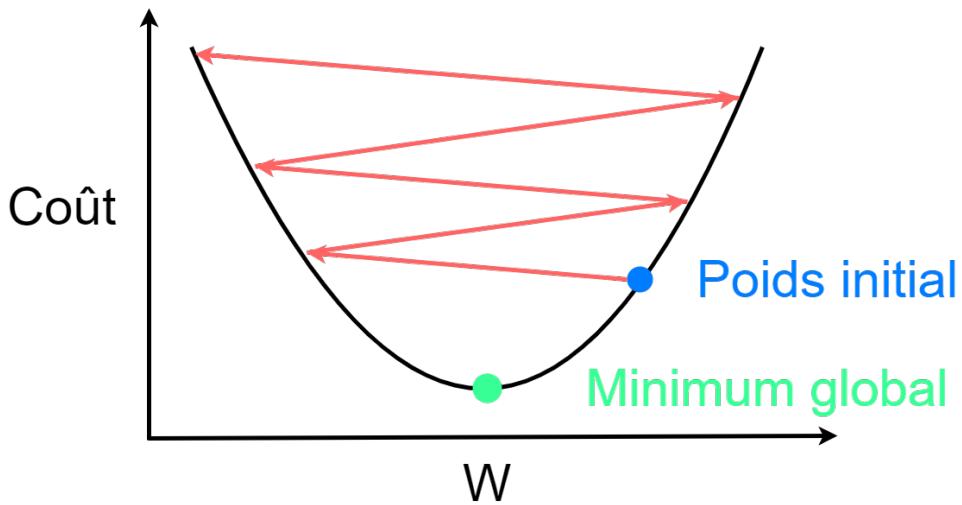


Figure 27 : « Descente de gradient avec un learning rate trop élevé: le point bleu représente le poids initial (choisi aléatoirement). Les flèches rouges indiquent les étapes de descente de gradient avec un learning rate beaucoup trop élevé, qui entraînent des oscillations importantes et empêchent totalement la convergence vers le minimum global, représenté par le point vert. »

Inversement, un taux d'apprentissage faible conduit à des mises à jour plus petites, ce qui peut assurer une convergence plus stable, mais à un rythme beaucoup plus lent. Il y a donc un risque de ne pas atteindre le minimum dans un délai raisonnable, surtout si l'on part d'un point initial éloigné du minimum.

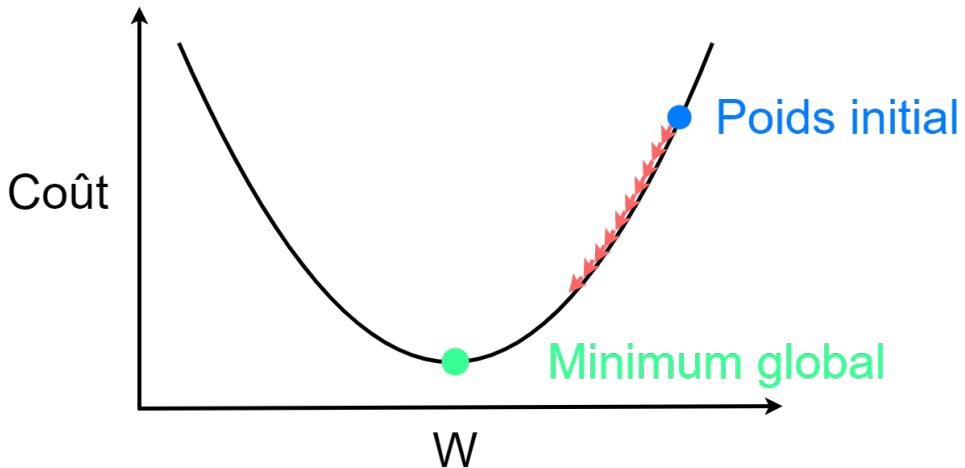


Figure 28 : Descente de gradient avec un learning rate trop faible. Les flèches rouges indiquent les étapes de descente de gradient avec un learning rate beaucoup trop faible, l'entraînement fait des pas trop faible.

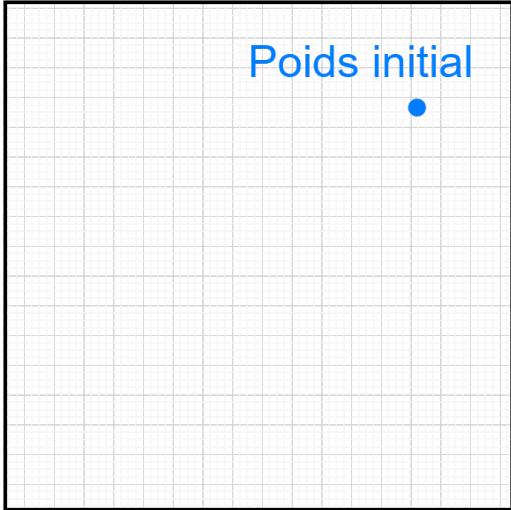
Le choix du learning rate est un exercice d'équilibrisme : il doit être assez grand pour permettre une convergence rapide (voir une convergence tout court), mais pas trop grand pour ne pas diverger et ni trop petit pour l'atteindre dans un délai raisonnable et ne pas rester coincé dans un minimum local. Dans la pratique, le taux d'apprentissage est souvent déterminé par essai et erreur, il n'y a pas de valeur universelle, ça dépend des architectures, mais ça se situe souvent entre 0.01 et 0.0001.

4.3.3 Descente de gradient en 2D

Dans le diagramme précédent, en forme de parabole d'une dimension, avait uniquement un paramètre à optimiser, nous pouvons imaginer qu'ici, il y aurait deux paramètres, le poids et son biais.

La descente de gradient est comme un explorateur perdu dans une vallée montagneuse par une nuit sans lune, il ne peut pas voir le paysage autour de lui. Tout ce qu'il sait, c'est où il se trouve (les valeurs actuelles du poids et de son biais), et il peut estimer la pente de la montagne sous ses pieds (avec le gradient son GPS).

Ce que la descente de gradient voit



La réalité

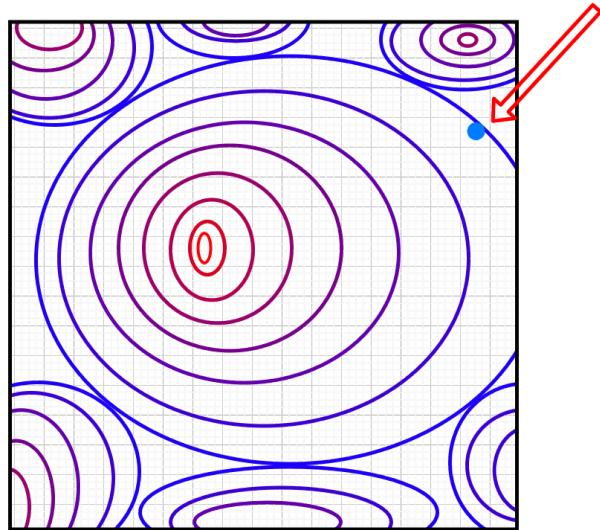


Figure 29 : Descente de gradient d'une vision en deux dimensions avec les couleurs pour jouer sur la « profondeur » comme une carte topographique. La fonction coût qui doit tendre vers le minimum global, plus la couleur tend vers le bleu, plus notre fonction coût est élevée, plus nous tendons vers le rouge, plus notre fonction coût tend à se minimiser.

Notre explorateur ne voit pas le relief complet, il est dans un brouillard complet et doit se fier uniquement au gradient (son GPS) pour décider de sa prochaine étape qui est où la pente est la plus raide. L'explorateur est totalement aveugle à la topographie générale de la vallée et doit faire confiance au gradient pour le guider.

La descente de gradient est une méthode de recherche à l'aveugle qui utilise seulement l'information locale du gradient pour naviguer dans l'espace des paramètres. C'est probablement sa grande faiblesse, car elle peut la conduire à des solutions sous-optimales ou requérir un grand nombre d'itérations pour atteindre une solution acceptable.

4.3.4 Exemple mathématique de la descente de gradient

Vous n'allez jamais implémenter mathématiquement une descente de gradient « from scratch » en Python, mais l'utilisation des mathématiques avec des exemples très simples et concrets (une addition de 1, ici) permet de démysterifier les concepts du deep learning. Que vous ayez une base en mathématiques sur les calculs différentiels ou non, les calculs sont assez simples à comprendre intuitivement si vous savez que la dérivée de x^2 est $2x$. Je ne m'attends pas à ce que vous preniez une feuille blanche et un crayon à papier pour commencer à calculer les itérations de la descente de gradient, mais que vous renforciez votre intuition de son fonctionnement. D'abord avec un seul paramètre, puis avec deux paramètres en utilisant une matrice jacobienne pour donner un exemple ayant plus d'un paramètre. Ce n'est pas grave si vous ne comprenez pas réellement les mathématiques suivantes tant que vous avez bien saisies pourquoi notre résultat finit par converger.

4.3.4.1 Avec un seul paramètre

Dans cet exemple, nous utiliserons un « toy dataset », un jeu de données simple créé uniquement à des fins d'illustration. La fonction de perte utilisée sera la MSE, Le tableau ci-dessous présente les valeurs de notre toy dataset, où x est la valeur d'entrée et y est le label, c'est-à-dire la valeur à prédire.

x	y
1	2
2	3
3	4
4	5

Tableau 2 : Valeur du jeu de notre toy dataset, x et la valeur d'entrée, y le label soit la valeur à prédire, le modèle devra tendre à additionner 1 à chaque valeur d'entrée

Pour commencer, nous initialisons notre paramètre W , à une valeur arbitraire, en pratique les poids sont initierés aléatoirement. Pour cet exemple, choisissons $W = 0$. Nous devons également choisir une valeur pour le taux d'apprentissage α , pourquoi pas $\alpha = 0.1$.

Ensuite, nous entrons dans une boucle d'itérations. Rappel de la formule :

$$W := W - \alpha \cdot \nabla J(W)$$

Première itération :

1. **Calcul des prédictions** : Avec $W = 0$, nos prédictions sont toutes 0, car $Wx = 0$ pour toutes les valeurs de x . Cela signifie que, peu importe ce que nous multiplions par 0, le résultat sera toujours 0.
2. **Calcul de l'erreur** : L'erreur pour chaque paire (x, y) est $(y - Wx)^2 = y^2$, car $Wx = 0$. Donc, pour nos données, les erreurs sont $(2^2, 3^2, 4^2, 5^2) = (4, 9, 16, 25)$.
3. **Calcul de la fonction de coût** : La fonction de coût est la moyenne des erreurs, donc $J(W) = \frac{1}{4} \times (4 + 9 + 16 + 25) = 13.5$. Ici, nous divisons par 4 pour faire la moyenne.
4. **Calcul du gradient** : Le gradient est déterminé par la dérivée de la fonction coût par rapport à notre paramètre W , ici nous dérivons la fonction MSE. Le gradient est une mesure de la pente de la fonction coût, c'est-à-dire à quel point la fonction coût change lorsque nous changeons W . La dérivée de $(y_i - Wx_i)^2$ par rapport à W est $-2x_i(y_i - Wx_i)$,
 - en dérivant, le « 2 » provient de la dérivation de la fonction carrée (à droite de la parenthèse de la MSE)
 - x_i est l'entrée correspondante à la prédition actuelle
 - $(y_i - Wx_i)$ est simplement notre erreur actuelle pour cette entrée

Donc la dérivée de $J(W)$ par rapport à W est : $\nabla J(W) = \frac{1}{n} \sum_{i=1}^n -2x_i(y_i - Wx_i)$
Pour nos données, cela donne : $\nabla J(W) = \frac{1}{4}((-2 \times 1 \times (2 - 0)) + (-2 \times 2 \times (3 - 0)) + (-2 \times 3 \times (4 - 0)) + (-2 \times 4 \times (5 - 0))) = \frac{1}{4} \times -80 = -20$.

5. **Mise à jour de W** : Nous utilisons maintenant notre formule de mise à jour pour obtenir le nouveau W :

- $W := W - \alpha \times \nabla J(W) = 0 - 0.1 \times -20 = 2.0$. C'est notre ancienne valeur de W moins notre taux d'apprentissage fois le gradient. Cela nous donne notre nouvelle valeur de W qui, espérons-le, a une fonction de coût plus faible.

```
import numpy as np

x = np.array([1, 2, 3, 4])
y = np.array([2, 3, 4, 5])

W = 0
alpha = 0.1

for i in range(6):
    predictions = W * x # notre prédiction aussi appelé y_hat parfois
    errors = (y - predictions)**2
    cost = errors.mean()
    gradient = -2 * ((y - predictions) * x).mean()
    W = W - alpha * gradient
    print(f"Iteration {i+1}, J(W) = {cost}, W = {W}")

# Output
>>> Iteration 1, J(W) = 13.5, W = 2.0
>>> Iteration 2, J(W) = 3.5, W = 1.0
>>> Iteration 3, J(W) = 1.0, W = 1.5
>>> Iteration 4, J(W) = 0.375, W = 1.25
>>> Iteration 5, J(W) = 0.2188, W = 1.375
>>> Iteration 6, J(W) = 0.1796, W = 1.3125
>>> Iteration 10, J(W) = 0.1667, W = 1.3320
>>> Iteration 100, J(W) = 0.1667, W = 1.3333
```

Nous ne calculerons pas les itérations suivantes, la logique reste la même, avec encore les mêmes étapes. Ce qui était à comprendre ici, c'est que nous utilisons l'algorithme de la descente de gradient pour avoir une fonction coût qui tend vers 0 en ajustant un paramètre. Chaque paramètre a son propre gradient, qui guide comment il doit être ajusté pour minimiser la fonction de coût. En pratique, dans un cas réel, votre ordinateur optimisera des millions de gradients, avec une descente de gradient pour optimiser des millions de paramètres afin d'ajuster simultanément tous les poids du réseau.

Nous pouvons vérifier intuitivement que notre paramètre initialement à 0 qui ne pouvait que donner de mauvais résultats puisque qu'il prédisait uniquement 0 devient 1.33 au bout de 6 itérations et que par exemple pour la quatrième valeur de x qui serait $x_4 = 4$ coupler avec notre paramètre optimisé : $1.33 \times 4 = 5.32$, notre perceptron prédit 5.32 quand il fallait prédir 5, c'est plutôt pas mal, mais vous pouvez voir tout de suite la limite d'utiliser un seul perceptron, notre modèle est trop simpliste et nécessiterait plusieurs perceptron pour être un meilleur algorithme de calculatrice.

4.3.4.2 Avec deux paramètres, le biais en plus.

Précédemment, nous avons utilisé seulement le paramètre w_1 pour simplifier les calculs et éviter d'utiliser les dérivées partielles et une matrice Jacobienne.

Cette fois, il y a deux paramètres à prendre en compte : w_1 et son biais b_1 . Cela implique que nous allons devoir utiliser les dérivées partielles et la matrice Jacobienne pour ajuster ces paramètres.

En machine learning, nos modèles ont entre des milliers ou des millions de paramètres. Pour comprendre l'effet qu'à chaque paramètre sur la fonction de coût, nous utilisons une dérivée partielle

Une dérivée partielle est la dérivée d'une fonction par rapport à l'une de ses variables, en gardant toutes les autres constantes. Par exemple, $\frac{\partial J(W)}{\partial w_1}$ mesure comment la fonction de coût $J(W)$ change lorsque nous changeons seulement w_1 , en gardant b_1 constant. De même, $\frac{\partial J(W)}{\partial b_1}$ mesure comment $J(W)$ change lorsque nous changeons uniquement b_1 , en gardant w_1 constant.

La matrice Jacobienne est un outil qui nous permet de rassembler toutes ces dérivées partielles en une seule entité. En d'autres termes, elle est une généralisation de la dérivée pour les fonctions multivariées. Chaque élément de la matrice Jacobienne est une dérivée partielle de la fonction par rapport à l'une de ses variables.

Dans notre cas, la matrice Jacobienne est définie comme suit :

$$\nabla J(W) = \left[\frac{\partial J(W)}{\partial w_1}, \frac{\partial J(W)}{\partial b_1} \right]$$

Ainsi, $\nabla J(W)$ est un vecteur dont les composantes sont les dérivées partielles de $J(W)$ par rapport à w_1 et b_1 . En utilisant ce vecteur, nous pouvons mettre à jour simultanément w_1 et b_1 de manière à minimiser la fonction de coût.

L'idée de base de la descente de gradient est de modifier les paramètres dans la direction qui réduit le plus la fonction de coût. Les dérivées partielles nous indiquent dans quelle direction la fonction de coût change le plus rapidement, c'est pourquoi nous les utilisons pour mettre à jour nos paramètres. En d'autres termes, elles nous donnent la direction de la pente la plus raide que nous pouvons descendre pour réduire la fonction de coût.

Très bien, voyons maintenant comment cette approche s'adapte lorsqu'il y a deux paramètres dans notre modèle. Pour simplifier, nous supposerons que nous avons maintenant une fonction affine de la forme $y = w_1 x + b_1$, où w_1 est le poids et b_1 est le biais.

Pour ce cas, nous initialisons nos paramètres w_1 et b_1 à des valeurs arbitraires. Prenons $w_1 = 0$ et $b_1 = 0$. Et nous prenons toujours $\alpha = 0.1$ qui sera notre taux d'apprentissage.

Le calcul de la mise à jour des poids devient maintenant :

$$[w_1, b_1] := [w_1, b_1] - \alpha \cdot \nabla J(W)$$

Où $\nabla J(W)$ est maintenant le gradient de J par rapport à W et est calculé comme suit :

$$\nabla J(W) = \left[\frac{\partial J(W)}{\partial w_1}, \frac{\partial J(W)}{\partial b_1} \right]$$

C'est la matrice Jacobienne de J par rapport à W .

Chaque $\frac{\partial J(W)}{\partial W_i}$ ou $\frac{\partial J(W)}{\partial b_i}$ est une dérivée partielle de J par rapport à W_i ou b_i qui est calculée en prenant la moyenne de $-2x_i(y_i - (w_1x_i + b_1))$ pour $\frac{\partial J(W)}{\partial w_1}$ et de $-2(y_i - (w_1x_i + b_1))$ pour $\frac{\partial J(W)}{\partial b_1}$ sur toutes les paires de données (x, y) dans notre jeu de données.

Ainsi, chaque paramètre est mis à jour individuellement en fonction de sa contribution à l'erreur totale, comme indiqué par sa dérivée partielle respective. C'est l'essence de l'utilisation des dérivées partielles et de la matrice Jacobienne dans l'algorithme de descente de gradient.

