



MINISTERE DE L'ENSEIGNEMENT
SUPÉRIEUR ET DE LA RECHERCHE
SCIENTIFIQUE
UNIVERSITE TUNIS EL MANAR
Faculté des Sciences de Tunis



Apprentissage Profond

Introduction Théorique et Exemple Pratique

Réalisé par:

Chaker Maleke

IDS5

October 11, 2024

Abstract

Ce projet vise à fournir une compréhension approfondie des principes fondamentaux du apprentissage profond (Deep Learning). Nous explorons les distinctions cruciales entre l'Apprentissage Machine (machine learning) et le l' apprentissage profond, mettant l'accent sur l'évolution des approches algorithmiques.

La première section du projet se consacre à l'exploration des concepts fondamentaux des réseaux de neurones. Elle propose une analyse détaillée du fonctionnement du perceptron monocouche (Single Layer Perceptron) ainsi que du perceptron multicouche (Multi-Layer Perceptron). Cette étape jette les bases essentielles pour la compréhension des réseaux de neurones artificiels (ANN).

Dans la seconde partie, nous approfondissons le concept des réseaux de neurones artificiels (ANN), nous explorons leur relation avec les réseaux neuronaux biologiques. Nous détaillons le processus du "Forward Pass" dans un ANN, suivi d'une explication approfondie de l'algorithme de rétropropagation (Backpropagation), qui joue un rôle essentiel dans l'ajustement des poids du réseau en fonction des erreurs. En outre, nous examinons les optimiseurs tels que la Descente de Gradient (Gradient Descent) et Adam, qui jouent un rôle crucial dans l'optimisation des modèles de réseaux de neurones.

Ce projet offre ainsi une introduction complète aux concepts clés du Deep Learning, de la structure des réseaux de neurones à l'application des optimiseurs, fournissant une base solide pour toute personne cherchant à approfondir ses connaissances dans ce domaine en constante évolution.

Contents

	Page
1 Introduction L'apprentissage profond	3
1.1 Introduction	3
1.2 Perceptron Monocouche	6
1.3 Fonction d'Activation	7
1.4 Réseaux de Neurones	9
2 Les Réseaux de Neurones Artificiels	11
2.1 Introduction des Réseaux de Neurones Artificiels	11
2.2 Relation entre réseau neurone artificiel (ANN) et réseau neurone biologique	12
2.3 Processus du Forward Pass et Backpropagation	13
3 Optimisation dans les Réseaux de Neurones Artificiels	17
3.1 Descente de Gradient	17
3.2 Optimiseurs Avancés	19
3.3 Sélection d'Optimiseurs en Fonction du Problème	28
3.4 Régularisation et Optimisation	29
3.5 Hyperparamètres et Stratégies de Recherche	30
3.6 Apprentissage par Transfert	33
4 Conclusion	36

1 Introduction L'apprentissage profond

1.1 Introduction

1.1.1 Définition de L'apprentissage automatique

L'apprentissage automatique est une sous-catégorie de l'intelligence artificielle (IA) qui permet aux ordinateurs d'apprendre à partir de données et de prendre des décisions sans programmation explicite. Il englobe diverses techniques et algorithmes qui permettent aux systèmes de reconnaître des motifs, de faire des prédictions et d'améliorer leurs performances au fil du temps.

1.1.2 Définition de L'apprentissage profond

L'apprentissage profond (Deep Learning) est un type d'apprentissage automatique (machine learning) qui apprend aux ordinateurs à accomplir des tâches en s'inspirant des exemples, tout comme le font les êtres humains.

Imaginez enseigner à un ordinateur à reconnaître les chats : au lieu de lui dire de rechercher des vibrisses, des oreilles et une queue, vous lui montrez des milliers de photos de chats. L'ordinateur découvre les motifs communs tout seul et apprend à identifier un chat. C'est l'essence de l'apprentissage profond.

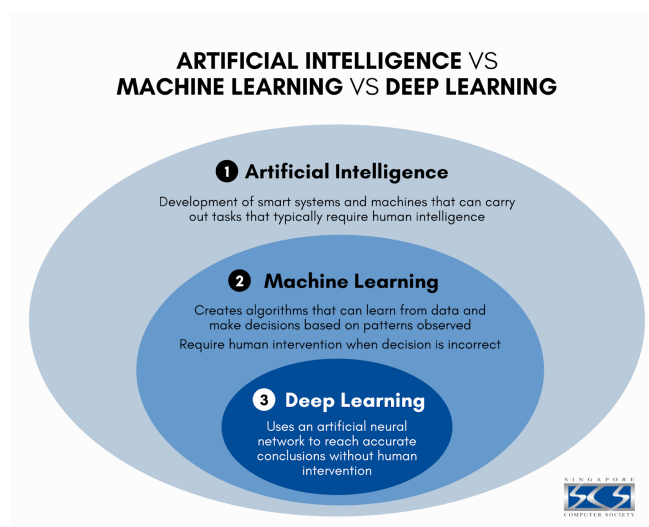


Figure 1.1: Intelligence Artificielle vs Machine Learning vs Deep Learning

1.1.3 L'importance de l'apprentissage profond

Les motivations derrière l'adoption généralisée de l'apprentissage profond dans l'industrie sont :

- **Gestion des données non structurées** : Capacité à traiter des informations complexes comme les images et vidéos, réduisant le besoin de normalisation des données.
- **Haute précision** : Les modèles d'apprentissage profond sont les plus précis en vision par ordinateur, NLP et traitement audio.
- **Reconnaissance de motifs** : Les modèles détectent automatiquement divers motifs sans intervention humaine.

1.1.4 Les applications de l'apprentissage profond

Récemment, le monde de la technologie a connu une augmentation des applications d'intelligence artificielle, toutes alimentées par des modèles d'apprentissage profond.

Voici quelques-unes des applications les plus célèbres construites à l'aide de l'apprentissage profond.

- **Vision par ordinateur** : Utilisée pour la détection d'objets dans les voitures autonomes et la reconnaissance faciale.
- **Reconnaissance vocale** : Intégrée dans les assistants vocaux comme "Google" et "Siri", pour la conversion et détection audio.
- **Intelligence artificielle générative** : Popularisée par des œuvres comme les NFTs et des modèles comme GPT-4 pour la génération de texte.



Figure 1.2: Collection de CryptoPunk

1.1.5 Émergence récente du deep learning

La montée en puissance du deep learning est due à la convergence de données abondantes et d'une puissance de calcul accrue.

Bien que les méthodes d'apprentissage machine existent depuis les années 60, avec des travaux fondateurs comme ceux de Geoffrey Hinton en 1986, leur popularité a récemment explosé grâce à l'immense disponibilité de données et à des capacités de calcul bien supérieures.

1. Les algorithmes traditionnels, comme la régression linéaire ou les SVM, cessent généralement de s'améliorer à un niveau spécifique, même si nous alimentons l'algorithme avec davantage de données.
2. En revanche, en formant un petit réseau neuronal avec la même quantité de données, vous pourriez obtenir de meilleures performances.
3. Si nous formons un réseau neuronal plus grand, nous pouvons obtenir des performances bien meilleures.

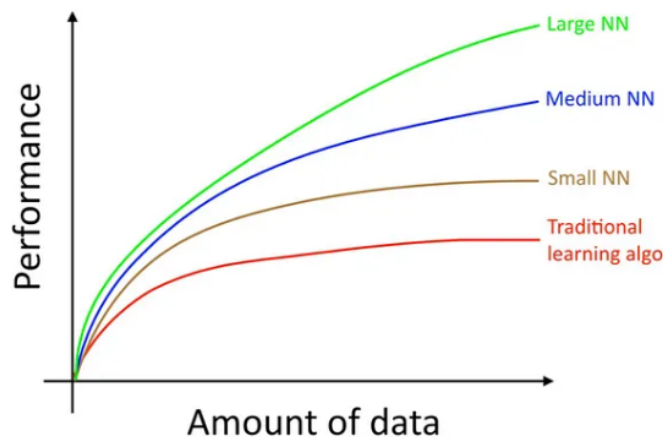


Figure 1.3: Algorithme traditionnel VS réseau neuronal en augmentant la quantité des données

Ainsi, si vous souhaitez atteindre ce très haut niveau de performance, vous avez besoin de deux éléments :

- Souvent, vous devez être capable d'entraîner un réseau neuronal suffisamment grand afin de tirer parti de l'énorme quantité de données.
- Vous avez besoin de beaucoup de données.

Cependant, cela fonctionne seulement jusqu'à un certain point, car finalement, vous pouvez épuiser vos données ou votre réseau neuronal devient si grand qu'il met trop de temps à être entraîné.

1.2 Perceptron Monocouche

Le perceptron monocouche est le modèle le plus simple de réseau de neurones artificiels. Il consiste en une seule couche de neurones.

Le graphe de calcul du réseau neuronal monocouche implique plusieurs étapes et composants clés:

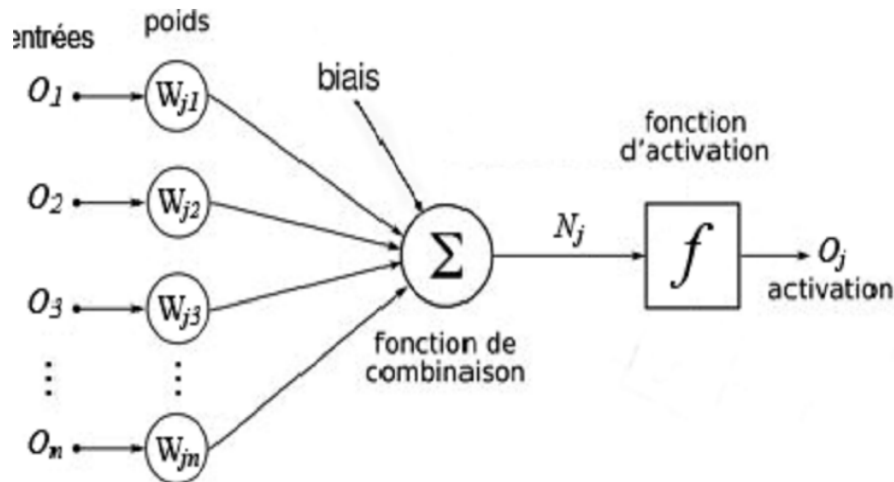


Figure 1.4: Le graphe de calcul du réseau neuronal monocouche

1. Couche d'Entrée: représente les caractéristiques d'entrée O_1, O_2, \dots, O_n
2. Somme Pondérée: Chaque caractéristique d'entrée est multipliée par son poids correspondant ($W_{j1}, W_{j2}, \dots, W_{jn}$) et puis calcule de la somme pondérée $Z = W_1X_1 + W_2X_2 + \dots + W_nX_n$
3. La fonction d'activation (la fonction de transfert): joue un rôle très important dans le comportement du neurone. Elle retourne une valeur représentative de l'activation du neurone, cette fonction a comme paramètre la somme pondérée des entrées ainsi que le seuil d'activation.
Exemple, La fonction sigmoïde est une fonction d'activation appliquée à la somme pondérée pour écraser la sortie entre 0 et 1, La fonction exponentielle...
4. La sortie: est le résultat de la fonction d'activation .