À titre d'exemple, nous calculerons la première itération de ce processus pour le jeu de données précédent.

Première itération :

1. Calcul des prédictions : Avec $w_1 = 0$ et $b_1 = 0$, nos prédictions sont toutes 0, car $w_1x + b_1 = 0$ pour toutes les valeurs de x .
2. Calcul de l'erreur : L'erreur pour chaque paire (x, y) est $(y - (w_1x + b_1))^2$, donc pour nos données, les erreurs sont les mêmes que dans le cas précédent.
3. Calcul de la fonction de coût : La fonction de coût reste la même, donc $J(W) = 13.5$.
4. Calcul du gradient : Le gradient est maintenant un vecteur de dérivées partielles :

- Pour $w_1 : \frac{\partial J(W)}{\partial w_1}$
 $= \frac{1}{4}(-2 \times 1 \times (2 - 0) + -2 \times 2 \times (3 - 0) + -2 \times 3 \times (4 - 0) + -2 \times 4 \times (5 - 0))$
 $= \frac{1}{4} \times -80 = -20.$
- Pour $b_1 : \frac{\partial J(W)}{\partial b_1}$
 $= \frac{1}{4}(-2 \times (2 - 0) + -2 \times (3 - 0) + -2 \times (4 - 0) + -2 \times (5 - 0))$
 $= \frac{1}{4} \times -28 = -7.$

Donc $\nabla J(W) = [-20, -7]$.

5. Mise à jour de W : Nous utilisons maintenant notre formule de mise à jour pour obtenir les nouvelles valeurs w_1 et b_1 :

- $w_1 := w_1 - \alpha \times \frac{\partial J(W)}{\partial w_1} = 0 - 0.1 \times -20 = 2.0$
- $b_1 := b_1 - \alpha \times \frac{\partial J(W)}{\partial b_1} = 0 - 0.1 \times -7 = 0.7$

Reste des calculs effectuer à l'ordinateur avec python.

```
import numpy as np
x = np.array([1, 2, 3, 4])
y = np.array([2, 3, 4, 5])

w1 = 0
b1 = 0
```

```

alpha = 0.1

for i in range(20):
    predictions = W1 * x + b1 # notre prédiction aussi appelé y_hat parfois
    errors = (y - predictions)**2
    cost = errors.mean()
    dW1 = -2 * ((y - predictions) * x).mean() # dérivée partielle par rapport à
    W1
    db1 = -2 * (y - predictions).mean() # dérivée partielle par rapport à b1
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    print(f"Iteration {i+1}, J(W) = {round(cost, 3)}, W1 = {round(W1, 3)}, b1 =
    {round(b1, 3)}")

# ouput
>>> Iteration 1, J(W) = 13.5, W1 = 2.0, b1 = 0.7
>>> Iteration 2, J(W) = 6.09, W1 = 0.65, b1 = 0.26
>>> Iteration 3, J(W) = 2.761, W1 = 1.545, b1 = 0.583
>>> Iteration 4, J(W) = 1.265, W1 = 0.936, b1 = 0.394
>>> Iteration 5, J(W) = 0.592, W1 = 1.335, b1 = 0.547
>>> Iteration 6, J(W) = 0.288, W1 = 1.059, b1 = 0.47
>>> Iteration 10, J(W) = 0.044, W1 = 1.127, b1 = 0.556
>>> Iteration 15, J(W) = 0.025, W1 = 1.13, b1 = 0.626
>>> Iteration 20, J(W) = 0.018, W1 = 1.109, b1 = 0.678
>>> Iteration 233, J(W) = 0.0, W1 = 1.0, b1 = 1.0

```

Ainsi, même avec plusieurs paramètres, le concept reste le même : nous ajustons chaque paramètre en fonction de son influence sur l'erreur globale. C'est ce qui fait l'efficacité de la descente de gradient.

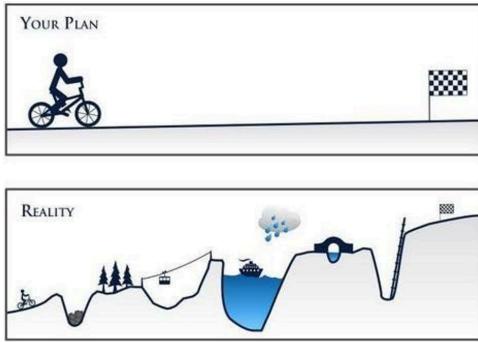
Dans ce deuxième exemple, nous avons ajouté un biais à notre modèle, ce qui le rend plus sophistiqué et plus flexible, et lui permet de s'adapter à nos données. Le biais permet au modèle de ne pas être contraint de passer par l'origine, ce qui peut améliorer la précision de nos prédictions.

Après 233 itérations, notre poids W_1 est de 1.0 et notre biais b_1 est également de 1.0. Cela signifie que notre modèle prédit $y = 1.0x + 1.0$. Par exemple, pour la quatrième valeur de x qui est $x_4 = 4$, notre modèle prédit $1.0 \times 4 + 1.0 = 5.0$, ce qui est exactement la valeur réelle de 5.

Cela montre que l'ajout d'un biais à notre modèle a permis d'améliorer considérablement la précision de nos prédictions. En fait, après suffisamment d'itérations, notre modèle est capable de prédire parfaitement les valeurs de y pour les données d'entrée données.

4.3.5 Qu'est-ce qu'un minimum local ?

Les descentes de gradient ne sont pas de simple parabole en forme de « U » vers laquelle on pourrait simplement tendre vers zéro sans rencontrer d'obstacle sur le chemin. Parmi ces obstacles, les minimums locaux sont l'un des problèmes que nous rencontrons fréquemment lors d'une descente de gradient.



En réalité, le paysage de la descente de gradient peut être parsemé de minimums locaux et globaux, d'interstices, de crêtes, et de plateaux, compliquant ainsi la convergence vers le minimum de notre fonction de coût. Tous ces problèmes viennent avec la chance que nous pouvons avoir lors de l'initialisation aléatoire des poids du modèle au démarrage de l'entraînement. Par exemple, si l'initialisation commence à gauche de la figure, nous pourrions nous retrouver dans un très mauvais minimum local. En revanche, si le poids d'initialisation est au centre, nous convergerons plus facilement vers le minimum global. Enfin, si le poids commence à droite, nous nous retrouverons également dans un minimum local, mais il s'agit d'un minimum local relativement acceptable.

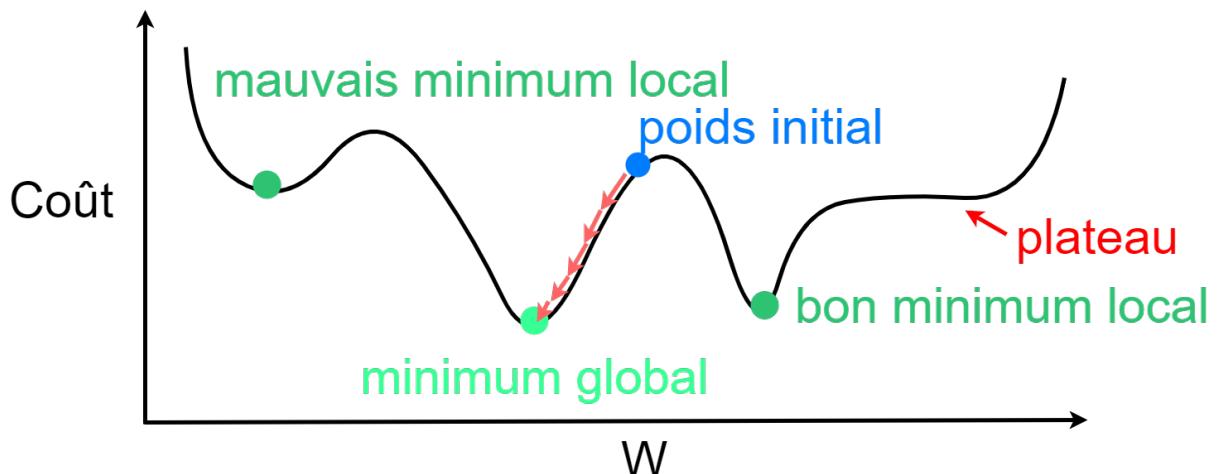


Figure 31 : « Descente de gradient : le point bleu représente le poids initial (choisi aléatoirement). Les flèches rouges indiquent les étapes de descente de gradient (aussi appelées ‹ pas ›, ou ‹ learning rate › en anglais) qui nous rapprochent progressivement du minimum global, représenté par le point vert. La flèche rouge nous montre un plateau problématique pour la convergence, c'est le phénomène du vanishing gradient que nous verrons vers la fin du chapitre. »

Ces minimums locaux nous apprennent que notre algorithme de la descente de gradient ne nous promet pas de nous donner le paramètre optimal par rapport à la fonction coût s'il tombe dans un minimum local. La nuance est que tous les minimums locaux ne sont pas mauvais. Certains peuvent être relativement acceptables, comme le minimum de droite dans la figure précédente.

Lorsque nous effectuons une descente de gradient, nous n'avons pas une vue d'ensemble de la fonction de coût. Nous ne voyons que la pente locale à l'endroit où notre paramètre se trouve.

Avec un algorithme de descente de gradient un peu plus sophistiqué, on pourrait imaginer qui aurait une « mémoire » des pas précédents et éviter de retomber dans le même minimum local.

Il existe aussi des méthodes d'initialisation des poids plus intelligente qu'une simple initialisation purement aléatoire pour éviter de commencer l'entraînement du modèle dans une région ayant plein de mauvais minimums locaux. Mais même avec les meilleures techniques d'initialisation de poids et d'algorithme de la descente de gradient les plus avancées, nous n'avons quand même pas la garantie de trouver le minimum global et il est même impossible d'avoir la certitude que notre algorithme n'est pas tombé dans un minimum local plutôt que global. Dans des cas d'application réelle (plus compliqué que la simple descente de gradient 1D présenté plus haut), le paysage d'une fonction coût multidimensionnelle, l'espace des possibilités que notre paramètre puisse prendre est gigantesque et souvent irrégulier. Le sujet de l'optimisation dans le deep learning est tout sauf un sujet trivial.

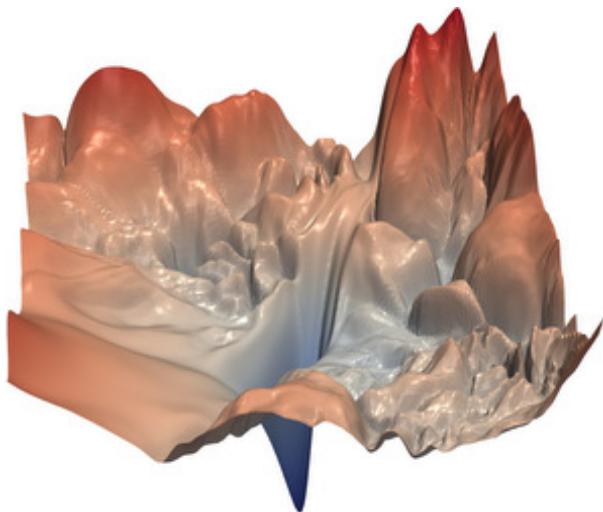


Figure 32 : « Descente de gradient sophistiqué d'un réseau »

Cette visualisation a été rendue possible via plusieurs méthodes pour rendre une descente de gradient multidimensionnelle à une visualisation en 3D [3].

4.3.6 Les points de selle (saddle points)

Le point de selle, ou « saddle point » en anglais, tire son nom de sa ressemblance avec une selle de cheval. Ce terme provient de la géométrie et se réfère à un point où la courbure de la surface change de signe. Visualisez une selle de cheval. Vous pouvez y observer une courbure ascendante dans une direction (comme le dos d'un cheval) et une courbure descendante dans l'autre (comme les côtés de la selle sur lesquels les jambes du cavalier reposent).

En termes mathématiques, dans un espace à deux dimensions, un point de selle est l'endroit où la courbe est à la fois concave et convexe. Ce phénomène se traduit par la formation d'un creux dans une direction et d'un pic dans l'autre. Dans le contexte de la descente de gradient, c'est un endroit où le gradient de la fonction coût est nul. C'est là que réside la distinction cruciale entre un point de selle et un minimum local. Un minimum local représente un creux dans toutes les directions, contrairement à un point de selle.

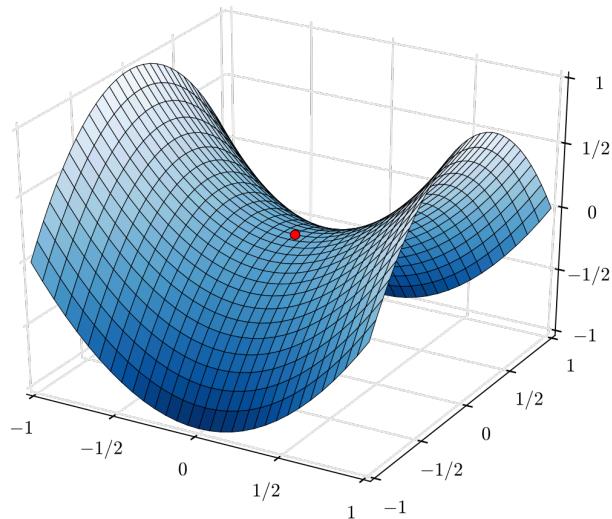


Figure 33 : « Une représentation d'un point de selle en rouge dans une fonction à deux variables. On observe une courbe ascendante dans une direction et une courbe descendante dans l'autre, formant un point de selle. »

Le problème de la descente de gradient... c'est qu'elle se repose sur le gradient pour déterminer la direction à prendre pour continuer à minimiser la fonction. Si le gradient est nul, le gradient « ne sait pas » dans quelle direction aller. Le point de selle est à la frontière entre une zone où la fonction est en train de diminuer et une zone où elle est en train d'augmenter.

Si visuellement, il est facile pour nous de voir que certaines directions sont ascendantes et d'autres descendantes, mais l'algorithme n'a pas cette vue d'ensemble. Comme dans la métaphore de la voiture au début du chapitre, le gradient est le « GPS » il est limité par l'information que le gradient lui fournit. Sur un point de selle la surface est mathématiquement plate, l'algorithme pas n'a aucune raison de bouger.

Contrairement à ce que notre intuition peut nous laisser penser, il est beaucoup plus probable de tomber dans une point de selle que de tomber dans un minimum local. Nos sens nous trompent du que nous n'avons pas une bonne intuition sur des espaces de milliers de dimensions, où il y a beaucoup plus de direction dans lesquelles la courbure peut changer.

Ce problème trompe les algorithmes d'optimisations classiques en laissant penser qu'ils ont trouvé le minimum. Comme son gradient est nul, l'algorithme s'arrête prématurément et créera un modèle sous-optimal qui n'aura pas atteint le plein potentiel de l'architecture du modèle de deep learning

Ces questions d'optimisation sont des sujets de recherche encore actifs, optimiser la vitesse d'apprentissage ce sont des entraînements plus court et plus économique. Pour résoudre ce problème les ingénieurs d'architecture n'utilise pas de descente de gradient « vanillia » mais des versions sophistiquées comme la descente de gradient stochastique que nous verrons en fin de ce chapitre

Une des choses à retenir de ce sous-chapitre assez théorique et que nos intuitions et compréhensions sur de faible dimension ne se généraliseront pas forcément dans cas de plusieurs milliers de dimensions.

4.3.7 La descente de gradient stochastique, l'apprentissage par lot

La descente de gradient classique n'est pas utilisée en pratique, des variantes comme la descente de gradient stochastique (SGD) et ses variantes résolvent plusieurs problèmes d'optimisation cités précédemment.

L'idée de la SGD est la même que la descente de gradient classique, nous cherchons à minimiser une fonction coût. La différence est que la SGD calcule différemment les mises à jour à effectuer.

Auparavant, nous avons calculé le gradient sur l'ensemble du jeu de données (batch) qui était de 4 échantillons avant de faire chaque mise à jour des poids. Ce processus n'est pas vraiment réalisable avec de gros jeu de données en plus d'être très coûteux en puissance de calculs. La SGD met à jour les poids pour chaque mini-lot (mini-batch) du jeu de données, une petite partie portion du jeu de données.

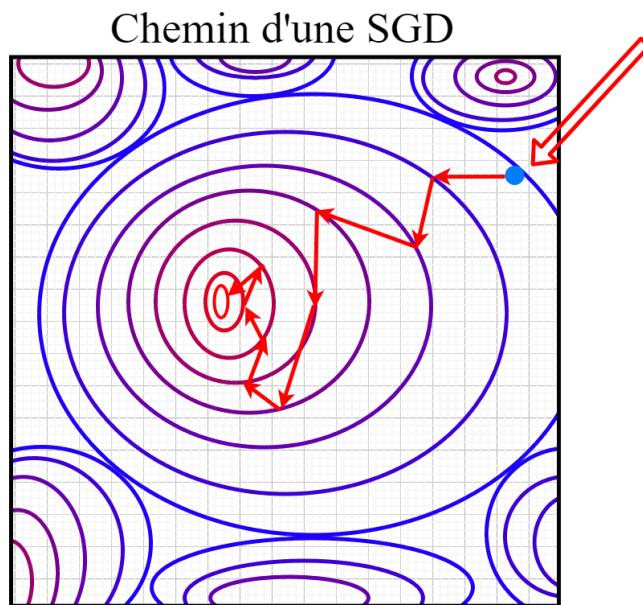


Figure 34 : « Visualisation de la Descente de Gradient Stochastique : La mise à jour des poids se fait après chaque mini-lot (mini-batch) plutôt qu'après avoir traversé l'ensemble de l'ensemble d'apprentissage. »

L'utilisation de mini-lot (mini-batch) permet de mettre plus régulièrement à jour les poids, l'algorithme peut converger plus rapidement, il n'a pas à calculer le gradient pour tout le jeu de données en entier, mais uniquement son mini-lot. En pratique, nous utilisons de grand ensemble de données qui rend l'utilisation d'une descente de gradient classique ni réellement réalisable ni optimisé, il faudrait des cartes graphiques ayant de gigantesque mémoire vive.

L'approche stochastique de la mise à jour des poids introduit cependant une certaine quantité de bruit dans le processus d'optimisation. Alors que dans la descente de gradient traditionnelle, chaque mise à jour est calculée en utilisant la totalité des données, garantissant ainsi que chaque mise à jour va dans la direction optimale, la SGD, en utilisant un échantillon ou un petit lot à la fois, peut effectuer des mises à jour qui ne vont pas exactement dans la direction optimale (vous pouvez le voir sur le diagramme d'ailleurs, il va même en arrière). Cela

peut être une bonne chose, car cela peut aider l'algorithme à éviter de rester coincé dans les minimums locaux, mais cela peut aussi rendre le processus de convergence plus chaotique.