Exercice : Perceptron Monocouche

Soit un perceptron simple avec deux entrées x_1 et x_2 , et un poids associé à chaque entrée w_1 et w_2 . La sortie du perceptron est donnée par :

$$y = \text{step}(w_1x_1 + w_2x_2 + b)$$

où b est le biais,

et $\text{step}(z) = 1$ si $z \geq 0$,

sinon $\text{step}(z) = 0$.

Avec les poids $w_1 = 1$, $w_2 = -1$, et $b = 0$, calculez la sortie du perceptron pour les entrées $(x_1, x_2) = (1, 1)$, $(0, 1)$, $(1, 0)$, et $(0, 0)$.

Correction

- Pour $(x_1, x_2) = (1, 1)$:

$$y = \text{step}(1 \times 1 + (-1) \times 1 + 0) = \text{step}(0) = 1$$

- Pour $(x_1, x_2) = (0, 1)$:

$$y = \text{step}(1 \times 0 + (-1) \times 1 + 0) = \text{step}(-1) = 0$$

- Pour $(x_1, x_2) = (1, 0)$:

$$y = \text{step}(1 \times 1 + (-1) \times 0 + 0) = \text{step}(1) = 1$$

- Pour $(x_1, x_2) = (0, 0)$:

$$y = \text{step}(1 \times 0 + (-1) \times 0 + 0) = \text{step}(0) = 1$$

1.3 Fonction d'Activation

Dans les réseaux de neurones, la fonction d'activation est essentielle pour introduire de la non-linéarité, permettant ainsi au modèle d'apprendre des relations complexes dans les données. Elle décide si un neurone doit s'activer en fonction de l'entrée reçue, ce qui influence le comportement global du réseau.

1.3.1 Importance des Fonctions d'Activation

Les fonctions d'activation jouent un rôle fondamental dans les réseaux de neurones pour plusieurs raisons :

- **Non-linéarité** : Elles permettent au réseau de capturer des relations non-linéaires, essentielles pour résoudre des problèmes complexes comme la reconnaissance d'images ou le traitement du langage naturel.

- **Modulation de l'apprentissage** : En filtrant certaines entrées ou en limitant la sortie dans une plage spécifique (comme avec la fonction sigmoïde ou tanh), elles aident à stabiliser l'apprentissage et à éviter la saturation des gradients.
- **Différenciabilité** : Une bonne fonction d'activation doit être différentiable, ce qui permet aux réseaux d'utiliser la rétropropagation pour ajuster les poids durant l'apprentissage.

1.3.2 Types de Fonctions d'Activation

Les fonctions d'activation les plus couramment utilisées sont :

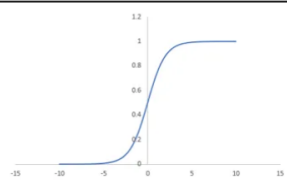
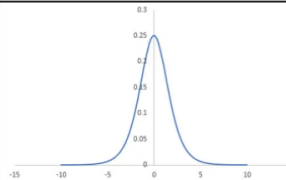
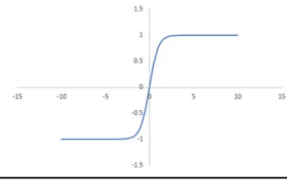
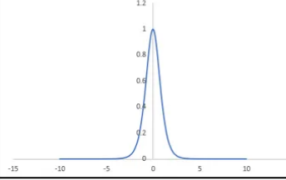
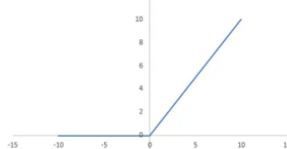
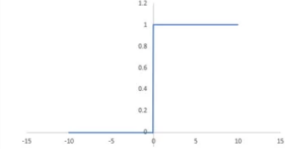
Activation function	$f(x)$	$f(x)$	$\frac{df(x)}{dx}$
sigmoid	$\frac{1}{1 + e^{-x}}$		
tanh	$\tanh(x)$		
ReLU	$\max(0, x)$		

Figure 1.5: Fonctions d'activation courantes

1. **ReLU (Rectified Linear Unit)** : La plus utilisée en deep learning, elle transmet les valeurs positives et bloque les négatives, aidant le modèle à se concentrer sur les caractéristiques importantes des données.
2. **Sigmoïde** : Utilisée en sortie pour la classification binaire, elle produit une valeur entre 0 et 1, interprétée comme une probabilité.
3. **tanh** : Normalise les entrées entre -1 et 1, souvent utilisée en alternative à la Sigmoïde pour les tâches de classification binaire.

1.3.3 Comment Choisir la Fonction d'Activation Adéquate

Le choix de la fonction d'activation dépend de plusieurs facteurs, notamment le type de problème à résoudre et les caractéristiques du réseau :

- **Problème de classification binaire** : La fonction Sigmoid est idéale pour les sorties binaires car elle produit une probabilité entre 0 et 1.
- **Problème de classification multi-classes** : La fonction Softmax est souvent utilisée en dernière couche pour des problèmes avec plusieurs classes, car elle produit une distribution de probabilités sur plusieurs classes.
- **Profondeur du réseau** : Pour les réseaux profonds, ReLU est généralement privilégiée en raison de sa simplicité et de son efficacité à atténuer les problèmes de vanishing gradients.
- **Tâches spécifiques** : Pour les réseaux récurrents ou les modèles nécessitant une régularisation, des fonctions comme tanh ou des variantes de ReLU (comme le Leaky ReLU ou ELU) peuvent être utilisées pour améliorer la performance.

Le choix de la fonction d'activation doit également être fait en fonction des performances observées pendant la phase de validation et du comportement du modèle lors de l'apprentissage.

1.3.4 Exercice : Fonction d'Activation

Soit une fonction d'activation sigmoïde définie par :

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

Calculez $\sigma(z)$ pour $z = -1$, $z = 0$, et $z = 2$.

1.3.5 Correction

- Pour $z = -1$:

$$\sigma(-1) = \frac{1}{1 + e^1} = \frac{1}{1 + 2.718} \approx 0.2689$$

- Pour $z = 0$:

$$\sigma(0) = \frac{1}{1 + e^0} = \frac{1}{2} = 0.5$$

- Pour $z = 2$:

$$\sigma(2) = \frac{1}{1 + e^{-2}} = \frac{1}{1 + 0.1353} \approx 0.8808$$

1.4 Réseaux de Neurones

Les réseaux de neurones artificiels (RNA) sont essentiels au deep learning, imitant le fonctionnement du cerveau humain et utilisés dans divers domaines tels que la reconnaissance d'images et la traduction.

Il revient essentiellement à prendre le réseau neuronal monocouche et à la répéter au moins deux fois.

En réseau neuronal monocouche, il y a la couche d'entrée et de sortie. Cependant, dans un réseau neuronal, il y a au moins une couche cachée entre la couche d'entrée et la couche de sortie.

Un réseau de neurones est organisé en couches :

1. **Couche d'entrée** : reçoit les données brutes.
2. **Couches cachées** : effectuent des transformations complexes pour extraire des caractéristiques abstraites.
3. **Couche de sortie** : produit les résultats, adaptés à la tâche (par exemple, classification binaire ou multi-classes).

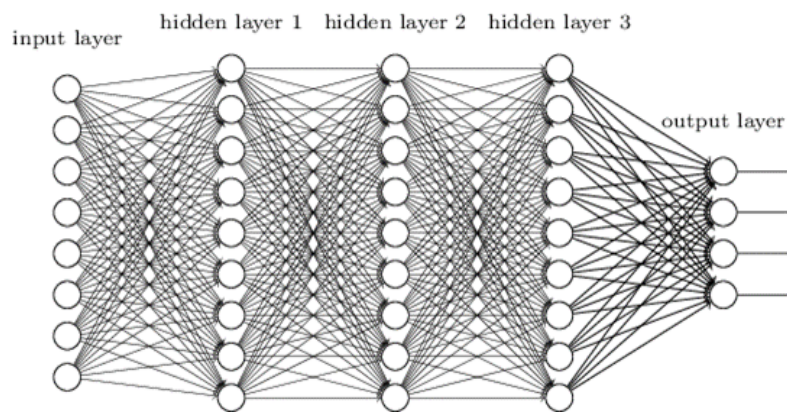


Figure 1.6: Architecture des Réseaux de Neurones

1.4.1 Exercice : Réseaux de Neurones

Dans un réseau de neurones à deux couches, la première couche a une sortie $h_1 = \sigma(w_1x + b_1)$, et la deuxième couche produit la sortie finale $y = \sigma(w_2h_1 + b_2)$,

où σ est la fonction sigmoïde. Avec $w_1 = 0.5$, $b_1 = -1$, $w_2 = 1$, $b_2 = 0$, et $x = 2$, calculez la sortie du réseau y .

1.4.2 Correction

- Première couche :

$$h_1 = \sigma(0.5 \times 2 - 1) = \sigma(1 - 1) = \sigma(0) = 0.5$$

- Deuxième couche :

$$y = \sigma(1 \times 0.5 + 0) = \sigma(0.5) \approx 0.6225$$

2 Les Réseaux de Neurones Artificiels

2.1 Introduction des Réseaux de Neurones Artificiels

Les Réseaux de Neurones Artificiels (RNA) constituent un élément fondamental du machine learning, s'inspirant des réseaux neuronaux biologiques. Les RNA, composés de neurones interconnectés, excellent dans l'apprentissage de motifs complexes à partir des données.

Les Composants Neuronaux sont:

1. **Neurones** : Traitent les entrées par des combinaisons pondérées et appliquent des fonctions d'activation pour produire des sorties.
2. **Structure du Réseau** : Organisée en couches - entrée, cachée, et sortie - facilitant l'apprentissage de représentations complexes.

2.1.1 Types de Réseaux de Neurones Artificiels

Les réseaux de neurones artificiels peuvent être classés en plusieurs types, chacun adapté à des cas d'utilisation spécifiques. Les types les plus courants incluent :

1. **Réseaux de Neurones à Propagation Avant** (*Feedforward Neural Networks* ou FNN) :

Ce type de réseau, aussi appelé Perceptron Multi-Couches (PMC), est celui que nous avons principalement étudié dans ce rapport. Il est composé d'une couche d'entrée, d'une ou plusieurs couches cachées, et d'une couche de sortie.

Contrairement aux perceptrons simples, les neurones de ces réseaux utilisent des fonctions d'activation sigmoïdes pour traiter des problèmes non-linéaires complexes. Ces réseaux sont largement utilisés dans des domaines comme la vision par ordinateur et le traitement du langage naturel.

2. **Réseaux Neuronaux Convolutifs** (*Convolutional Neural Networks* ou CNN) :

Ces réseaux, très proches des réseaux à propagation avant, sont spécifiquement conçus pour des tâches comme la reconnaissance d'images, la détection de motifs ou la vision par ordinateur.