La taille de ces mini-lots (batch size) deviennent alors un nouveau hyperparamètre à ajuster au développeur comme l'est le learning rate. Si nous avons un batch size = 1, la SGD effectuera une mise à jour de ses paramètres après chaque échantillon qui serait la SGD « pure ». Si nous utilisons un batch size de la taille du jeu de données, on revient à une descente de gradient classique. Comme le taux d'apprentissage, le choix de la taille des batchs (mini-lot) est un exercice d'équilibrisme entre la vitesse convergence et la stabilité de l'algorithme. Il est bon de savoir que de trop petits batchs size augmentent la probabilité d'être coincé dans un minimum local. En pratique, nous essayons d'avoir un batch size le plus gros possible selon notre matérielle informatique (la mémoire vive de la carte graphique) afin de ne pas avoir à effectuer la rétropropagation trop régulièrement qui demande beaucoup de ressource computationnelle.

4.4 La rétropropagation (Backpropagation)

La rétropropagation (< backpropagation > ou simplement < backprop >) est au cœur de l'apprentissage du deep learning, c'est la descente de gradient appliquée aux réseaux de neurones. Sans la backpropagation, nous ne pourrions pas apprendre efficacement les poids dans un réseau de neurones. Il deviendrait vite extrêmement coûteux en puissance de calculs d'entraîner un modèle de deep learning à mesure que la profondeur du réseau augmente.

Il s'agit là du chapitre le plus complexe de ce livre. Vous n'aurez probablement jamais de votre vie besoin d'implémenter vous-même une backpropagation, car tous les frameworks modernes de deep learning possède une implémentation optimisée de la backpropagation. Néanmoins, il est tout de même important de comprendre cet algorithme auquel le deep learning en dépendant pour son apprentissage.

La backpropagation a été introduit par des pionniers du domaine, David E. Rumelhart, Geoffrey E. Hinton et Ronald J. Williams en 1986[4]. Elle permet de quantifier l'erreur de chaque neurone par rapport à la sortie attendue, et de répartir ensuite cette erreur à travers le réseau pour mettre à jour les poids de chaque couche caché les unes après les autres.

Geoff Hinton after writing the paper on backprop in 1986



Figure 35 : « G. Hinton après avoir écrit l'article sur la rétropropagation en 1986 au début du deuxième hiver de l'IA. « *Je suppose que vous n'êtes pas encore prêts pour cela, mais vos enfants vont adorer.* » Meme tiré du film *Retour vers le futur*. »

4.4.1 La backpropagation dans le processus d'apprentissage

La première étape dans le processus d'apprentissage du modèle est la propagation en avant (forward propagation). Durant cette phase, le modèle fait une prédition basée sur les données d'entrée (forward pass), en traitant l'information à travers les différentes couches de neurones, de la couche d'entrée à la couche de sortie. C'est lors de cette phase que le réseau quantifie l'erreur qu'il fait par rapport à la valeur cible, noté \hat{y} (y chapeau) cette erreur et calculée selon la fonction coût choisie par le développeur (MSE, BCE (Binary Cross Entropy) etc).

La deuxième étape est la rétropropagation (backpropagation). Avec l'erreur calculée lors de la forward pass, la backpropagation va prendre le chemin inverse de la propagation en avant en remontant l'erreur de la couche de sortie vers la couche d'entrée.

La backpropagation calcule le gradient de la fonction de coût par rapport à chaque paramètre du réseau, c'est-à-dire chaque poids et biais. Je rappelle que le gradient est votre GPS qui indique à l'algorithme dans quelle direction converger pour minimiser l'erreur. Si le gradient d'un poids est grand en valeur absolue, cela signifie qu'une petite modification de ce poids pourrait entraîner une grande réduction de l'erreur.

L'utilisation des gradients permet d'ajuster chaque paramètre du modèle pour réduire l'erreur avec la descente de gradient, la pondération de l'ajustement des paramètres se fait selon le taux d'apprentissage (learning rate). En répétant plusieurs fois ce processus de prédition, calculer l'erreur, ajuster les paramètres du réseau et l'on répète jusqu'à que le modèle soit suffisamment entraîné.

1. Fordward pass

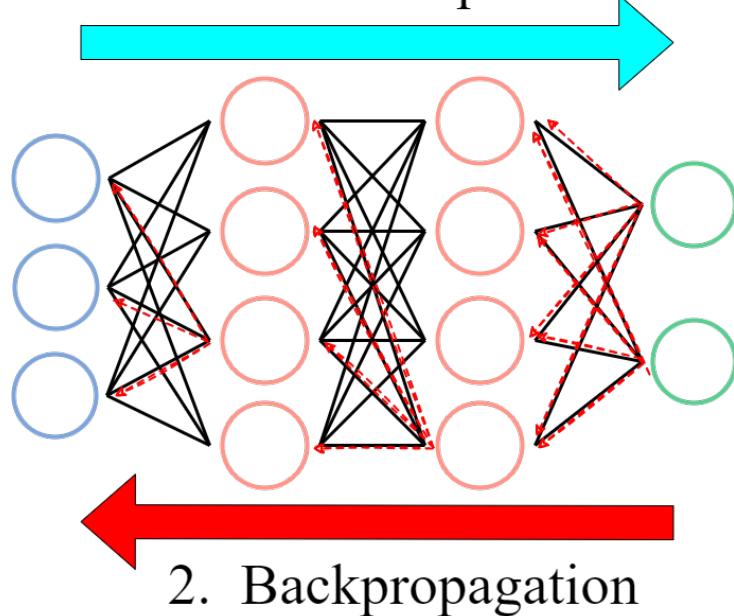


Figure 36 : « 1. La forward pass, la première étape, effectue des prédictions sur un lot de données (la taille du lot, ou ‹ batch size ›, est choisie par le développeur). 2. Avec l'erreur obtenue, la backpropagation calcule le gradient pour chaque paramètre (illustré en rouge) et effectue un ajustement proportionnel à ce gradient. Cet ajustement est fait dans le sens qui minimise la fonction de coût. »

4.4.2 Les Principes Fondamentaux de la Backpropagation

La backpropagation utilise les calculs différentiels, notamment les dérivées partielles et la règle de la chaîne. Nous avons déjà utilisé les dérivées partielles quand nous avions effectué une descente de gradient ayant 2 paramètres.

4.4.2.1 Le Rôle des Dérivées Partielles

Les dérivées partielles permettent de quantifier la manière dont un changement infime d'un poids ou d'un biais affecte la fonction de coût. Elles donnent une mesure de l'importance de chaque poids et biais dans la détermination de la sortie du réseau.

Plus la valeur absolue d'une dérivée partielle est grande, plus cela signifie qu'un petit changement du poids ou du biais (paramètre) correspondant entraînera une grande modification de la fonction de coût. À l'inverse, plus la dérivée partielle est proche de zéro, plus la fonction de coût sera insensible à des modifications de ce paramètre.

4.4.2.2 La Dérivée en Chaîne dans la Backpropagation

La règle de la chaîne est utilisée pour calculer la backpropagation, elle permet de calculer la dérivée d'une fonction composée. La backpropagation propage les erreurs de la couche de sortie vers la couche d'entrée.

Supposons que nous avons une fonction coût J (« J », comme « jacobien » de matrice jacobien utilisé lors de la descente de gradient), qui est une fonction de sortie d'un neurone ayant une fonction d'activation a (« a » comme « activation ») dans le réseau, c'est-à-dire $J = J(a)$ où a est lui-même une fonction des entrées pondérées à ce neurone, $a = \sigma(z)$, et z est une

fonction des poids (weight) w et des biais b , c'est-à-dire $z = w \cdot x + b$ la sortie du perceptron avant la fonction d'activation. Ici σ est la fonction d'activation du neurone.

Notre objectif est de comprendre comment un petit changement des poids w ou des biais b affectent la fonction coût J . Cela revient à calculer les dérivées partielles $\frac{\partial J}{\partial w}$ et $\frac{\partial J}{\partial b}$. Pour calculer cela, on applique une dérivée en chaîne :

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

De la même manière pour $\frac{\partial J}{\partial b}$ si nous cherchons à calculer l'effet du biais b à la place d'un poids w .

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b}$$

Dans ces deux formules permettent de calculer le gradient de J par rapport à tous les poids et biais du réseau, en une seule fois à travers le réseau.

4.4.3 L'apprentissage d'un réseau avec la Backpropagation : pas à pas

L'apprentissage du réseau est un processus en quatre étapes : feedforward, calcul de l'erreur, calcul du gradient et mise à jour des poids.

Dans cet exemple, nous allons examiner un réseau de neurones très simple avec une seule entrée, deux couches cachées contenant chacune un seul perceptron, et une seule sortie.

Pour comprendre comment fonctionne la backpropagation, nous allons utiliser l'exemple d'un réseau ayant un seul perceptron avec son biais par couche. Ce réseau aura une couche d'entrée, deux couches cachées et une couche de sortie.

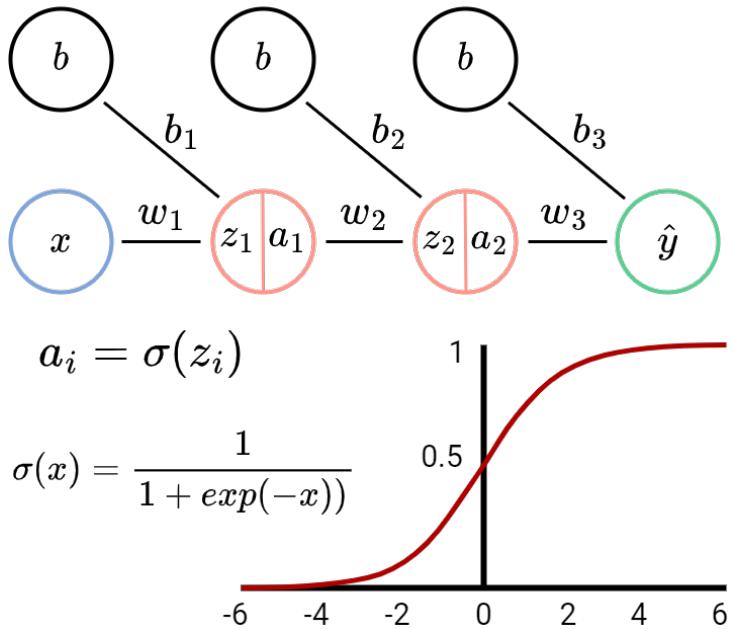


Figure 37 : Notre petit réseau de neurones. Il se compose d'une entrée, de deux couches cachées (avec un perceptron et son biais dans chacune), et d'une couche de sortie. Le terme z_i représente la somme pondérée des entrées d'un perceptron avant l'application de la fonction d'activation σ , qui est une fonction sigmoïde dans notre cas.

4.4.3.1 Feedforward: la première étape

La première étape est quand un réseau de neurones fait une prédiction (forward pass). Dans cette phase, le réseau de neurones fait des prédictions en utilisant ses poids et biais actuels.

Prenons un réseau de neurones avec deux couches cachées, chacune ayant un seul perceptron, et une couche de sortie avec une sortie. Chaque perceptron utilise la fonction d'activation sigmoïde, et l'erreur est calculée à l'aide de l'erreur quadratique moyenne (MSE).

Pour simplifier, disons que notre réseau ne contient qu'un seul échantillon d'entrée, x , et une sortie attendue, y .

Tout d'abord, l'entrée x est multipliée par le poids w du premier perceptron, puis un biais b est ajouté. C'est ce qu'on appelle une combinaison linéaire.

$$z_1 = w_1 x + b_1$$

Ce résultat z_1 est ensuite passé à travers une fonction d'activation (la fonction sigmoïde dans notre cas) pour donner le résultat a_1 (pour « activation »). La fonction sigmoïde est définie par :

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Le processus est ensuite répété pour la seconde couche cachée et la couche de sortie.

$$a^1 = \sigma(z_1)$$

La deuxième couche cachée, fait de même avec la sortie de la première couche cachée, a_1 , pour produire une autre sortie, a_2 .

$$a^2 = \sigma(w^2 a^1 + b^2)$$

Voilà notre prédiction est faite, la prédiction est nommée \hat{y} . Mais, comment savons-nous à quel point nous avons bien (ou mal) fait ?

4.4.3.2 Calcul de l'erreur : la deuxième étape

C'est là qu'intervient notre fonction coût pour calculer l'erreur de notre modèle, la MSE ici. Nous prenons notre prédiction \hat{y} et la soustrayons à la valeur réelle y , on met au carré le résultat et nous avons l'erreur.

$$J = \frac{1}{2}(y - \hat{y})^2$$

Ici c'est l'erreur pour une seule prédiction pour faire simple. Pour un lot de données, nous ferions cela chaque prédiction, puis nous prendrions la moyenne de toutes ces erreurs. Pour le moment, restons-en à une seule prédiction.

Le facteur $\frac{1}{2}$ est une commodité mathématique pour la dérivation, ça simplifie les calculs que vous verrez plus tard.

Maintenant que nous savons à quel point nous avons bien (ou mal?) fait, comment utilisons-nous cette information pour améliorer notre réseau de neurones ?

4.4.3.3 Calcul du gradient: la troisième étape

C'est là qu'intervient la backpropagation. Nous allons prendre notre erreur et la « propager en arrière » à travers notre réseau pour trouver à quel point chaque poids et chaque biais a contribué à cette erreur.

La backpropagation calcule le gradient de chaque paramètre, c'est-à-dire les poids et les biais, chacun a son propre gradient. Notre réseau a deux couches cachées, chaque couche ayant un seul perceptron (un seul poids et un seul biais), plus un poids et un biais pour la couche de sortie, nous avons six gradients à calculer.

Les voici :

1. Le gradient du poids w^1 de la première couche cachée
2. Le gradient du biais b^1 de la première couche cachée
3. Le gradient du poids w^2 de la deuxième couche cachée
4. Le gradient du biais b^2 de la deuxième couche cachée
5. Le gradient du poids w^3 de la couche de sortie
6. Le gradient du biais b^3 de la couche de sortie

Cependant, pour simplifier notre explication, nous allons uniquement calculer le gradient pour le poids w^2 de la deuxième couche cachée. Ces calculs peuvent être appliqués de la même manière pour les autres paramètres du réseau.

Nous allons calculer le gradient de la fonction de coût par rapport à notre paramètre w^2 (le poids de la deuxième couche cachée) affecte la fonction coût J . Le gradient donnera une indication de la façon dont nous devons ajuster w^2 pour minimiser l'erreur.

On commence par la fonction de coût MSE que l'on a défini précédemment :

$$J = \frac{1}{2}(y - \hat{y})^2$$

Pour cela, on va utiliser la règle de la chaîne qui dit que la dérivée d'une fonction composée est le produit des dérivées. Donc, on va décomposer $\frac{\partial J}{\partial w^2}$ en trois parties.

1. La dérivée de J par rapport à \hat{y} (la prédiction), que l'on notera $\frac{\partial J}{\partial \hat{y}}$
2. La dérivée de \hat{y} par rapport à z^2 (la sortie du neurone avant la couche d'activation), que l'on notera $\frac{\partial \hat{y}}{\partial z^2}$.
3. La dérivée de z^2 par rapport à w^2 (le poids), que l'on notera $\frac{\partial z^2}{\partial w^2}$.

On a donc :

$$\frac{\partial J}{\partial w^2} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^2} \cdot \frac{\partial z^2}{\partial w^2}$$

Allons-y étape par étape :

1. La dérivée de J par rapport à \hat{y} est simplement :

$$\frac{\partial J}{\partial \hat{y}} = y - \hat{y}$$

2. La dérivée de \hat{y} par rapport à z^2 est la dérivée de la fonction d'activation sigmoïde. Si on note $\sigma(z)$ la fonction sigmoïde, alors sa dérivée est $\sigma(z) \cdot (1 - \sigma(z))$. On a donc :

$$\frac{\partial \hat{y}}{\partial z^2} = \hat{y} \cdot (1 - \hat{y})$$

3. La dérivée de z^2 par rapport à w^2 est simplement la valeur de l'entrée a^1 , car $z^2 = w^2 \cdot a^1 + b^2$. Donc on a :

$$\frac{\partial z^2}{\partial w^2} = a^1$$

Si on empile tout ensemble on obtient :

$$\frac{\partial J}{\partial w^2} = (y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}) \cdot a^1$$

On a notre gradient du poids w^2 ! En utilisant cette formule, nous pouvons calculer la direction dans laquelle nous devons ajuster le poids w^2 pour minimiser la fonction de coût J .

4.4.3.4 Mise à jour des poids: la quatrième étape

Maintenant, supposons que nous avons nos six gradients, nous pouvons faire ce pour quoi nous sommes ici : mettre à jour nos poids et nos biais. Pour cela, nous prenons chaque poids

et chaque biais et le déplaçons un petit peu dans la direction opposée à son gradient. C'est la descente de gradient que nous avons déjà vu.

Pour chaque poids :

$$w := w - \alpha \frac{\partial J}{\partial w}$$

Pour chaque biais :

$$b := b - \alpha \frac{\partial J}{\partial b}$$

Où α est le taux d'apprentissage (learning rate) et $\frac{\partial J}{\partial w}$ ou $\frac{\partial J}{\partial b}$ est le gradient de la fonction coût rapport à w ou b .

Voilà, enfin, une itération complète de la backpropagation ! C'est le processus que notre réseau de neurones parcourt à chaque fois qu'il apprend à partir d'un lot (batch) de nos données, et faire toutes sortes de chose cool.