Les CNN utilisent des opérations de convolution qui exploitent les propriétés des matrices pour identifier des motifs dans les images, facilitant ainsi la détection de caractéristiques importantes.

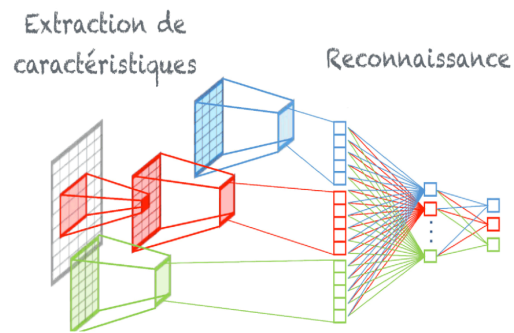


Figure 2.1: Exemple de Réseau Neuronal Convolutif

3. Réseaux de Neurones Récurrents (*Recurrent Neural Networks* ou RNN) :

Ce type de réseau est caractérisé par ses boucles de rétroaction, permettant aux informations d'être réutilisées dans le temps. Les RNN sont principalement utilisés pour traiter des données séquentielles, comme les séries temporelles, et sont souvent employés dans des domaines tels que la prédiction boursière ou les prévisions de ventes.

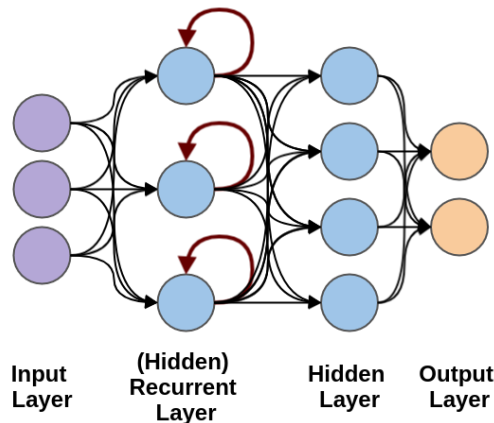


Figure 2.2: Exemple de Réseau Neuronal Récurrent

2.2 Relation entre réseau neurone artificiel (ANN) et réseau neurone biologique

L'impulsion reçue par les dendrites dans le BNN correspond à la couche d'entrée dans l'ANN. L'impulsion passe à travers les axones dans le BNN comme le processus des couches d'entrée aux couches de sortie à travers les couches cachées dans l'ANN. Enfin, la transmission des terminaisons synaptiques vers les autres dendrites dans le BNN correspond à la couche de sortie dans l'ANN.

Cependant, année après année, l'ANN est devenu beaucoup plus distinct du "vrai" réseau neuronal. Néanmoins, nous savons qu'il existe encore de nombreuses possibilités de nouvelles inventions qui

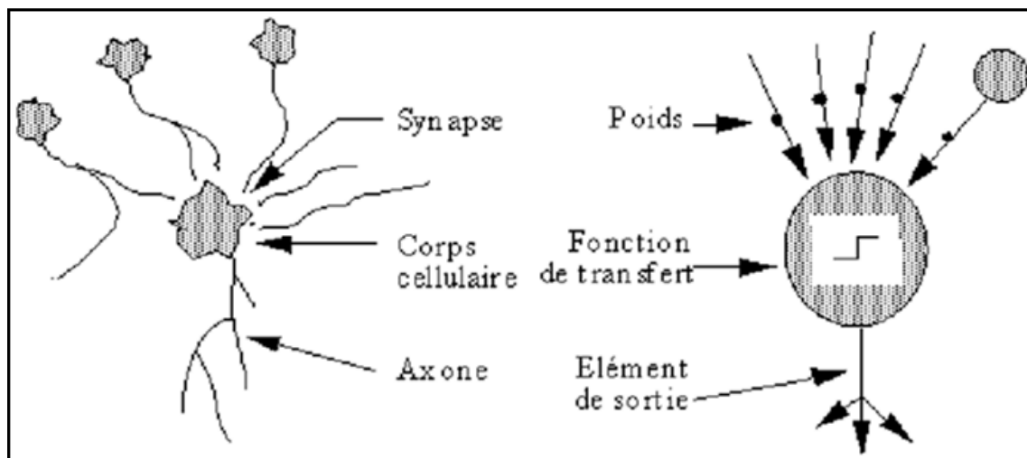


Figure 2.3: Comparaison entre le neurone biologique et le neurone formel.

peuvent être inspirées par la nature.

2.3 Processus du Forward Pass et Backpropagation

Le fonctionnement interne d'un réseau neuronal est orchestré à travers deux phases cruciales : la passe avant (forward pass) et la rétropropagation (backpropagation).

La forward pass, ou étape de propagation avant, représente le processus initial où le réseau évalue ses entrées pour générer des prédictions.

À la suite de cette phase, la rétropropagation entre en jeu, marquant une étape décisive dans laquelle le réseau ajuste ses paramètres internes pour minimiser les écarts entre les prédictions et les sorties réelles.

Explorons de manière approfondie ces deux aspects fondamentaux du fonctionnement des réseaux neuronaux : la façon dont la passe avant permet la génération de prédictions, et comment la rétropropagation joue un rôle essentiel dans l'ajustement itératif des paramètres pour améliorer la performance globale du réseau.