4.4.3.5 Exemple en code

Voici un exemple de code en PyTorch pour illustrer ces étapes

```
import torch
from torch import nn

# Définition de notre réseau de neurones simple
class SimpleNetwork(nn.Module):
    def __init__(self):
        super(SimpleNetwork, self).__init__()
        self.layer1 = nn.Linear(1, 1)
        self.layer2 = nn.Linear(1, 1)
        self.layer3 = nn.Linear(1, 1)

    def forward(self, x):
        x = torch.sigmoid(self.layer1(x))
        x = torch.sigmoid(self.layer2(x))
        return self.layer3(x)

# Instanciation de notre réseau
net = SimpleNetwork()

# Définition de notre fonction de coût MSE
criterion = nn.MSELoss()

# Définition de notre optimiseur (SGD)
optimizer = torch.optim.SGD(net.parameters(), lr=0.01) # lr est notre taux
d'apprentissage

# Exemple d'entrée et de sortie
x = torch.tensor([2.0])
y = torch.tensor([1.0])
```

```

# Étape 1 : Feedforward
y_pred = net(x)

# Étape 2 : Calcul de l'erreur
loss = criterion(y_pred, y)

# Étape 3 : Backpropagation
loss.backward()

# Étape 4 : Mise à jour des poids
optimizer.step()

# Remise à zéro des gradients pour la prochaine étape
optimizer.zero_grad()

```

Dans ce code, `loss.backward()` effectue la backpropagation et calcule les gradients pour tous les paramètres du réseau. L'appel à `optimizer.step()` effectue ensuite la mise à jour des poids en fonction des gradients calculés. La dernière ligne, `optimizer.zero_grad()`, remet les gradients à zéro pour préparer la prochaine itération.

4.4.4 Les problèmes du vanishing gradient et exploding gradient

Le vanishing gradient (gradient qui disparaît) et l'exploding gradient (gradient qui explose) sont des problèmes que la backpropagation rencontre avec des réseaux profonds. Ils sont liés à la manière dont les gradients de la fonction coût sont calculés et propagés dans le réseau lors de la backpropagation.

Ils se produisent lorsqu'un réseau de neurones est suffisamment profond, c'est-à-dire un réseau avec beaucoup de couche cachée, et rendent l'entraînement de tels réseaux difficile, voire impossible à entraîner.

Le problème du vanishing gradient (gradient qui disparaît) se produit lorsque les gradients des couches profondes deviennent très petits, presque zéro. Cela rend le réseau de neurones très lent à apprendre, voire incapable d'apprendre. Ces petits gradients entraînent des modifications minuscules des poids, rendant l'apprentissage extrêmement lent ou stagnant.

À l'inverse, le problème du gradient explosif se produit quand les gradients deviennent trop grands. Ce qui entraîne une mise à jour trop agressive des paramètres et provoque une instabilité et faire diverger l'apprentissage.

Mais pourquoi ces problèmes se produisent-ils ? Pour répondre à cette question, nous devons regarder de plus près le processus de backpropagation et la manière dont les gradients sont calculés.

4.4.4.1 Pourquoi ces problèmes se produisent-ils

Ces problèmes sont inhérents à la nature de la backpropagation dans les réseaux très profonds. Lorsque nous calculons le gradient de la fonction de coût par rapport aux poids et aux biais, nous utilisons la règle de la chaîne pour propager l'erreur de la couche de sortie à la couche d'entrée.

Ce produit est ensuite transmis à la couche précédente. Si la dérivée de la fonction d'activation est très petite (proche de zéro), le produit sera également très petit, ce qui peut conduire à un

vanishing gradient si ce processus est répété à travers de nombreuses couches. À l'inverse, si la dérivée de la fonction d'activation ou les poids du réseau sont trop grands, le produit sera également trop grand, ce qui peut conduire à un exploding gradient si ce processus est répété à travers de nombreuses couches.

Certaines fonctions d'activation, comme la sigmoïde et la tanh, ont tendance à saturer. C'est-à-dire que leurs sorties tendent vers leurs limites pour les grandes valeurs d'entrée. Pour la sigmoïde, ces limites sont 0 et 1, tandis que pour la tanh, elles sont -1 et 1. Cela signifie que leurs dérivées deviennent très petites, proches de zéro, pour ces grandes entrées. C'est pourquoi ces fonctions d'activation ne sont pas utilisées pour des réseaux profonds, car elles causent du vanishing gradient. À la place, la fonction d'activation ReLU est généralement préférée pour les réseaux profonds, car la ReLU ne sature pas pour les entrées positives.

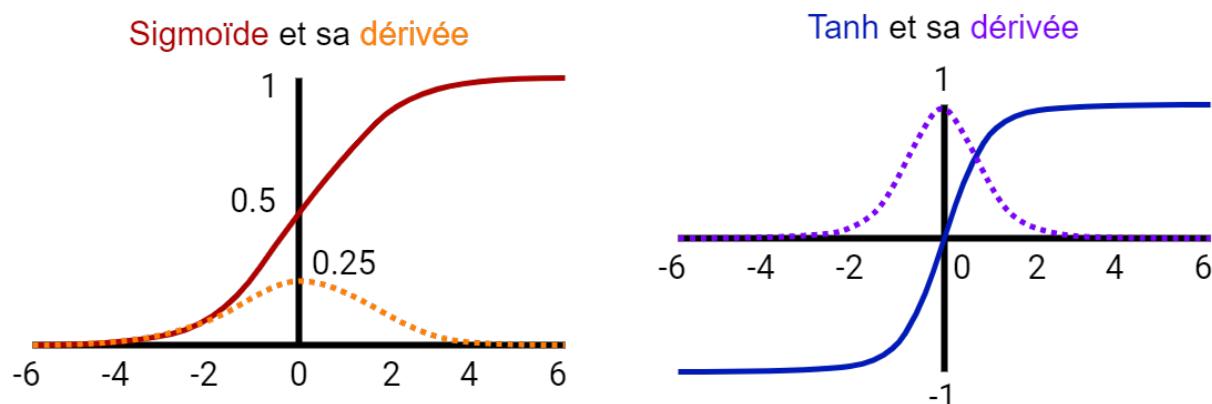


Figure 38 : Fonction d'activation sigmoïde et Tanh avec leur dérivée.

Pour être plus claire, si nous avons un réseau de neurones avec 5 couches cachées, et que le gradient de la fonction d'activation est de 0,1 pour chaque neurone, le gradient pour la première couche cachée serait de $(0,1)^5 = 0,001$. Cela rendrait la mise à jour des poids de cette couche presque insignifiante. Cela rend le réseau de neurones très lent à apprendre, voire incapable d'apprendre. Si mathématiquement le réseau finira par converger au bout d'un long moment, en informatique la précision n'étant pas infini, le modèle ne convergera jamais.

Il existe plein de petites méthodes pour éviter le vanishing gradient. La plus facile à appliquer est d'utiliser des fonctions d'activation qui ne saturent pas, comme la fonction ReLU. Néanmoins, même la ReLU n'est pas sans problème. Par exemple, si les neurones sont cloués à 0, alors les poids auront un gradient nul, ce qui entraînera le problème du « dying ReLU » (la mort du ReLU), d'où ses variantes comme la Leaky ReLU.

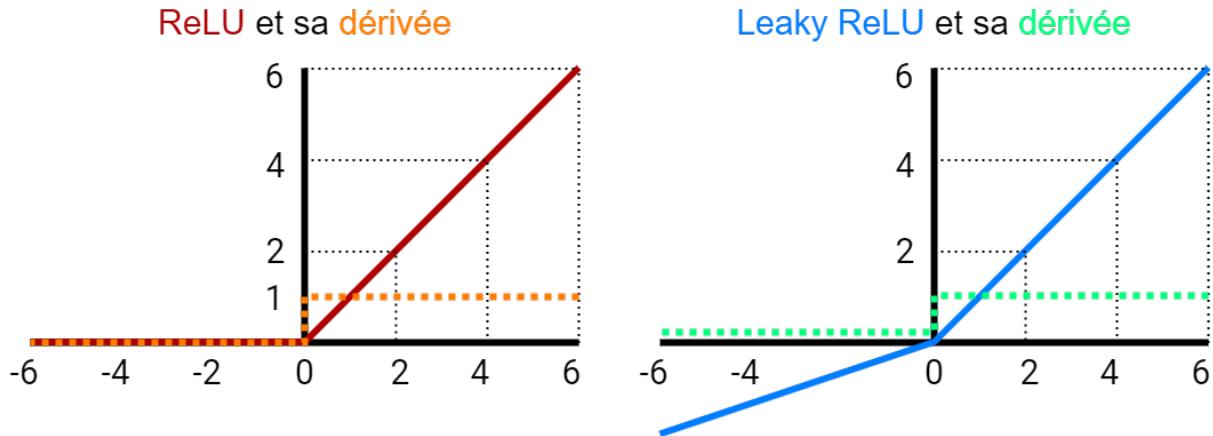


Figure 39 : Fonction d'activation ReLU et Leaky ReLU avec leur dérivée.

Lorsqu'un gradient devient trop grand, il fera une grande mise à jour des poids lors de l'étape de descente de gradient. Ce serait comme avoir un taux d'apprentissage anormalement élevé. La mise à jour du poids peut être si extrême que l'algorithme d'apprentissage ne convergerait jamais, oscillant autour du minimum global ou même divergeant complètement.

L'exploding gradient est plus souvent observé dans les tâches d'apprentissage où les sorties sont sensibles aux valeurs d'entrée plus anciennes, par exemple pour des tâches de texte (NLP).

Reprendons l'exemple précédent, mais supposons maintenant que le gradient de la fonction d'activation est de 10 pour chaque neurone. Le gradient pour la première couche cachée serait alors de $(10)^5 = 100000$, ce qui entraînerait une mise à jour massive des poids de cette couche. Cela peut provoquer une oscillation du modèle autour de la solution optimale, ou dans le pire des cas, le modèle pourrait diverger et ne jamais trouver une bonne solution.

Des poids initialement élevés dans le réseau contribuent à l'exploding gradient. Si les poids du réseau sont initialisés avec de grandes valeurs, cela entraînera des valeurs d'activation élevées. Puis, lors de la backpropagation, ces valeurs élevées sont multipliées pour donner des gradients encore plus élevés, amplifiant encore plus le problème. Pour éviter une initialisation avec des poids trop élevé, nous utilisons certaines initialisations des poids, comme l'initialisation Xavier[5].

4.4.5 Avenir de la Backpropagation dans le Deep Learning

La recherche sur la manière dont nos algorithmes apprennent est un sujet de recherche très actif, de meilleurs algorithmes d'apprentissage résoudraient d'énorme problème, comme avoir besoin de moins de données (de bonnes qualités notamment) et moins de puissance de calculs s'ils apprennent plus vite.

Les alternatives actuelles sont toujours inférieures à la backpropagation classique[6], [7], [8], [9], mais qui sait dans quelques années, peut-être que la backpropagation sera devenu ringarde.

Geoffrey Hinton, l'un des pères de la backpropagation, travaille encore activement sur de nouveaux algorithmes en quête d'une alternative plus efficace à la backpropagation[10]. G. Hinton et Y. Bengio, tous deux lauréats du prix Turing (l'équivalent du prix Nobel dans le domaine de l'informatique) pour leur travail sur le deep learning, admettent que la backpropagation n'est

probablement pas l'approche optimale pour l'apprentissage en deep learning. Ces chercheurs étudient la manière de créer des modèles de deep learning « intelligent », et il semble improbable que notre cerveau apprenne avec un algorithme similaire à la backpropagation.

De nombreux chercheurs dans le domaine pensent que la clé d'une meilleure méthode d'apprentissage pourrait résider dans la biologie et le fonctionnement de notre cerveau[11]. Ces recherches, bien que prometteuses, sont encore à leurs débuts. Il est trop tôt pour déterminer si les futures alternatives à la backpropagation seront inspirées par la biologie ou non.

4.5 Résumé

La fonction de perte, aussi appelée fonction d'erreur, est un élément central du deep learning. Elle mesure la différence entre les prédictions d'un modèle \hat{y} et les valeurs réelles y . La fonction de coût est la moyenne des pertes pour l'ensemble d'un batch, la taille de ce batch, le batch size est défini par le développeur. L'objectif est de minimiser la fonction de perte en trouvant l'ensemble de paramètres qui réduit au maximum l'erreur calculée par la fonction de perte.

Il existe différents types de fonctions de perte selon que le problème est de régression ou de classification. Pour les problèmes de régression, deux fonctions de perte communes sont l'Erreur Absolue Moyenne (MAE) et l'Erreur Quadratique Moyenne (MSE).

- La MAE est la moyenne des valeurs absolues des différences entre les prédictions du modèle et les valeurs réelles. Elle est utile car elle peut être interprétée directement dans les unités de la variable que vous essayez de prédire. Par exemple, si vous prédisez les températures en degrés Celsius et que votre MAE est de 2, cela signifie que vos prédictions sont en moyenne à 2 degrés de la véritable température.
- La MSE est une alternative à la MAE, elle mesure la moyenne des carrés des erreurs, c'est-à-dire la moyenne des différences au carré entre les valeurs prédites et les valeurs réelles. La MSE pénalise plus lourdement les grandes erreurs que les petites, ce qui la rend plus sensible aux valeurs aberrantes que d'autres métriques d'erreur comme la MAE.

Pour les problèmes de classification, la Cross-Entropy Loss est la plus utilisée, elle mesure la dissimilarité entre la distribution de probabilité prédictive par le modèle et la vérité terrain. Il existe deux types de Cross-Entropy Loss : Binary Cross-Entropy pour les problèmes de classification binaire, et Categorical Cross-Entropy pour les problèmes de classification multilabels.

- La Binary Cross-Entropy est utilisée lorsque nous avons deux classes possibles (0 et 1). Elle pénalise fortement les prédictions qui sont loin des étiquettes réelles, ce qui encourage le modèle à faire des prédictions précises.
- La Categorical Cross-Entropy est utilisée pour les problèmes de classification multilabels. Elle est similaire à la Binary Cross-Entropy mais adaptée pour plus de deux classes.

Nous avons vu que la descente de gradient est une méthode d'optimisation, elle optimise nos paramètres pour minimiser une fonction de coût (fonction erreur). Elle procède par itérations, modifiant progressivement selon le taux d'apprentissage (learning rate) les paramètres du modèle afin de minimiser la fonction de coût.

Dans le cas d'un modèle à un seul paramètre, la mise à jour des paramètres est effectuée en utilisant la règle de mise à jour de la descente de gradient.

La notion de minimum local est introduite, qui est un point où la fonction de coût atteint une valeur basse, mais pas nécessairement la plus basse possible. L'algorithme de descente de gradient peut se retrouver coincé dans ces minimums locaux.

Les points de selle sont également abordés. Ce sont des points où la courbe est à la fois concave et convexe, formant un creux dans une direction et un pic dans l'autre. Pour la descente de gradient, un point de selle est un endroit où le gradient de la fonction coût est nul.

Enfin, la descente de gradient stochastique (SGD) est mentionnée. C'est une variante de la descente de gradient qui résout plusieurs problèmes d'optimisation. La SGD met à jour les poids pour chaque mini-lot du jeu de données, ce qui la rend plus réalisable et moins coûteuse en puissance de calcul pour de grands ensembles de données.

Nous avons vu la backpropagation qui est l'application de la descente de gradient pour les réseaux de neurones. Cet algorithme a été introduit par David E. Rumelhart, Geoffrey E. Hinton et Ronald J. Williams en 1986, il permet de quantifier l'erreur de chaque neurone par rapport à la sortie attendue et de la répartir à travers le réseau pour mettre à jour les poids de chaque couche cachée.

L'apprentissage par backpropagation comprend deux étapes principales : la forward pass et la backpropagation. La forward pass prédit la sortie basée sur les données d'entrée, quantifie l'erreur par rapport à la valeur à prédire y et la calcule selon la fonction de coût choisie. La backpropagation prend ensuite cette erreur et la remonte à travers le réseau, en calculant le gradient de la fonction de coût par rapport à chaque paramètre du réseau.

La backpropagation utilise des calculs différentiels, notamment les dérivées partielles et la règle de la chaîne, pour calculer le gradient de la fonction de coût par rapport à tous les poids et biais du réseau en une seule fois.

Cependant, la backpropagation peut rencontrer des problèmes, tels que le vanishing gradient ou exploding gradient. Ces problèmes sont liés à la manière dont les gradients de la fonction de coût sont calculés et propagés dans le réseau. De nombreuses recherches sont menées pour trouver des alternatives à la backpropagation, mais à l'heure actuelle, aucune n'a encore réussi à surpasser la backpropagation classique.

4.6 Questions

1. **Qu'est-ce que la descente de gradient ?**
 - a. Une méthode d'optimisation pour trouver le minimum d'une fonction
 - b. Un algorithme pour trouver le maximum d'une fonction
 - c. Une méthode pour calculer la dérivée d'une fonction
 - d. Une technique pour trouver le zéro d'une fonction
2. **Quelle est la grande faiblesse de la descente de gradient ?**
 - a. Elle nécessite beaucoup de mémoire pour stocker toutes les données
 - b. Elle peut conduire à des solutions sous-optimales ou nécessiter un grand nombre d'itérations pour atteindre une solution acceptable
 - c. Elle ne peut pas être utilisée avec des fonctions non linéaires
 - d. Elle nécessite que toutes les variables soient indépendantes
3. **Qu'est-ce que le gradient dans le contexte de la descente de gradient ?**
 - a. La direction dans laquelle se trouve le minimum de la fonction de coût
 - b. La vitesse à laquelle la fonction de coût change
 - c. La direction dans laquelle la fonction de coût augmente le plus rapidement
 - d. La valeur maximale que la fonction de coût peut atteindre
4. **Qu'est-ce qu'un minimum local dans le contexte de la descente de gradient ?**
 - a. Le point le plus bas de la fonction de coût
 - b. Un point où la fonction de coût est plus faible que dans les points voisins, mais pas nécessairement le plus bas de tous
 - c. Un point où la fonction de coût est plus élevée que dans les points voisins
 - d. Le point le plus élevé de la fonction de coût
5. **Qu'est-ce que le taux d'apprentissage dans l'algorithme de descente de gradient ?**
 - a. Le nombre d'itérations que l'algorithme va effectuer
 - b. La taille des pas que l'algorithme va faire à chaque itération
 - c. Le nombre de variables que l'algorithme va optimiser simultanément
 - d. La vitesse à laquelle l'algorithme va converger vers le minimum
6. **Pourquoi la descente de gradient stochastique est-elle souvent préférée à la descente de gradient classique ?**
 - a. Elle permet de traiter des ensembles de données plus importants
 - b. Elle est moins susceptible de tomber dans des minimums locaux
 - c. Elle est plus rapide à calculer
 - d. Toutes les réponses précédentes sont correctes
7. **Dans le contexte de la descente de gradient, qu'est-ce qu'un point de selle ?**
 - a. Un point où le gradient de la fonction de coût est nul
 - b. Un point où la fonction de coût atteint son maximum
 - c. Un point où la fonction de coût atteint son minimum
 - d. Un point où la fonction de coût change de signe
8. **Qu'est-ce qu'une fonction de perte (loss function) dans le contexte de l'apprentissage machine ?**
 - a. Elle quantifie la différence entre la prédiction d'un modèle et la valeur réelle.

- b. Elle mesure la précision de la prédiction d'un modèle.
- c. Elle calcule le temps nécessaire pour former un modèle.
- d. Elle évalue la complexité d'un modèle.

9. Pourquoi la MSE est-elle plus sensible aux valeurs aberrantes que la MAE ?

- a. Parce que chaque erreur est élevée au carré dans la MSE.
- b. Parce que chaque erreur est prise en valeur absolue dans la MSE.
- c. Parce que la MSE calcule la somme des erreurs, pas leur moyenne.
- d. Parce que la MSE utilise la racine carrée des erreurs.

Réponse : a

10. Pour un problème de classification multiclass j'utilise quelle loss ?

- a. MSE
- b. MAE
- c. Binary Cross-Entropy
- d. Categorical Cross-Entropy

11. Qu'est-ce que la backpropagation en deep learning ?

- a. Un moyen d'optimiser les poids dans un réseau de neurones
- b. Un algorithme pour générer de nouvelles données
- c. Un outil pour visualiser l'apprentissage d'un modèle
- d. Une méthode pour créer des architectures de réseaux de neurones

Réponse : a

12. Quelle est l'ordre du processus d'apprentissage d'un modèle ?

- a. Calcul de l'erreur, feedforward, calcul du gradient, mise à jour des poids.
- b. Feedforward, calcul du gradient, calcul de l'erreur, mise à jour des poids.
- c. Feedforward, calcul de l'erreur, calcul du gradient, mise à jour des poids.
- d. Calcul du gradient, feedforward, mise à jour des poids, calcul de l'erreur.

13. Quelle est la fonction de la backpropagation ?

- a. Elle calcule l'erreur du modèle
- b. Elle fait une prédiction basée sur les données d'entrée
- c. Elle calcule le gradient de la fonction de coût par rapport à chaque paramètre du réseau
- d. Elle calcule le taux d'apprentissage

14. Qu'est-ce que la < forward propagation >?

- a. Une méthode pour initialiser les poids dans un réseau de neurones.
- b. Le processus d'ajustement des poids dans un réseau de neurones.
- c. Le processus par lequel le modèle fait une prédiction basée sur les données d'entrée.
- d. Une mesure de l'erreur d'un réseau de neurones.

15. Qu'est-ce que le gradient dans le contexte de la backpropagation ?

- a. Une mesure de la vitesse d'apprentissage du modèle.
- b. Une mesure du temps de calcul nécessaire pour former le modèle.
- c. Une mesure de l'importance de chaque poids et biais dans la détermination de la sortie du réseau.

- d. Une mesure de la complexité du modèle.
16. **Qu'est-ce que le problème du vanishing gradient?**
- a. Un problème où les gradients deviennent trop grands, provoquant une mise à jour trop agressive des paramètres.
 - b. Un problème où les gradients deviennent très petits, rendant le réseau de neurones très lent à apprendre.
 - c. Un problème où le réseau de neurones ne peut pas apprendre de nouvelles caractéristiques.
 - d. Un problème où le réseau de neurones oublie les caractéristiques qu'il a apprises précédemment.

4.7 Réponse

1. Qu'est-ce que la descente de gradient ?

Réponse: A - Une méthode d'optimisation pour trouver le minimum d'une fonction

2. Quelle est la grande faiblesse de la descente de gradient ?

Réponse: B - Elle peut conduire à des solutions sous-optimales ou nécessiter un grand nombre d'itérations pour atteindre une solution acceptable

3. Qu'est-ce que le gradient dans le contexte de la descente de gradient ?

Réponse: A - La direction dans laquelle se trouve le minimum de la fonction de coût

4. Qu'est-ce qu'un minimum local dans le contexte de la descente de gradient ?

Réponse: B - Un point où la fonction de coût est plus faible que dans les points voisins, mais pas nécessairement le plus bas de tous

5. Qu'est-ce que le taux d'apprentissage dans l'algorithme de descente de gradient ?

Réponse: B - La taille des pas que l'algorithme va faire à chaque itération

6. Pourquoi la descente de gradient stochastique est-elle souvent préférée à la descente de gradient classique ?

Réponse: D - Toutes les réponses précédentes sont correctes

7. Dans le contexte de la descente de gradient, qu'est-ce qu'un point de selle ?

Réponse: A - Un point où le gradient de la fonction de coût est nul

8. Qu'est-ce qu'une fonction de perte (loss function) dans le contexte de l'apprentissage machine ?

Réponse: A - Elle quantifie la différence entre la prédiction d'un modèle et la valeur réelle

9. Pourquoi la MSE est-elle plus sensible aux valeurs aberrantes que la MAE ?

Réponse: A - Parce que chaque erreur est élevée au carré dans la MSE

10. Pour un problème de classification multiclass j'utilise quelle loss ?

Réponse: D - La Categorical Cross-Entropy

11. Qu'est-ce que la backpropagation en deep learning ?

Réponse: A Un moyen d'optimiser les poids dans un réseau de neurones

12. Quelle est l'ordre du processus d'apprentissage d'un modèle ?

Réponse: C Feedforward, calcul de l'erreur, calcul du gradient, mise à jour des poids

13. Quelle est la fonction de la backpropagation ? Réponse: C

Elle calcule le gradient de la fonction de coût par rapport à chaque paramètre du réseau

14. Qu'est-ce que la forward propagation ?

Réponse: C Le processus par lequel le modèle fait une prédiction basée sur les données d'entrée.

15. Qu'est-ce que le gradient dans le contexte de la backpropagation ?

Réponse: c Une mesure de l'importance de chaque poids et biais dans la détermination de la sortie du réseau.

16. Qu'est-ce que le problème du vanishing gradient ?

Réponse: b Un problème où les gradients deviennent très petits, rendant le réseau de neurones très lent à apprendre.

5 Projet: Projet avec des MLP et mesurer ses performances

Félicitations d'être arrivé jusqu'ici ! Dans ce chapitre, vous allez suivre et réaliser deux projets guidés à partir des connaissances théoriques apprises. Vous aurez un cahier des charges qui vous donnera la procédure à appliquer. Vous allez suivre de bout en bout deux miniprojets. Le premier miniprojet est..... le deuxième miniprojet est..... ils sont utile à apprendre car....

5.1 Projet 1: Créer une calculatrice

Dans ce projet, nous allons plonger dans un problème de régression : prédire le résultat d'une opération mathématique (addition ou multiplication). Nous utiliserons un jeu de données que nous générerons nous-mêmes, composé de paires de nombres et de leurs résultats. Vous serez amené à construire un Modèle de Perceptron Multicouche (MLP) pour résoudre ce problème.

Dans ce chapitre, vous apprendrez à construire des réseaux de neurones avec le framework PyTorch, les projets seront relativement faciles à réaliser, l'idée n'est pas de réaliser un fantastique projet, mais de prendre en main un framework de deep learning.

5.1.1 Cahier des charges

1. **Générer un dataset pour une opération mathématique de deux nombres avec leurs résultats correspondants.** Pour cela, vous pourriez générer des paires de nombres entiers aléatoires dans une certaine plage, disons, de 0 à 100, mais je vous invite à jouer avec ces valeurs et calculer leur somme. Assurez-vous de générer suffisamment de données pour entraîner votre modèle, 1000 échantillons me semble correct, mais pouvez très bien essayer d'en générer 100 ou 10000 pour voir l'effet sur l'entraînement.
2. **Crée un jeu de données pour la phrase d'entraînement et la phase de test** ou alors générer un jeu de données que vous allez diviser pour les phases d'entraînement et de test, vous pouvez utiliser la fonction `train_test_split` de scikit-learn.
3. **Construire un modèle de MLP pour prédire le résultat de l'opération.** Le réseau aura deux neurones en entrée (pour les deux nombres à additionner) et un neurone en sortie (pour le résultat de l'addition). Vous pouvez commencer par un modèle simple avec une seule couche cachée, puis essayer d'ajouter plus de couches et voir l'influence du nombre de couches cachées sur les résultats. Je vous invite à tester des réseaux avec peu de couches cachées, mais avec beaucoup de perceptron par couche cachée et les comparer.
4. **Entraîner le modèle et optimiser les hyperparamètres.** Vous pouvez utiliser une stratégie d'optimisation comme la descente de gradient stochastique (SGD) ou un optimiseur plus sophistiqué comme Adam, j'expliquerai le fonctionnement de l'optimiseur Adam dans le prochain chapitre. Assurez-vous de suivre attentivement la perte d'entraînement et la perte de validation pendant l'entraînement pour éviter le « surajustement » (overfitting).
5. **Évaluer le modèle sur l'ensemble de test.** Une fois que vous êtes satisfait de votre modèle, évaluez-le sur l'ensemble de test pour voir comment il se comporte sur des données qu'il n'a jamais vues auparavant.
6. **Analyser les résultats.** Le réseau est-il capable de prédire correctement les sommes ? Comment la performance varie-t-elle avec différentes plages de nombres ? Le réseau est-il capable de généraliser à des nombres qu'il n'a jamais vus précédemment ?

7. Évaluer le modèle en utilisant différentes métriques de performance et fonction de perte :

- Mean Squared Error (MSE)
- Mean Absolute Error (MAE)
- R-squared (R^2)

5.1.2 importer vos bibliothèques python et configurer votre matériel.

Dans notre mini-projet nous utiliser divers bibliothèques Python très utilisée pour le deep learning.

```
import numpy as np # manipulation de tableaux multidimensionnels
import matplotlib.pyplot as plt # visualisation de données
import torch # notre framework de deep learning
from torch import nn # définition d'architectures de réseaux de neurones (nn comme Neural Network)
from torch.utils.data import DataLoader, TensorDataset # gestion et la manipulation des ensembles de données
from tqdm import tqdm # affiche une barre de progression pendant l'entraînement du modèle
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score # métrique d'évaluation des modèles
import random # génération de nombres aléatoires
from torchsummary import summary # affiche le nombre de paramètres du modèle
```

Après avoir importé ces bibliothèques, la question du matériel se pose. Le deep learning est notoirement gourmand en ressources de calcul, et choisir entre un CPU et un GPU peut faire la différence entre un modèle qui prend des heures ou des jours à s'entraîner. Pour cela, nous allons utiliser PyTorch pour identifier le meilleur matériel disponible :

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

Cette ligne détecte si un GPU compatible avec CUDA est disponible, et dans ce cas, configure device pour utiliser ce GPU. Sinon, le CPU sera utilisé.

Pour une reproductibilité des résultats j'ai ajouté un seed qui vous permettra d'avoir exactement les mêmes résultats que moi si vous choisissez d'utiliser comme valeur 42, c'est une valeur aléatoire vous pouvez utiliser 1 ou toute autre valeur, et à chaque fois que lanceriez un entraînement, vous aurez toujours le même « hasard » produit par l'ordinateur.

```
def set_seed(seed_value=42):
    random.seed(seed_value)
    np.random.seed(seed_value)
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

Pour finaliser la configuration, nous exécutons ces lignes :

```
set_seed(42)

if torch.cuda.is_available():
    print("Le GPU est utilisé")
```

```

else:
    print("Le GPU n'est PAS utilisé, le CPU est utilisé")

```

Ces dernières lignes donnent un retour sur le matériel qui sera utilisé.

5.1.3 Le jeu de données

L'objectif de notre modèle de deep learning est de prédire le résultat d'opérations mathématiques comme les additions **ou** les multiplications sur deux nombres entier. Pour ce faire, nous devons générer un ensemble de données synthétiques qui représente bien le problème en question.

La fonction `create_calculator_dataset` est un générateur de données conçu pour ce but précis. Il génère une matrice où chaque ligne est une paire de nombre entiers aléatoires. Ensuite nous créons l'étiquettes, c'est-à-dire la réponse, ce que notre modèle doit prédire, le résultat d'une addition ou d'une multiplication, ces étiquettes seront stockés dans le vecteur `y`. Les modèles de deep learning sont bien plus performant quand les données d'entraînement sont normalisés, alors nous divisons `X` et `y` par des constantes qui garantit que les valeurs restent dans une plage plus maniable pour les modèles de machine learning, entre 0 et 1.

```

def create_calculator_dataset(num_samples, min_value, max_value, operation):
    X = np.random.randint(min_value, max_value+1, (num_samples, 2))
    if operation == 'add':
        y = X[:, 0] + X[:, 1]
    elif operation == 'multiply':
        y = X[:, 0] * X[:, 1]
    else:
        raise ValueError("Operation not recognized. Use 'add' or 'multiply'")

    X = X / max_value
    if operation == 'add':
        y = y / (2 * max_value)
    elif operation == 'multiply':
        y = y / (max_value * max_value)
    return X, y

```

- `num_samples` définit le nombre total d'échantillons que vous souhaitez générer. Plus vous aurez d'échantillons plus votre modèle sera reboste, cela enrichie de variété de combinaison possible votre dataset, ce qui aidera votre modèle à être plus performant sur une plus grande variété de données.
- `min_value` et `max_value` fixent la plage de valeurs pour les opérandes, nous irons de 0 à 100
- `operation` indique le type d'opération à effectuer.

Nous utiliserons notre fonction plus tard pour générer notre jeu de données.

5.1.4 Architecture du réseau

La prochaine étape consiste à définir l'architecture du réseau de neurones qui fera l'objet de notre étude. Nous allons utiliser PyTorch. Il existe deux approches pour définir une architecture de deep learning avec PyTorch: l'une à l'aide `Sequential` de PyTorch et l'autre via une classe personnalisée qui hérite de `nn.Module`.

5.1.4.1 Modèle séquentiel

La première approche exploite la capacité de `Sequential` pour définir une séquence linéaire de couches. C'est une méthode simple et efficace pour des architectures avec un enchaînement ordonné de couche. Je vous la présente ici, mais je ne l'utiliserai que très peu, elle sera parfois utilisée dans des classes personnalisées héritant de `nn.module` pour définir une partie d'une architecture sophistiquée que nous verrons dans ce livre.

```
sequential_model = nn.Sequential(  
    nn.Linear(2, 40, bias=True), # Couche d'entrée  
    nn.ReLU(),  
    nn.Linear(40, 40, bias=True), # Première couche cachée  
    nn.ReLU(),  
    nn.Linear(40, 40, bias=True), # Deuxième couche cachée  
    nn.ReLU(),  
    nn.Linear(40, 40, bias=True), # Troisième couche cachée  
    nn.ReLU(),  
    nn.Linear(40, 40, bias=True), # Quatrième couche cachée  
    nn.ReLU(),  
    nn.Linear(40, 1, bias=True) # Couche de sortie  
)
```

Dans cette architecture, chaque `nn.Linear` représente un perceptron, il est nommé comme cela puisque qu'un perceptron effectue des calculs linéaires qui effectuent une somme pondérée des entrées. Dans le code le paramètre `bias=True` est le réglage par défaut pour les perceptrons dans PyTorch, indiquant que chaque perceptron à son propre biais individuel qui s'ajoute à sa sortie. Le paramètre `bias` est par défaut à `True`, j'aurai pu ne pas le préciser qu'il en serait de même pour la définition de l'architecture du réseau, je l'ai explicitement mentionné ici pour en clarifier la présence, si vous ne voyez pas `bias=False`, c'est que le biais est inclus.

- **`nn.Linear(2, 40)`**: représente la couche d'entrée du réseau. Elle prend un vecteur d'entrée de taille 2 et le transforme en un vecteur de dimension 40. Ce nombre (40) est arbitraire et pourrait être modifié selon les besoins de l'application.
- **`nn.ReLU()`**: La fonction d'activation ReLU (Rectified Linear Unit) est utilisée après chaque couche linéaire. Elle introduit une non-linéarité dans le modèle, ce qui permet au réseau d'apprendre des fonctions non linéaires.
- **`nn.Linear(40, 40)`**: Ces couches représentent les couches cachées du réseau. Chacune prend un vecteur de taille 40 en entrée et produit un vecteur de taille 40 en sortie. Encore une fois, le choix de 40 est arbitraire. L'idée est souvent de conserver la même dimensionnalité pour toutes les couches cachées pour simplifier l'architecture, mais ce n'est pas une règle stricte. Il faut quand même éviter de réduire vers la fin du réseau, cela compresserait l'information qui avait été développé par les couches précédentes, soit on conserve la même largeur des couches tout au long, soit nous élargissons le réseau vers les dernières couches.
- **`nn.Linear(40, 1)`**: Cette dernière couche linéaire transforme le vecteur de taille 40 en un vecteur de taille 1, qui correspond à la sortie du réseau.

L'architecture du réseau est illustrée ci-dessous :

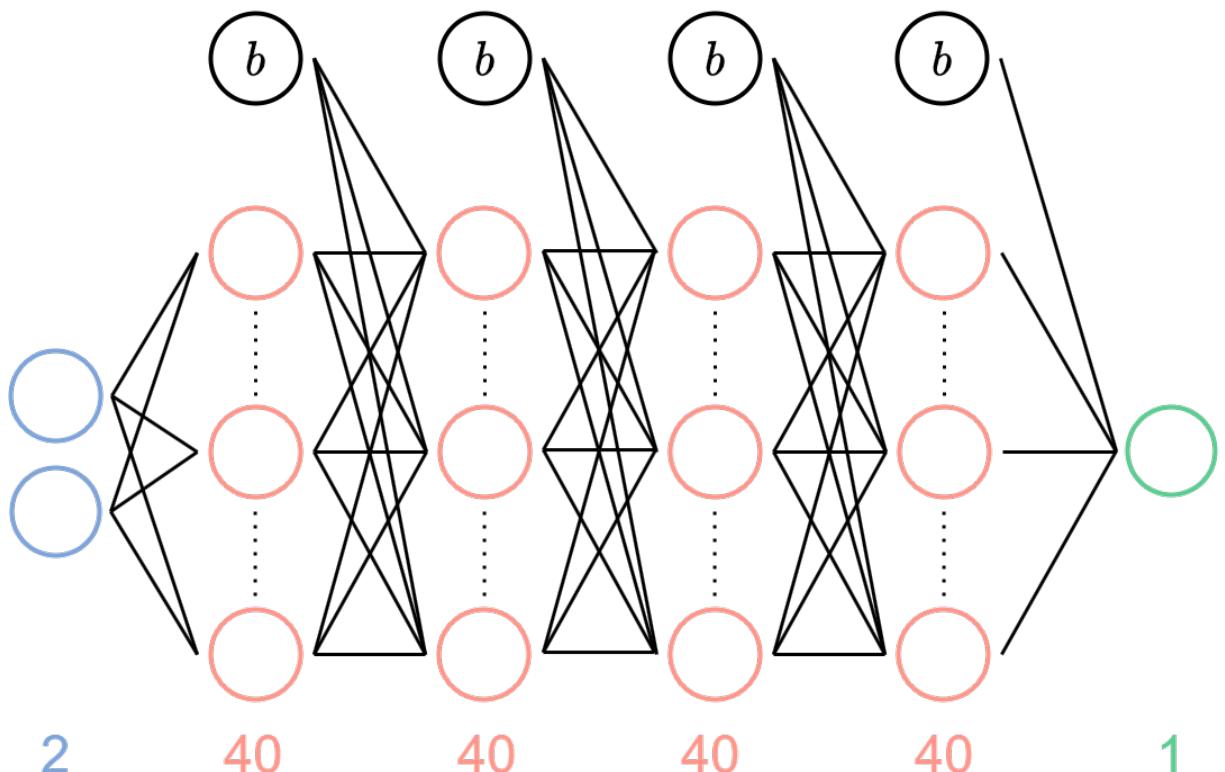


Figure 40 : Architecture d'un réseau ayant 2 entrées, 4 couches cachées ayant chacune 40 perceptrons par couche cachée et 1 perceptron dans la couche de sortie, chaque perceptron à son propre biais.

5.1.4.1.1 Pourquoi 40 ?

Le choix du nombre « 40 » est une valeur complètement aléatoire. En général, des valeurs correspondant à une puissance de 2 sont généralement choisies (2, 4, 8, 16, 32, 64...). Il est possible de choisir une architecture plus large, mais cela risque de ne pas améliorer les performances et augmente le risque de sur-apprentissage (overfitting).

5.1.4.2 Quelle architecture choisir ?

Le choix de l'architecture de votre modèle de deep learning est davantage un art qu'une science exacte. Il n'existe pas de théorème mathématique pour déterminer quelle doit être votre architecture pour votre problème et avec vos données. La définition de votre architecture est un processus itératif d'expérimentation et d'ajustement. Vous apprendrez rapidement à choisir les bonnes architectures de deep learning en expérimentant. C'est pour cela que je vous conseille vivement d'essayer par vous-même les codes du livre qui se trouvent sur le répertoire [GitHub](#) du livre.

5.1.4.2.1 Complexité du Problème

Le premier facteur à considérer est la complexité intrinsèque du problème à résoudre. Si on décide de créer un jeu de donnée ayant une plage de 0 à 100 000 à la place de 0 à 100 comme actuellement, la tâche devient plus sophistiquée, et nécessitera une architecture plus sophistiquée pour résoudre un problème plus sophistiqué.

5.1.4.2.2 Contraintes Computationnelles

La disponibilité des ressources computationnelles peut également influencer le choix de l'architecture. Des architectures plus grandes et plus complexes nécessiteront plus de mémoire et de temps de calcul, ce qui peut être problématique pour des applications en temps réel ou des dispositifs avec des capacités limitées.

Des entraînement sur des réseaux de taille moyenne (en million de paramètres) peut nécessiter plusieurs jours d'entraînement

5.1.4.3 Modèle Basé sur une Classe Personnalisée

Gagner en flexibilité vous permettra de créer de meilleure architecture de deep learning. Cette seconde méthode consiste à définir une classe qui hérite de `nn.Module`. Elle permet une personnalisation plus poussée de l'architecture, en particulier pour des structures de réseau plus complexes. Par conséquent, c'est la méthode utiliser par les chercheurs et ingénieurs du domaine et celle que nous utiliserons tout le long du livre.

Lorsque vous définissez une classe personnalisée en PyTorch, deux méthodes essentielles doivent être implémentées :

- `__init__`: Cette méthode est appelée lors de l'initialisation de votre objet. Ici, vous définirez tous les composants nécessaires à votre réseau, tels que les couches linéaires, les fonctions d'activation et autres techniques de deep learning.
- `forward`: La méthode `forward` définit la logique de la propagation avant, autrement dit le chemin emprunté par le signal à travers le réseau lors de l'apprentissage ou de l'évaluation.

Maintenant que ceci est compris créons une classe personnalisée reprenant l'architecture conçue avec `nn.Sequential`.