2.3.1 Forward Pass ou Forward Propagation

Le processus de forward propagation consiste à faire passer les données d'entrée à travers le réseau de neurones pour obtenir une prédiction. Il suit plusieurs étapes :

1. **Entrée :** Les données sont introduites dans le réseau. Chaque entrée x_i est associée à un poids w_i , et ce pour chaque neurone.
2. **Propagation à travers les couches :**

- **Combinaison linéaire** : Pour chaque neurone, les entrées x_i sont pondérées par les poids w_i , et une somme pondérée est calculée :

$$z = \sum_{i=1}^n w_i x_i + b$$

où z est la combinaison linéaire, b est le biais, et n est le nombre d'entrées.

- **Activation** : Une fonction d'activation f (comme la fonction sigmoïde ou ReLU) est appliquée à z pour introduire de la non-linéarité dans le modèle :

$$a = f(z)$$

où a est la sortie activée du neurone.

3. **Sortie** : Ce processus de combinaison linéaire et d'activation est répété à travers toutes les couches jusqu'à atteindre la couche de sortie. La sortie de la dernière couche représente la prédiction du réseau.

2.3.2 Exercice : Forward Pass

Dans un réseau de neurones simple à une couche, la sortie $y_{\text{prédit}}$ est donnée par :

$$y_{\text{prédit}} = \sigma(w_1 x + b_1)$$

où σ est la fonction sigmoïde, $w_1 = 0.5$, $b_1 = -1$, et $x = 2$.

Calculez la sortie $y_{\text{prédit}}$ après le forward pass.

2.3.3 Correction

La fonction sigmoïde est définie par :

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

Calculons $z = w_1 \cdot x + b_1$:

$$z = 0.5 \times 2 - 1 = 1 - 1 = 0$$

Ensuite, appliquons la fonction sigmoïde :

$$y_{\text{prédit}} = \sigma(0) = \frac{1}{1 + e^0} = \frac{1}{2} = 0.5$$

La sortie du réseau est donc $y_{\text{prédit}} = 0.5$.

2.3.4 Backward Pass ou Backpropagation

Le processus de backpropagation consiste à ajuster les poids du réseau de neurones pour minimiser l'erreur de prédiction. Voici les étapes détaillées avec les formules mathématiques associées :

1. Calcul du coût :

- **Comparaison avec la vérité terrain :** La sortie du réseau \hat{y} est comparée à la vérité terrain y à l'aide d'une fonction de coût. Par exemple, pour la perte quadratique (ou erreur quadratique moyenne) :

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

où J est la fonction de coût.

2. Rétropropagation de l'erreur :

- **Calcul du gradient :** Le gradient de la fonction de coût J par rapport à chaque poids w_i est déterminé en utilisant la règle de la chaîne. Par exemple, le gradient du coût J par rapport à un poids w_i est calculé ainsi :

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

où z est la combinaison linéaire des entrées pondérées. Cela mesure la sensibilité de la fonction de coût aux changements dans chaque poids.

- **Rétropropagation des gradients :** Après avoir calculé le gradient à la sortie du réseau, les gradients sont propagés à travers les couches du réseau en sens inverse, depuis la dernière couche jusqu'à la première. Pour une couche donnée l , le gradient des poids w_l est calculé en utilisant :

$$\frac{\partial J}{\partial w_l} = \delta_l \cdot a_{l-1}^T$$

où :

- $\delta_l = \frac{\partial J}{\partial z_l}$, représente l'erreur à la couche l ,
- a_{l-1} est l'activation de la couche précédente $l - 1$.

Cette équation exprime la manière dont les poids w_l sont ajustés en fonction de l'erreur observée dans la couche actuelle et des activations de la couche précédente.

• Mise à jour des poids :

- **Optimisation :** Les poids w_i du réseau sont mis à jour en utilisant un algorithme d'optimisation tel que la descente de gradient. La règle de mise à jour des poids est la suivante :

$$w_i := w_i - \eta \frac{\partial J}{\partial w_i}$$

où η est le taux d'apprentissage, et $\frac{\partial J}{\partial w_i}$ est le gradient du coût par rapport au poids w_i .

- **Répétition :**

- Les étapes du forward pass et de la backpropagation sont répétées sur plusieurs itérations (appelées epochs) jusqu'à ce que le modèle converge vers une solution optimale où le coût est minimisé.

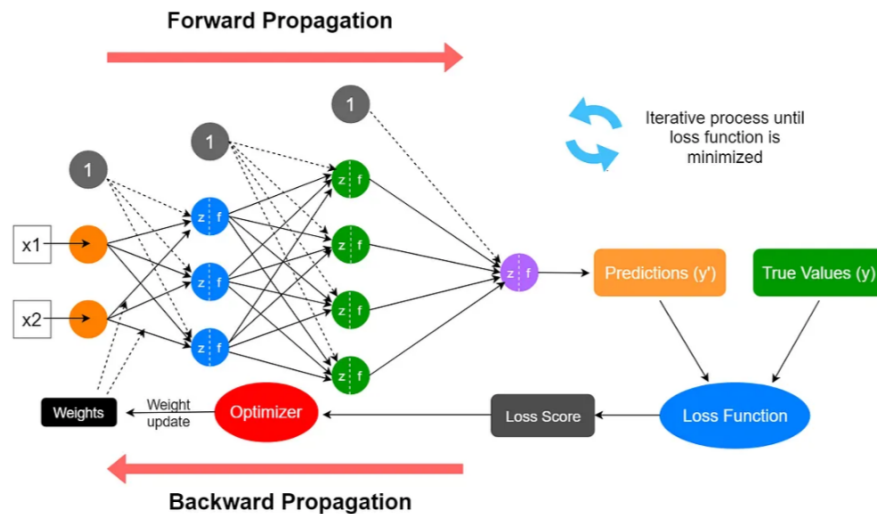


Figure 2.4: Forward Propagation et Backward Propagation

2.3.5 Exercice : Backpropagation

Soit une fonction de coût quadratique simple :

$$C = \frac{1}{2}(y_{\text{prédit}} - y_{\text{réel}})^2$$

où $y_{\text{prédit}} = 0.5$ et $y_{\text{réel}} = 1$.