```
class MLP(nn.Module):  
    def __init__(self):  
        super(CustomModel, self).__init__()  
        self.layer1 = nn.Linear(2, 40)  
        self.layer2 = nn.Linear(40, 40)  
        self.layer3 = nn.Linear(40, 40)  
        self.layer4 = nn.Linear(40, 40)  
        self.layer5 = nn.Linear(40, 40)  
        self.layer_out = nn.Linear(40, 1)  
        self.relu = nn.ReLU()  
  
    def forward(self, x):  
        x = self.layer1(x)  
        x = self.relu(x)  
        x = self.layer2(x)  
        x = self.relu(x)  
        x = self.layer3(x)  
        x = self.relu(x)  
        x = self.layer4(x)  
        x = self.relu(x)  
        x = self.layer5(x)  
        x = self.relu(x)
```

```

        x = self.layer_out(x)
    return x

```

La fonction `super(MLP, self).__init__()` permet d'hériter de toutes les fonctionnalités de la classe parente `nn.Module`. Les couches sont définies dans le constructeur (`__init__`) et la logique de la propagation vers l'avant (la forward pass) est mise en place dans la méthode `forward`. Si ceci n'est pas très clair pour vous, je vous invite à comprendre les concepts de Programmation Orienté Objet (POO) en Python.

5.1.4.3.1 Classe Évolutive

Comme dit précédemment, pour découvrir la bonne architecture il faut expérimenter alors nous allons la définir notre class de cette manière :

```

class MLP(nn.Module):
    def __init__(self, input_size=2, hidden_size=40, output_size=1,
num_hidden_layers=4, activation_fn=nn.ReLU()):
        super(MLP, self).__init__()
        self.layers = nn.ModuleList([nn.Linear(input_size, hidden_size)])
        self.layers.extend([nn.Linear(hidden_size, hidden_size) for _ in
range(num_hidden_layers-1)])
        self.output_layer = nn.Linear(hidden_size, output_size)
        self.activation_fn = activation_fn

    def forward(self, x):
        for layer in self.layers:
            x = self.activation_fn(layer(x))
        x = self.output_layer(x)
        return x

```

Dans cette version de notre classe `MLP` (*Multi Layer Perceptron*), nous avons introduit plusieurs arguments dans le constructeur pour permettre une flexibilité, en utilisant `nn.ModuleList`, une classe conteneur spécialement conçue pour stocker une liste de couches. Nous utilisons une boucle pour générer dynamiquement le nombre désiré de couches cachées, tel que spécifié par le paramètre `num_hidden_layers`. Cet artifice rend l'architecture du réseau facilement modifiable, simplifiant ainsi les expérimentations avec différentes profondeurs de réseau. Vous pourrez l'influencer d'ajouter plus ou moins de profondeur à votre réseau. Vous pourrez aussi ajouter plus ou moins de largeur (nombre d'unité par couche) à votre réseau avec le paramètre `hidden_size`.

La méthode `forward` représente la procédure de propagation avant (ou forward pass) du réseau. Lorsqu'un ensemble de données d'entrée (ou un batch) est introduit dans le réseau, il passe successivement à travers chaque couche cachée, subissant à chaque étape une transformation linéaire suivie d'une activation non-linéaire (ici, ReLU). Finalement, les données transformées sont acheminées vers la couche de sortie à travers `self.output_layer`.

Je vous invite à expérimenter avec les différentes fonctions d'activation. Chacune a ses propres avantages et inconvénients, et le choix peut affecter significativement les performances de votre modèle. Ce n'est qu'en expérimentant que vous pourrez véritablement comprendre leur impact et leur utilité dans des contextes spécifiques.

5.1.5 Fonctions d'entraînement et de préparation

Maintenant que nous avons posé les bases de notre infrastructure — la création de données, la constitution du modèle et la configuration du matériel — nous pouvons aborder la quintessence de l'apprentissage machine : le processus d'entraînement. Maintenant nous allons jusqu'à la mise à jour des poids du modèle

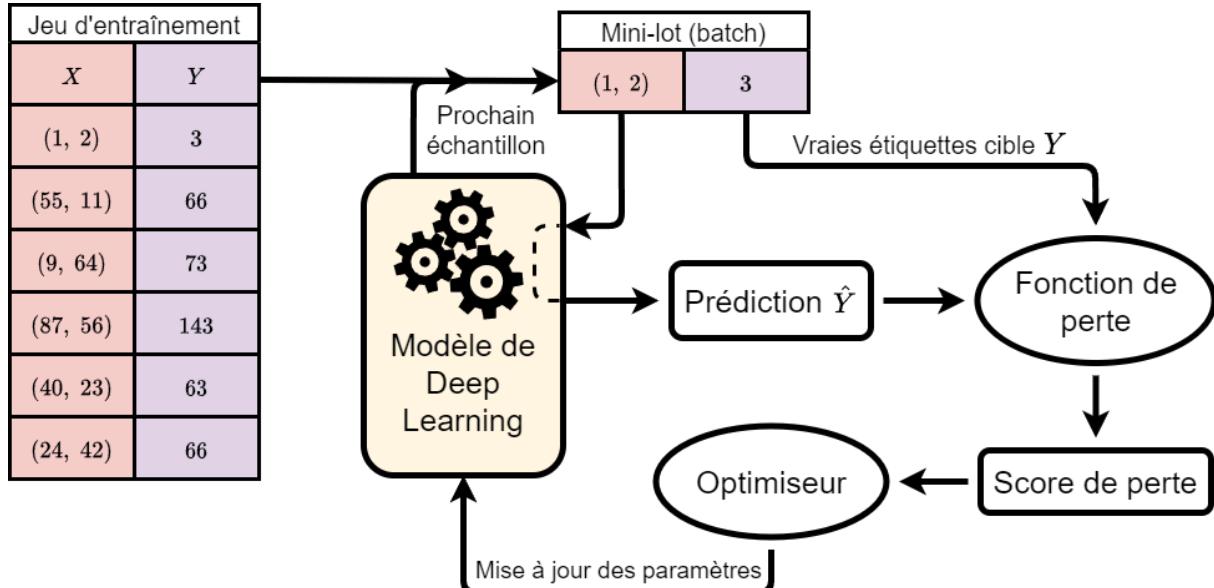


Figure 41 : Représentation du processus d'entraînement d'un modèle de deep learning à partir d'un ensemble de données.

5.1.5.1 Dataloader: Diviser pour mieux régner

Nous avons déjà vu comment générer un jeu de données synthétiques. Le DataLoader intervient à ce stade pour diviser ces données en lots (batches), ce qui permet une mise à jour plus fréquente des poids du réseau, accélérant ainsi la convergence. Si vous n'utilisez pas de DataLoader, votre réseau fera une prédiction sur le jeu de données en entier, soit une « époque », avant de faire la moindre optimisation. Par ailleurs, il est impératif de séparer les données en ensembles d'entraînement et de validation. L'ensemble de validation sert à estimer la performance du modèle sur des données non vues pendant l'entraînement, et offre une vision sur un surajustement potentiel, et à partir de combien d'époque.

```
from sklearn.model_selection import train_test_split

def create_calculator_dataloaders(num_samples=1000, min_value=0, max_value=10,
                                   operation='add', batch_size=64, val_split=0.2):
    X, y = create_calculator_dataset(num_samples, min_value, max_value,
                                      operation)
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=val_split,
                                                      random_state=42)

    X_train_tensor = torch.tensor(X_train, dtype=torch.float32).to(device)
    y_train_tensor = torch.tensor(y_train, dtype=torch.float32).to(device)
    X_val_tensor = torch.tensor(X_val, dtype=torch.float32).to(device)
    y_val_tensor = torch.tensor(y_val, dtype=torch.float32).to(device)
```

```

    train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
    train_dataloader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
    val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
    val_dataloader = DataLoader(val_dataset, batch_size=batch_size)

    return train_dataloader, val_dataloader

```

5.1.5.2 L'Optimiseur: Le Maître d'Œuvre

L'optimiseur c'est le type d'algorithme de descente de gradient choisi pour minimiser la fonction perte. Deux des plus populaires sont la Descente de Gradient Stochastique (SGD) et Adam. Adam n'a pas encore été présenté, c'est une évolution de la SGD, je vous invite à essayer les deux pour voir s'il y a ou non des différences. L'algorithme Adam sera présenté dans un autre chapitre.

```

def get_optimizer(model, optimizer_name="sgd", lr=0.1):
    if optimizer_name.lower() == "adam":
        return torch.optim.Adam(model.parameters(), lr=lr)
    elif optimizer_name.lower() == "sgd":
        return torch.optim.SGD(model.parameters(), lr=lr)
    else:
        raise ValueError("Optimizer not recognized. Use 'adam' or 'sgd'.")

```

5.1.5.3 Le Processus d'Entraînement: Une Symphonie en Plusieurs Actes

L'entraînement d'un modèle de deep learning est un processus itératif qui se déroule en plusieurs étapes clés :

- Propagation avant, la prédiction (Forward Pass):** Les données d'entrée traversent le réseau de neurones pour générer une prédiction.
- Calcul de la Perte:** La prédiction est comparée à la vérité terrain à l'aide d'une fonction de perte, généralement une mesure d'erreur.
- Rétropropagation (Backpropagation):** Le gradient de cette perte est calculé par rapport à chaque paramètre du modèle.
- Optimisation:** Les poids du modèle sont mis à jour dans la direction qui minimise la perte.

Ce cycle est répété pour chaque lot d'échantillons (batch) de l'ensemble d'entraînement jusqu'à ce que le modèle ait vu toutes les données une fois. Chaque passage complet à travers l'ensemble de données est appelé une époque.

```

def train_model(model, train_dataloader, val_dataloader, epochs=10,
optimizer_name="sgd", lr=0.1):
    model.to(device)
    optimizer = get_optimizer(model, optimizer_name, lr)
    loss_fn = nn.MSELoss()
    train_loss_history, val_loss_history = [], []

    for epoch in tqdm(range(epochs), desc="Training Progress"):
        model.train()
        running_train_loss = 0.0
        for X_batch, y_batch in train_dataloader:
            optimizer.zero_grad()

```

```

outputs = model(X_batch)
loss = loss_fn(outputs.view(-1), y_batch)
loss.backward()
optimizer.step()
running_train_loss += loss.item()

epoch_train_loss = running_train_loss / len(train_dataloader)
train_loss_history.append(epoch_train_loss)

model.eval()
running_val_loss = 0.0
with torch.no_grad():
    for X_batch, y_batch in val_dataloader:
        outputs = model(X_batch)
        loss = loss_fn(outputs.view(-1), y_batch)
        running_val_loss += loss.item()

epoch_val_loss = running_val_loss / len(val_dataloader)
val_loss_history.append(epoch_val_loss)

return train_loss_history, val_loss_history

```

5.1.6 Entraînement et visualisation des résultats

Il est temps de passer à la phase de l'entraîner notre de modèle. L'entraînement est le processus par lequel le modèle ajuste ses paramètres internes (ou « poids ») pour minimiser une fonction de perte. Cette fonction mesure l'écart entre les prédictions du modèle et les vérités de terrain. L'objectif est de parvenir à un modèle qui généralise bien, c'est-à-dire qui performe efficacement sur de nouvelles données jamais vues auparavant.

5.1.6.1 Initialisation du modèle

Nous débutons par la définition des paramètres (appelé « hyper-paramètre ») qui vont régir la création de notre jeu de données ainsi que la configuration de l'architecture de notre modèle.

```

# Paramètres pour la création du jeu de données
num_samples = 10000      # échantillons générer. Ex: 5000, 10000, etc.
min_value = 0             # Valeur minimale des nombres. Ex: 0, 10, -10, etc.
max_value = 100           # Valeur maximale des nombres. Ex: 100, 200 etc.
operation = 'add'         # Opération à effectuer. 'add' ou 'multiply'.
batch_size = 64            # Taille des lots pour l'entraînement. Ex: 32, 64, etc.
learning_rate = 0.1        # Taux d'apprentissage. "adam", 0.001 ou "sgd", 0.1
optimizer = "sgd"          # Optimiseur. "adam" ou "sgd"

# Paramètres pour le modèle MLP
input_size = 2              # Taille de l'entrée. Pour notre cas, c'est toujours 2.
hidden_size = 16             # Nombre de perceptron dans les couches cachées. Ex: 8, 16, etc.
output_size = 1              # Taille de la sortie. Pour notre cas, c'est toujours 1.
num_hidden_layers = 3         # Nombre de couches cachées. Ex: 1, 2, 3, 4, etc.
activation_fn = nn.ReLU()    # Fonction d'activation. Ex: nn.ReLU(), nn.Sigmoid()
nn.Tanh() et nn.LeakyReLU().

```