Calculez la dérivée de la fonction de coût par rapport à $y_{\text{prédit}}$, c'est-à-dire $\frac{\partial C}{\partial y_{\text{prédit}}}$, pour l'étape de rétropropagation.

2.3.6 Correction

La fonction de coût est :

$$C = \frac{1}{2}(0.5 - 1)^2 = \frac{1}{2}(-0.5)^2 = \frac{1}{2} \times 0.25 = 0.125$$

La dérivée de la fonction de coût par rapport à $y_{\text{prédit}}$ est :

$$\frac{\partial C}{\partial y_{\text{prédit}}} = y_{\text{prédit}} - y_{\text{réel}} = 0.5 - 1 = -0.5$$

3 Optimisation dans les Réseaux de Neurones Artificiels

3.1 Descente de Gradient

La descente de gradient (GD) est un algorithme d'optimisation itératif de premier ordre, utilisé pour trouver un minimum ou un maximum d'une fonction donnée. Cet algorithme est largement utilisé en apprentissage machine (ML) et en apprentissage profond (DL) pour minimiser une fonction de coût ou de perte.

Principe de l'algorithme

L'algorithme de descente de gradient itère de manière répétée pour ajuster les poids en fonction du gradient de la fonction de coût. Mathématiquement, cela peut être exprimé par la formule suivante :

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

où :

- w représente les poids actuels du modèle,
- α est le taux d'apprentissage (learning rate),
- $\frac{\partial J}{\partial w}$ est le gradient de la fonction de coût J par rapport aux poids w .

La soustraction du gradient permet de minimiser la fonction de coût J , car le gradient pointe dans la direction d'augmentation de la fonction.

Importance du taux d'apprentissage

Le taux d'apprentissage α joue un rôle crucial dans la convergence de l'algorithme :

- Un α trop petit entraîne une convergence lente et peut ne pas atteindre le point optimal avant le nombre maximal d'itérations.

- Un α trop grand peut provoquer des oscillations ou même empêcher la convergence, en « sautant » autour du point optimal.

Le choix du taux d'apprentissage dépend du problème spécifique et nécessite souvent des expérimentations.

Étapes de l'algorithme

Voici les étapes principales de la méthode de descente de gradient :

- Initialisation** : Choisir un point de départ pour les poids w (généralement de façon aléatoire).
- Calcul du gradient** : Calculer le gradient de la fonction de coût par rapport aux poids à la position actuelle.
- Mise à jour des poids** : Ajuster les poids en fonction du gradient avec la formule :

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

- Répétition** : Répéter les étapes 2 et 3 jusqu'à ce qu'un critère d'arrêt soit atteint :

- Le nombre maximal d'itérations est atteint.
- La taille du pas (mise à jour des poids) devient inférieure à un seuil de tolérance (petit gradient ou petit taux d'apprentissage).

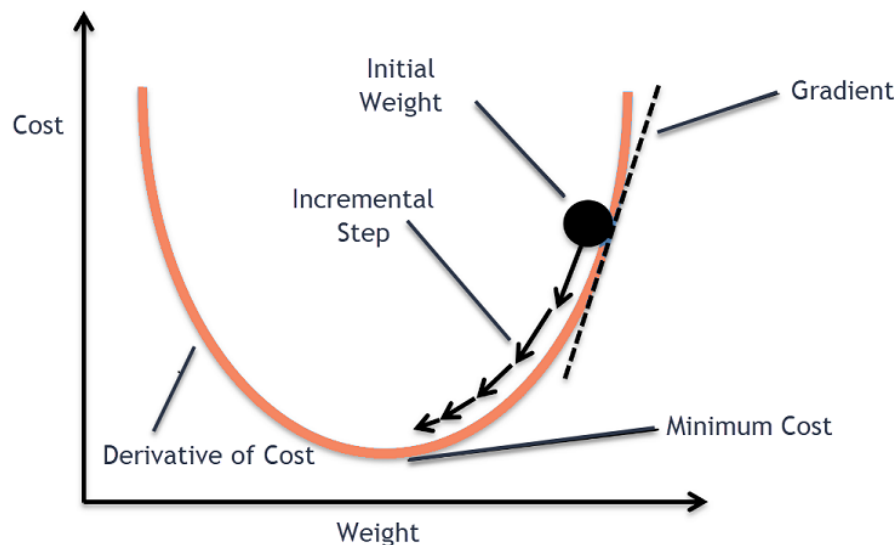


Figure 3.1: Descente de gradient

3.1.1 Exercice : Descente de Gradient

Soit une fonction de coût quadratique simple $C(w) = (w - 3)^2$.

Utilisez la méthode de la descente de gradient avec un taux d'apprentissage de $\alpha = 0.1$ pour mettre à jour le poids w , en partant de $w_0 = 0$.

Calculez la nouvelle valeur de w après une itération.