```
# Paramètres pour l'entraînement
epochs = 10      # Nombre d'époques pour l'entraînement. Ex: 5, 10, 20, etc.
```

Chaque hyperparamètre a un rôle précis :

- **num_samples** influence la quantité de données sur laquelle le modèle va s'entraîner. Un plus grand nombre d'échantillons peut améliorer la généralisation du modèle, mais augmente également le temps d'entraînement, car votre modèle exécutera plus d'itération par époque.
- **min_value** et **max_value** déterminent la plage des données et peuvent être ajustés pour tester la capacité du modèle à généraliser sur différentes plages de nombres.
- **operation** définit l'opération arithmétique que le modèle doit apprendre à résoudre. Changer cette opération peut grandement influencer la complexité de la tâche d'apprentissage, les multiplications sont plus difficiles à < approximer > qu'une addition.
- **batch_size** a un impact sur la stabilité et la vitesse de l'entraînement. Des lots plus petits peuvent conduire à une convergence plus rapide mais peuvent aussi causer une instabilité pendant l'entraînement.
- **input_size**, **hidden_size**, **output_size**, et **num_hidden_layers** définissent l'architecture du réseau. Varier ces valeurs peut avoir des conséquences significatives sur la capacité du modèle à apprendre des représentations complexes.
- **activation_fn** est cruciale pour introduire des non-linéarités dans le modèle, permettant d'apprendre des fonctions plus complexes que de simples combinaisons linéaires des entrées.
- **epochs** détermine le nombre de fois que le modèle verra l'ensemble du jeu de données. Un nombre insuffisant d'époques peut conduire à un sous-apprentissage, tandis qu'un nombre trop élevé peut causer du surapprentissage.
- **learning_rate** : Ce paramètre contrôle la taille des pas faits dans la direction du gradient. Un taux d'apprentissage trop élevé peut conduire à des sauts trop grands, manquant potentiellement le minimum, tandis qu'un taux trop bas peut ralentir l'entraînement et se coincer dans un minimum local.
- **optimizer** : Le choix de l'optimiseur peut influencer la vitesse et la qualité de la convergence du modèle. -SGD (Stochastic Gradient Descent) est un choix classique, mais des optimiseurs plus avancés comme Adam peuvent accélérer l'entraînement et parfois aboutir à de meilleures performances.

L'expérimentation avec diverses configurations d'hyperparamètres est essentielle pour maîtriser l'art du deep learning. Je vous incite à manipuler le taux d'apprentissage, à osciller entre des valeurs élevées pour une convergence rapide et des valeurs faibles pour une plus grande finesse dans la recherche du minimum de la fonction de perte. Interrogez-vous sur l'optimiseur le plus adapté à votre problème : serait-ce Adam avec ses corrections adaptatives qui facilitent la navigation dans des paysages de perte complexes, ou SGD, dont la simplicité et la constance pourraient se montrer avantageuses dans certains contextes ?

Ne négligez pas l'impact potentiel des fonctions d'activation : chaque choix, qu'il s'agisse de ReLU, de Tanh, ou de LeakyReLU, apporte une dynamique différente à la propagation de l'activation dans votre réseau. Le deep learning est une discipline empirique où l'intuition

se forge à travers la pratique et l'expérimentation. En testant une pléthore de combinaisons, vous affûterez non seulement la performance de votre modèle, mais aussi votre compréhension conceptuelle. Chaque essai est une étape vers une expertise accrue et chaque erreur, une leçon précieuse. C'est en explorant activement cette vaste étendue de possibilités que vous progresserez dans le domaine fascinant du deep learning.

Pour ceux qui visent l'excellence, je vous encourage à incorporer dans mon code des techniques d'optimisation des hyperparamètres, comme la recherche par grille (grid search) ou la recherche aléatoire (random search). Ces méthodes systématiques permettent d'explorer l'espace des hyperparamètres de manière structurée et peuvent être particulièrement utiles lorsqu'on travaille avec des jeux de données de petite ou moyenne taille. Cependant, dans des contextes professionnels où l'on doit composer avec des modèles de grande envergure et des contraintes temporelles strictes, ces méthodes peuvent se révéler prohibitives en termes de ressources et de temps.

Pour pallier cela, des techniques plus avancées telles que l'optimisation bayésienne offre un compromis entre exhaustivité et efficience, en se basant sur des principes probabilistes pour guider la recherche vers les régions les plus prometteuses de l'espace des hyperparamètres.

N'oubliez pas que l'entraînement de modèles de deep learning est souvent une entreprise de longue haleine. Les compétences acquises à travers l'expérimentation sur des modèles plus simples vous seront inestimables lorsque vous aborderez des problématiques plus ardues. Chaque expérimentation vous rapproche de la maîtrise du deep learning, et chaque échec est une occasion d'apprentissage. C'est en osant explorer et en apprenant de chaque essai que vous progresserez dans cet art.

5.1.6.2 Préparation des données

Les données sont préparées à l'aide de la fonction `create_calculator_dataloaders`, qui génère des jeux de données pour l'entraînement et la validation et les encapsule dans des `DataLoaders`. Cela permet de les découper en lots et d'optimiser le processus d'entraînement.

```
train_dataloader, val_dataloader = create_calculator_dataloaders(  
    num_samples=num_samples, min_value=min_value, max_value=max_value,  
    operation=operation, batch_size=batch_size, val_split=0.2  
)
```

5.1.6.3 Processus d'entraînement

Maintenant appelons notre fonction `train_model()` avec les hyper-paramètres choisis pour lancer le cycle d'apprentissage, orchestrant la descente de gradient par l'intermédiaire de l'optimiseur sélectionné (SGD ou Adam), ajustant les poids de notre modèle à travers les époques, tout en gardant trace de l'historique de perte pour l'analyse future.

```
train_loss_history, val_loss_history = train_model(  
    model, train_dataloader, val_dataloader, epochs=epochs,  
    optimizer_name=optimizer, lr=learning_rate  
)
```

5.1.6.4 Affichage des courbes de perte

Nous observons la représentation graphique des courbes de perte d'entraînement et de validation d'un modèle au fil des époques. L'axe des abscisses indique les époques, qui sont les cycles complets de passage de l'ensemble des données d'entraînement à travers le modèle. L'axe des ordonnées montre la perte, calculée par une fonction de coût, qui mesure l'écart entre les prédictions du modèle et les valeurs réelles. L'échelle logarithmique utilisée sur cet axe met en évidence les variations de la perte, même quand ces variations sont très petites, permettant une meilleure distinction entre les phases d'amélioration rapide et plus lente de la performance du modèle. La courbe bleue représente la perte d'entraînement qui diminue avec le temps, reflétant l'apprentissage du modèle, tandis que la courbe orange montre la perte de validation, qui évalue la performance du modèle sur un ensemble de données non vu pendant l'entraînement, un indicateur clé de la capacité du modèle à généraliser.

```
plt.figure(figsize=(12, 6))
plt.plot(train_loss_history, label="Training Loss")
plt.plot(val_loss_history, label="Validation Loss")
plt.title("Training and Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.yscale('log') # Appliquer une échelle logarithmique
plt.legend()
plt.show()
```

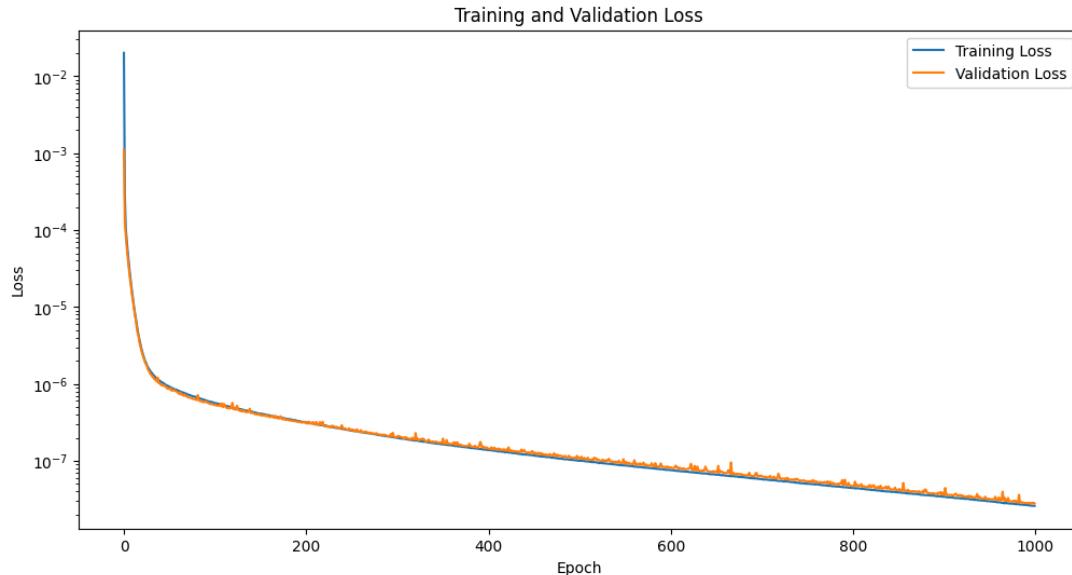


Figure 42 : Courbes de perte d'entraînement (bleu) et de validation (orange) en fonction des époques, sur une échelle logarithmique. Ces tendances illustrent comment le modèle apprend et s'ajuste pour minimiser l'erreur à travers les cycles d'entraînement

5.1.6.5 Résumé de notre architecture

La fonction `summary` a été utilisée pour fournir un aperçu concis de l'architecture de notre modèle, révélant la configuration des couches, leurs formes de sortie et le décompte des paramètres. L'architecture se compose de quatre couches linéaires alternées avec des activations ReLU, chaque couche linéaire ayant respectivement 48, 272, 272 et 17 paramètres.

```

summary(model, input_size=(1, input_size))

-----

```

Layer (type)	Output Shape	Param #
Linear-1	[-1, 1, 16]	48
ReLU-2	[-1, 1, 16]	0
Linear-3	[-1, 1, 16]	272
ReLU-4	[-1, 1, 16]	0
Linear-5	[-1, 1, 16]	272
ReLU-6	[-1, 1, 16]	0
Linear-7	[-1, 1, 1]	17

```

Total params: 609
Trainable params: 609
Non-trainable params: 0
-----
```

5.1.7 Test et Visualisation: Révéler l'Âme du Modèle

Après avoir entraîné un modèle, l'étape suivante consiste à évaluer son efficacité et à interpréter ses performances. Cette phase ne se limite pas à l'obtention d'une métrique de performance. Il s'agit aussi de comprendre ce que le modèle a appris et comment il prend ses décisions.

```

# Fonction de test du modèle
def test_model(model, num_samples, min_value, max_value, operation):
    X_test, y_test = create_calculator_dataset(num_samples, min_value, max_value,
                                                operation)
    X_test_tensor = torch.tensor(X_test, dtype=torch.float32).to(device)
    with torch.no_grad():
        y_pred = model(X_test_tensor)
        y_pred_denormalized = (y_pred * (2 * max_value)).squeeze().cpu().numpy()
    return y_pred_denormalized, y_test * (2 * max_value)

# Fonction de tracé des prédictions
def plot_predictions(y_true, y_pred):
    plt.figure(figsize=(12, 8))
    plt.scatter(range(len(y_true)), y_true, color='green', label="True values")
    plt.scatter(range(len(y_pred)), y_pred, color='blue', alpha=0.5,
                label="Predictions")
    for i, (true, pred) in enumerate(zip(y_true, y_pred)):
        difference = pred - true
        if abs(difference) > 1:
            plt.annotate("", xy=(i, pred), xytext=(i, true),
                        arrowprops=dict(arrowstyle="->", color='red'))
    plt.title("True values vs. Predictions")
    plt.legend()
    plt.show()
```

Dans le code ci-dessus, `test_model` est une fonction qui prend en entrée un modèle entraîné, un nombre d'échantillons, une plage de valeurs et une opération mathématique. Elle génère un ensemble de données de test (`X_test` et `y_test`) et utilise le modèle pour faire des prédictions. L'utilisation de `torch.no_grad()` est utilisé lorsqu'on fait une inférence, ce qu'il se

passee, c'est qu'on indique à PyTorch de ne pas stocker les gradients du modèle, d'où le « no grad ». Lors de l'entraînement, non seulement les poids du modèle sont chargés en mémoire, mais également les gradients associés à ces poids. Ces gradients, nécessaires pour la mise à jour des poids durant l'apprentissage, doublent pratiquement la quantité de mémoire requise. En revanche, lors de l'inférence, les gradients ne sont pas nécessaires car on n'entraîne pas le modèle, donc pas besoin des gradients et on économise beaucoup de RAM à notre GPU.

La fonction `plot_predictions` visualise les valeurs réelles (`y_true`) et les prédictions du modèle (`y_pred`). Les annotations rouges indiquent où les prédictions diffèrent significativement des valeurs réelles, ce qui aide à identifier les points où le modèle est moins précis.

Le code suivant montre comment ces fonctions sont utilisées pour tester et évaluer un modèle :

```
#? Test et visualisation des prédictions
num_samples = 2000

y_pred, y_true = test_model(model, num_samples, min_value, max_value, operation)
plot_predictions(y_true, y_pred)

#? Évaluation des performances à l'aide de différentes métriques
mse = mean_squared_error(y_true, y_pred)
mae = mean_absolute_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)

print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"R-squared (R2): {r2:.4f}")

>>> Mean Squared Error (MSE): 1.6901
>>> Mean Absolute Error (MAE): 1.0354
>>> R-squared (R2): 0.9990
```

Ce segment de code évalue le modèle en utilisant des métriques standard comme l'erreur quadratique moyenne (MSE), l'erreur absolue moyenne (MAE), et le coefficient de détermination R^2 , qui mesure la proportion de la variance des données expliquée par le modèle. Ces mesures fournissent une vue quantitative de la performance du modèle.

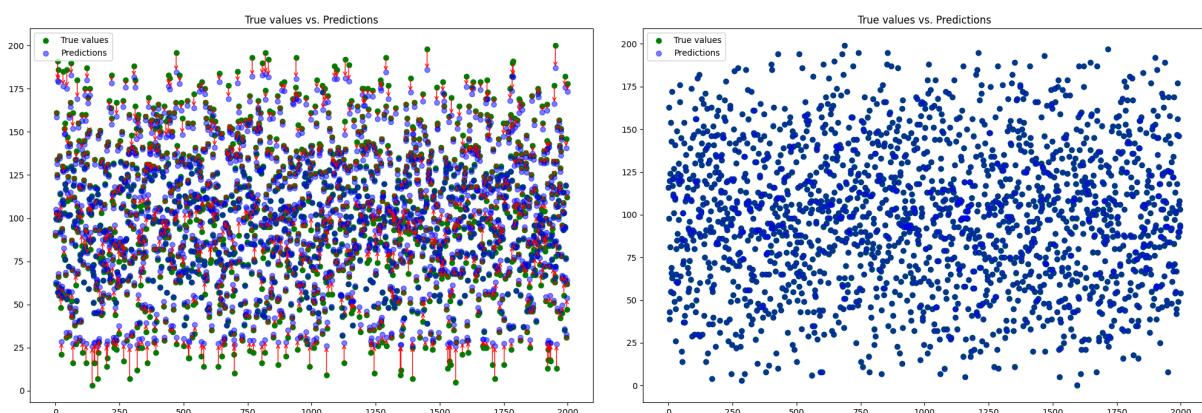


Figure 43 : Modèle sous-entraîné à gauche, modèle bien entraîné à droite. La différence de l'erreur est indiquée par une flèche rouge.

Il est important de tester vos modèles et si possible de rendre ça visuel pour le développement de vos modèles. Cette phase permet non seulement de mesurer la performance, mais aussi de comprendre où et comment le modèle peut être amélioré.

5.1.7.1 Faire des inférences à la main

Pour ceux qui souhaitent tester le modèle avec leurs propres valeurs, la fonction `predict_sum` est conçue à cet effet. Elle permet de faire des prédictions sur des paires de nombres de votre choix. Voici comment elle fonctionne :

```
def predict_sum(model, num1, num2, max_value=100):
    # Normalisation des entrées
    X_input = torch.tensor([[num1/max_value, num2/max_value]],
    dtype=torch.float32).to(device)

    # Inférence
    with torch.no_grad():
        prediction = model(X_input)

    # Retour à l'échelle originale pour obtenir le résultat
    predicted_sum = prediction * (2 * max_value)
    return predicted_sum.item()

# Exemple d'utilisation
result = predict_sum(model, 50, 50)
print(f"La somme prédite de 50 et 50 est : {result:.2f}")
print(f"La somme réelle de 50 et 50 si on arrondit au entier est :
{np.round(result):.0f}")

# Test de la fonction avec d'autre valeur
result = predict_sum(model, 6, 6)
print(f"La somme prédite de 6 et 6 est : {result:.2f}")
print(f"La somme réelle de 6 et 6 si on arrondit au entier est :
{np.round(result):.0f}")

>>> La somme prédite de 50 et 50 est : 100.02
>>> La somme réelle de 50 et 50 si on arrondit au entier est : 100

>>> La somme prédite de 6 et 6 est : 11.89
>>> La somme réelle de 6 et 6 si on arrondit au entier est : 12
```

Cette fonction est un outil pratique pour expérimenter avec le modèle et observer comment il gère différentes entrées numériques.

5.2 Projet 2: Reconnaissance des chiffres manuscrits

5. Projet 2: Reconnaissance des chiffres manuscrits (classification) MNIST

1. **Description du projet:** En utilisant le dataset « MNIST », les étudiants devront construire un modèle de réseau de neurones pour classer les images de chiffres manuscrits. Ce projet leur permettra de pratiquer leurs compétences en matière de prétraitement des données d'image, de construction de modèles et d'optimisation des hyperparamètres.

2. Cahier des charges:

1. Importer et explorer le dataset
2. Prétraiter les données (normalisation, remodelage, etc.)
3. Diviser le dataset en ensembles d'entraînement, de validation et de test
4. Construire un MLP pour classer les images de chiffres
5. Entrainer le modèle et optimiser les hyperparamètres
6. Évaluer le modèle sur l'ensemble de test
7. Évaluer le modèle en utilisant différentes mesures de performance :
 - Accuracy
 - Precision
 - Recall
 - F1 score
 - Temps de calcul

9. Gérer le déséquilibre des données :

- Analyser la distribution des classes dans le dataset
- Discuter des méthodes pour gérer le déséquilibre des données (sous-échantillonnage, suréchantillonnage, augmentation de données)

3. **Lien vers le dataset:** Vous pouvez trouver le dataset « MNIST » dans le package torch-vision.datasets ou le télécharger directement à partir de [ce lien](<http://yann.lecun.com/exdb/mnist/>).

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split

batch_size = 64
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Chargement des datasets
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
transform=transform, download=True)
```

```

# Séparation des données d'entraînement en ensembles d'entraînement et de validation
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

# Création des DataLoaders
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)

class MLP(nn.Module):
    def __init__(self, input_size=784, hidden_size=40, output_size=10,
num_hidden_layers=4):
        super(MLP, self).__init__()
        self.layers = nn.ModuleList()
        self.layers.append(nn.Linear(input_size, hidden_size, bias=True))
        for i in range(num_hidden_layers):
            self.layers.append(nn.Linear(hidden_size, hidden_size, bias=True))
        self.layers.append(nn.Linear(hidden_size, output_size, bias=True))

    def forward(self, x):
        x = x.view(x.size(0), -1) # Mettre à plat les images
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = self.layers[-1](x)
        return x

model = MLP().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

# Entraînement du modèle
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

print('Entraînement terminé')

# Évaluation du modèle

```

```
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

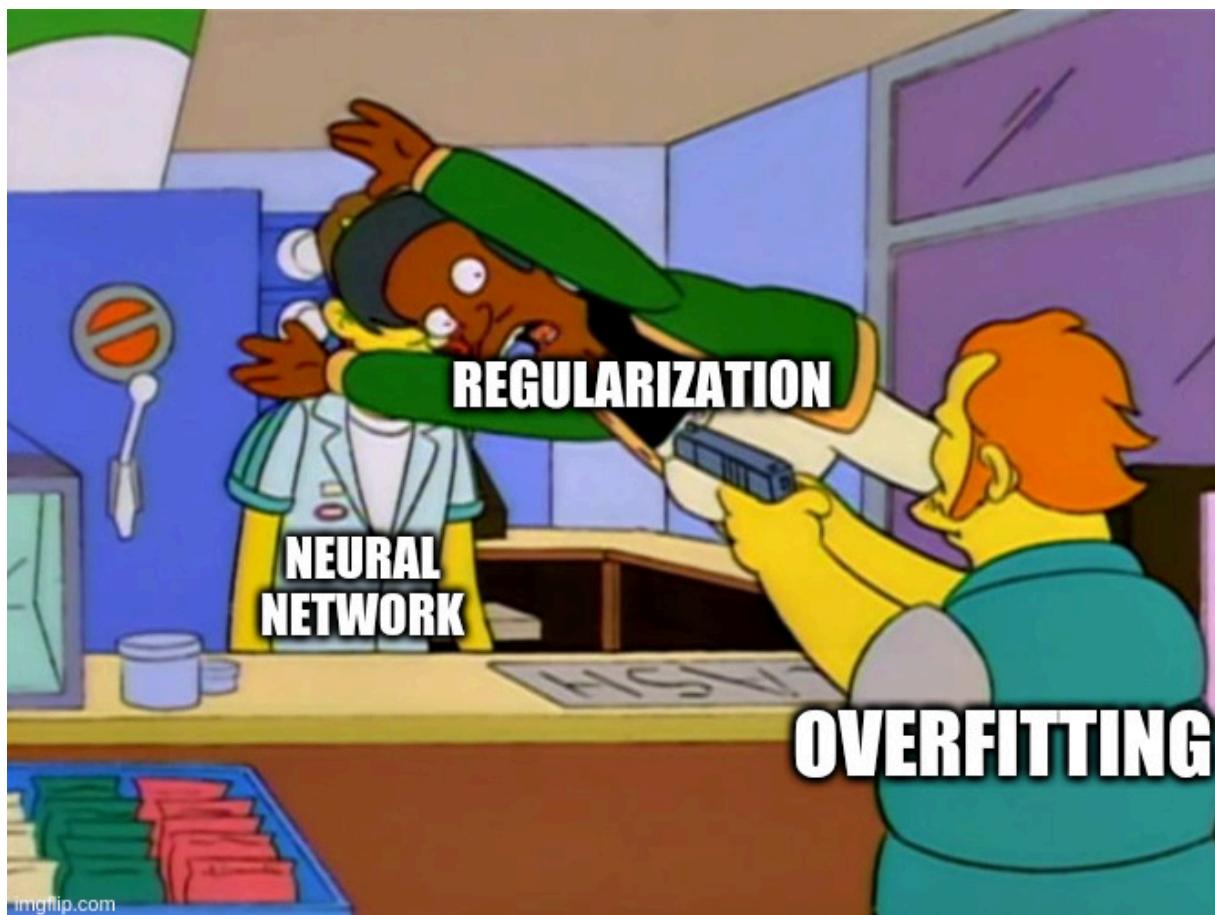
accuracy = 100 * correct / total
print(f'Précision sur l\'ensemble de test : {accuracy:.2f}%')
```

6 Régularisation

Un ingénieur en deep learning est comparable à un sculpteur d'une œuvre d'art, l'art de sculpter des modèles précis et robustes, chaque ajustement, doit être effectué avec précision, guidé par une vision claire de l'objectif final. Tout comme le sculpteur est limité par le bloc de pierre devant lui, les data scientists confrontés à la nécessité de contraindre leurs modèles pour éviter qu'ils ne deviennent trop complexes. C'est ici que la notion de régularisation, et plus précisément, la pénalité des normes des paramètres, entre en jeu. La régularisation est un outil de plus que les ingénieurs en machine learning ont pour maîtriser leur art.

6.1 Introduction à la régularisation

La régularisation est une technique extrêmement importante machine learning, statistique et deep learning pour prévenir l'overfitting d'un modèle. L'objectif d'un modèle est qu'il se généralise bien à de nouvelles données, c'est-à-dire qu'il soit capable de faire des prédictions précises sur des données qu'il n'a jamais vues lors de son entraînement. Cependant, si un modèle est trop complexe (beaucoup de paramètres) ou s'il est entraîné trop longtemps. La régularisation est là pour contrer l'overfitting et permet d'utiliser des modèles plus complexes grâce aux régularisateurs. Dans ce chapitre vous allez apprendre pourquoi sans régularisateur nos modèles de deep learning seraient limités à résoudre de simples problèmes avec une simple architecture. Cet outil, les régularisateurs, permettent de



6.1.1 Qu'est-ce que la régularisation ?

La régularisation, dans le contexte du machine learning, est une technique utilisée pour prévenir la suradaptation (ou overfitting) des modèles. L'overfitting se produit lorsque le modèle apprend « par cœur » les exemples de données d'entraînement, au lieu de généraliser à partir d'elles. En conséquence, bien que le modèle puisse avoir d'excellentes performances sur les données d'entraînement, il peut échouer lamentablement lorsqu'il est confronté à de nouvelles données, inconnues.

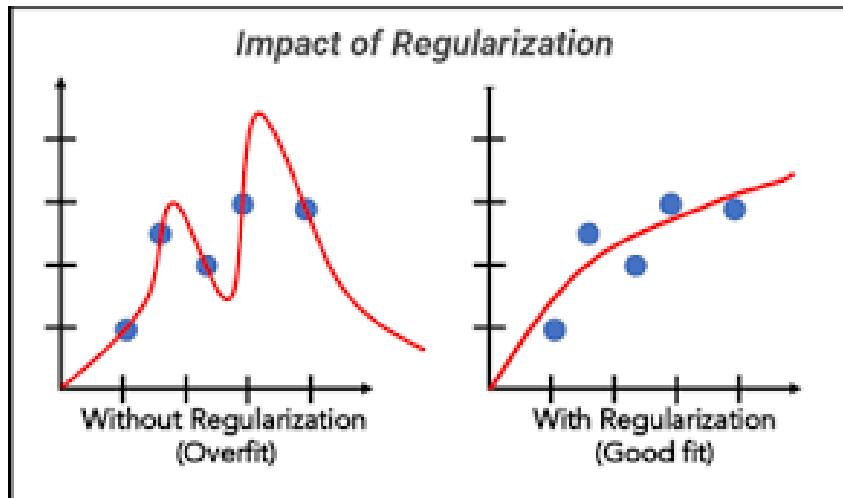


Figure 44 : Perceptron sans biais qui calcule la moyenne des entrées.

Description du diagramme: Un graphique à deux dimensions montrant la comparaison entre un modèle sous-ajusté, un modèle bien ajusté et un modèle surajusté. L'axe des x représente les données d'entrée et l'axe des y représente les prédictions du modèle. Les points réels sont dispersés de manière aléatoire, et trois courbes sont tracées : une ligne droite (sous-ajustée), une courbe qui suit de près la tendance des points (bien ajustée) et une courbe qui passe par presque tous les points individuels (surajustée).

La régularisation introduit une forme de contrainte ou de pénalité sur la complexité du modèle. Cette contrainte est souvent appliquée en ajoutant un terme à la fonction de coût, qui pénalise certaines caractéristiques du modèle, comme la magnitude des poids dans un réseau de neurones. En imposant cette contrainte, la régularisation encourage le modèle à adopter une solution plus « simple », qui est moins susceptible de surajuster aux données d'entraînement.

Il existe plusieurs formes de régularisation, mais les plus courantes dans le contexte des réseaux de neurones sont la régularisation L1 et L2. La régularisation L1 pénalise la somme absolue des poids, tandis que la régularisation L2 pénalise la somme des carrés des poids. Ces techniques peuvent être utilisées seules ou en combinaison, en fonction des besoins spécifiques du problème à résoudre.

6.1.2 Pénalités des Normes des Paramètres : Une Introduction

D'un point de vue mathématique, cette pénalité se manifeste par l'ajout d'un terme supplémentaire à notre fonction de coût. Si nous notons cette fonction de coût par $J(\theta; X, y)$, où θ représente les paramètres (poids w et biais b) du modèle, X les données d'entrée et y les labels

de données correspondantes, la régularisation introduit un nouveau terme, souvent désigné par $\Omega(\theta)$. Ce terme est directement lié à la magnitude des paramètres du modèle

La fonction coût régularisée est alors donnée par :

$$J'(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

α est un hyperparamètre, c'est-à-dire une valeur que nous choisissons avant l'entraînement du modèle. Il détermine l'importance relative de la pénalité par rapport à la fonction de coût originale.

Le rôle de α est crucial. Si α est fixé à zéro, la pénalité n'a aucun effet et nous revenons à notre fonction de coût d'origine, n'importe quoi multiplié par 0, donne 0. Si α est très grand, la pénalité dominera, et les paramètres du modèle seront fortement contraints. Dans la pratique, le choix de α est souvent déterminé par validation croisée, une technique où l'on évalue la performance du modèle sur un sous-ensemble de données non utilisé pendant l'entraînement.

Pour mieux appréhender la signification de $\Omega(\theta)$, considérons un simple modèle linéaire. Sans régularisation, ce modèle chercherait à adapter une ligne aussi près que possible de chaque point de données. Avec régularisation, la ligne sera plus « lisse », évitant les virages brusques pour s'adapter à un point de données particulier.

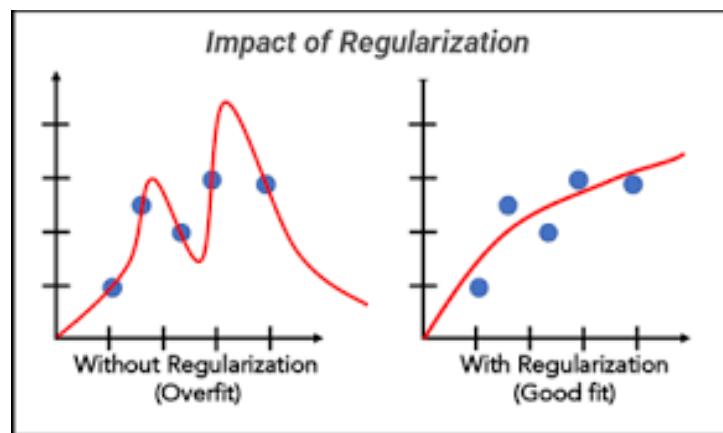


Figure 45 : ????????????????????

La nature exacte de la pénalité $\Omega(\theta)$ dépend de la forme de régularisation choisie. Les plus courantes sont les régularisations L1 et L2, qui pénalisent respectivement la valeur absolue et le carré des paramètres. Ces formes de régularisation ont des propriétés et des applications différentes, mais elles partagent toutes le même objectif fondamental: contraindre la complexité du modèle.

6.2 Techniques de régularisation

La régularisation est une méthode qui introduit une pénalité sur la complexité du modèle pour éviter l'overfitting. Dans le contexte des réseaux de neurones, cette pénalité est souvent appliquée sur les poids du réseau. Les régularisations L1 et L2 sont deux des techniques les plus populaires pour cela.

Lorsque nous parlons de régularisation dans le contexte des réseaux de neurones, nous faisons souvent référence à la pénalité des normes des paramètres. Cette approche limite la capacité

des modèles en ajoutant une pénalité, souvent notée $\Omega(\theta)$, à la fonction objective, J . La formulation mathématique de cette idée est :

$$J'(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

6.2.1 Régularisation L1 et L2 (Weight Decay)

La régularisation L1 et L2 sont deux des techniques les plus couramment utilisées pour pénaliser la magnitude des paramètres dans un modèle. Elles sont souvent appelées « Lasso » (pour L1) et « Ridge » (pour L2) dans le contexte de la régression linéaire, mais ces principes s'appliquent également aux réseaux de neurones et à d'autres modèles de machine learning.

6.2.1.1 L2 (Ridge) Régularisation

La régularisation L2, souvent appelée Ridge, ajoute une pénalité basée sur le carré des poids. Mathématiquement, cela ressemble à :

$$\Omega(\theta) = \frac{1}{2} \| w \|_2^2$$

Cette forme de régularisation est également connue sous le nom de régularisation de Tikhonov. Lorsque nous dérivons cette fonction de perte par rapport à w , nous obtenons :

$$\frac{\partial}{\partial w} J'(w; X, y) = \nabla_w J(w; X, y) + \alpha w$$

Ce qui est fascinant ici, c'est la manière dont la régularisation L2 ajuste les poids pendant la descente de gradient. Au lieu de simplement suivre la direction de la plus grande descente de la fonction de coût, chaque poids est « retrécî » vers zéro par un facteur de α . Cela a pour effet de régulariser ou de lisser les poids, évitant ainsi des valeurs extrêmes qui pourraient causer de l'overfitting.

De plus, la régularisation L2 a une interprétation bayésienne intéressante. Elle équivaut à avoir une distribution a priori gaussienne sur les poids, ce qui signifie que nous supposons initialement que les poids sont distribués normalement autour de zéro.

6.2.1.2 L1 (Lasso) Régularisation

La régularisation L1, ou Lasso, est une autre technique qui ajoute une pénalité basée sur la valeur absolue des coefficients. Elle est mathématiquement représentée par :

$$\omega(\theta) = \|w\|_1 = \sum_i |w_i|$$

Lorsque nous dérivons cette fonction de perte par rapport à w , nous obtenons :

$$\frac{\partial}{\partial w} J'(w; X, y) = \nabla_w J(w; X, y) + \alpha \text{sign}(w)$$

où $\text{sign}(w)$ est une fonction qui renvoie -1 pour les poids négatifs, 1 pour les poids positifs, et 0 pour les poids nuls. Cette propriété de la régularisation L1 est particulièrement intéressante car elle conduit à des solutions éparses. En d'autres termes, elle a tendance à produire des modèles où de nombreux poids sont exactement zéro. Cela a des implications profondes, notamment en termes de sélection de caractéristiques : les poids qui sont poussés à zéro in-

diquent essentiellement que les caractéristiques correspondantes ne sont pas utiles pour la prédiction.

la régularisation L1 est une méthode de maximisation de la robustesse. En forçant certains poids à être exactement zéro, elle réduit la sensibilité du modèle aux variations dans ces dimensions, le rendant ainsi plus robuste aux perturbations et aux bruits.

6.2.1.3 analyse mathématique approfondie

La distinction entre L1 et L2 peut également être comprise en termes de géométrie des niveaux de régularisation. Pour L2, les niveaux de régularisation forment des cercles concentriques autour de l'origine dans un espace à 2 dimensions, tandis que pour L1, ils forment des losanges. Cela signifie que la régularisation L1 est plus susceptible de toucher l'axe (c'est-à-dire de rendre un poids nul) avant la régularisation L2

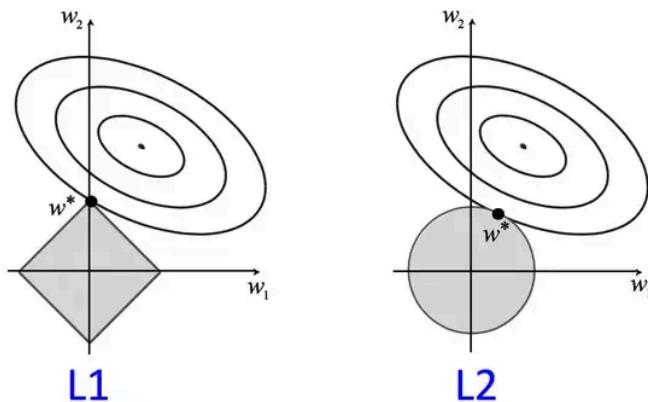


Figure 46 :

De plus, du point de vue de la distribution a priori des poids, la régularisation L2 équivaut à supposer que les poids suivent une distribution normale (ou gaussienne), tandis que la régularisation L1 équivaut à supposer une distribution de Laplace pour les poids.

Enfin, du point de vue de la solution obtenue, la régularisation L2 tend à donner des solutions où la plupart des poids sont petits, mais non nuls, tandis que la régularisation L1 donne des solutions où de nombreux poids sont exactement zéro, mais ceux qui ne le sont pas peuvent être relativement grands.

Bibliographie

- [1] H. Touvron *et al.*, « LLaMA: Open and Efficient Foundation Language Models », *arXiv preprint arXiv:2302.13971*, févr. 2023, Disponible sur: <https://arxiv.org/pdf/2302.13971.pdf>
- [2] A. Bodin et F. Recher, *Deep Math Mathématiques (simples) des réseaux de neurones (pas trop compliqués)*. Exo7, 2021. Disponible sur: <https://exo7math.github.io/deepmath-exo7/>
- [3] H. Li, Z. Xu, G. Taylor, C. Studer, et T. Goldstein, « Visualizing the Loss Landscape of Neural Nets », *arXiv preprint arXiv:1712.09913*, 2018.
- [4] D. E. Rumelhart, G. E. Hinton, et R. J. Williams, « Learning representations by back-propagating errors », *Nature*, n° 6088, p. 533–536, 1986, Disponible sur: <http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf>
- [5] X. Glorot et Y. Bengio, « Understanding the difficulty of training deep feedforward neural networks », 2010, p. 249–256. Disponible sur: [http://proceedings.mlr.press/v9/glorot10a.pdf](http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)
- [6] D.-H. Lee, S. Zhang, A. Fischer, et Y. Bengio, « Difference Target Propagation », 2015, p. 498–515. doi: 10.48550/arXiv.1412.7525.
- [7] W.-D. K. Ma, J. Lewis, et W. B. Kleijn, « The HSIC Bottleneck: Deep Learning without Back-Propagation », *arXiv preprint arXiv:1908.01580*, 2019, doi: 10.48550/arXiv.1908.01580.
- [8] A. Choromanska *et al.*, « Beyond Backprop: Online Alternating Minimization with Auxiliary Variables », *arXiv preprint arXiv:1806.09077*, 2018, doi: 10.48550/arXiv.1806.09077.
- [9] M. Jaderberg *et al.*, « Decoupled Neural Interfaces using Synthetic Gradients », *arXiv preprint arXiv:1608.05343*, 2016, doi: 10.48550/arXiv.1608.05343.
- [10] G. Hinton, « The Forward-Forward Algorithm: Some Preliminary Investigations », *arXiv preprint arXiv:2212.13345*, 2022, doi: 10.48550/arXiv.2212.13345.
- [11] Y. Bengio, D.-H. Lee, J. Bornschein, T. Mesnard, et Z. Lin, « Towards Biologically Plausible Deep Learning », *arXiv preprint arXiv:1502.04156*, 2015, doi: 10.48550/arXiv.1502.04156.