3.1.2 Correction

La dérivée de la fonction de coût est :

$$\frac{dC}{dw} = 2(w - 3)$$

En partant de $w_0 = 0$, on a :

$$\frac{dC}{dw} = 2(0 - 3) = -6$$

La mise à jour de w est donnée par :

$$w_{\text{nouveau}} = w_0 - \alpha \frac{dC}{dw} = 0 - 0.1 \times (-6) = 0.6$$

3.2 Optimiseurs Avancés

Nous avons précédemment abordé les principes fondamentaux de l'optimisation des réseaux neuronaux via la descente de gradient. Nous allons maintenant introduire plusieurs optimiseurs avancés, en exposant leurs concepts ainsi que leurs avantages et inconvénients, sans entrer dans les détails mathématiques complexes.

3.2.1 Descente de Gradient

Le concept de la descente de gradient repose sur la détermination d'un vecteur directionnel \mathbf{d} à partir du gradient (vecteur des dérivées partielles). Les paramètres sont ensuite mis à jour selon la règle suivante :

$$w = w - \alpha \cdot \mathbf{d}$$

où α représente le taux d'apprentissage. Ce processus est répété jusqu'à convergence.

L'intérêt principal réside dans le fait que le gradient s'annule dans les points critiques (minima, maxima ou points selles).

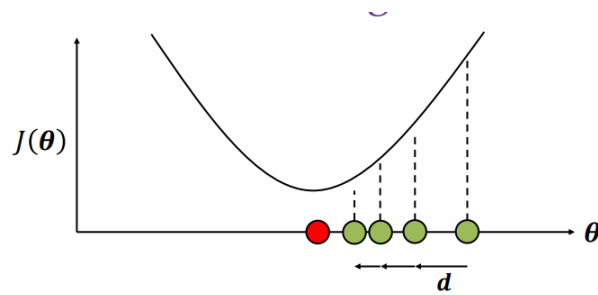


Figure 3.2: Descente de gradient

Limites de la Descente de Gradient par Batch

L'un des inconvénients majeurs de la descente de gradient en batch est **sa sensibilité aux minima ou maxima locaux**, ce qui peut rendre l'optimisation inefficace dans certains cas pratiques.

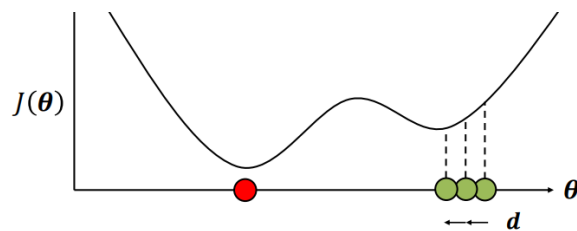


Figure 3.3: Inconvénient de la descente de gradient

En outre, cette méthode peut s'avérer **lente sur de grands ensembles de données**, car elle nécessite de calculer le gradient sur l'intégralité du lot à chaque mise à jour.

3.2.2 Descente de Gradient Stochastique (SGD)

La descente de gradient stochastique (SGD) est une technique d'optimisation largement utilisée pour entraîner des modèles de deep learning. Contrairement à la descente de gradient par batch qui calcule le gradient en utilisant l'ensemble des données d'entraînement, la SGD met à jour les paramètres après **chaque exemple d'entraînement**.

Mathématiquement, la mise à jour des poids se fait selon la formule suivante :

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t; x^{(i)}, y^{(i)})$$

où : - θ_t représente les paramètres du modèle à l'itération t , - η est le taux d'apprentissage, - $\nabla J(\theta_t; x^{(i)}, y^{(i)})$ est le gradient de la fonction de coût J par rapport aux paramètres θ calculé pour l'exemple i .

Inconvénients de la SGD

L'inconvénient majeur de la SGD réside dans **sa variabilité élevée**, car chaque exemple d'entraînement contribue à une mise à jour distincte. Cela peut entraîner une trajectoire de convergence moins stable par rapport à la descente de gradient par batch, qui bénéficie d'une estimation plus précise du gradient.

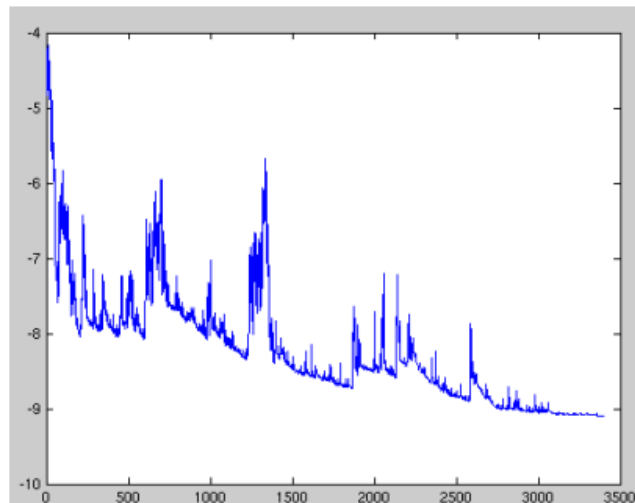


Figure 3.4: Inconvénient de la Descente du gradient stochastique

3.2.3 Descente de Gradient avec Momentum

La Descente de Gradient avec Momentum est une amélioration de la Stochastic Gradient Descent (SGD) qui intègre un terme de momentum, permettant ainsi d'accélérer la convergence en tenant compte des directions et vitesses des mises à jour antérieures.

L'idée fondamentale, illustrée par la métaphore de l'alpiniste, est que **si l'on est constamment dirigé dans une même direction, il est logique de prendre des pas plus grands**. Cela reflète le comportement d'une balle qui acquiert de l'élan en descendant une pente.

Mathématiquement, la mise à jour avec momentum s'exprime par les équations suivantes :

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla J(\theta_t) \quad (3.1)$$

$$\theta_{t+1} = \theta_t - \eta v_t \quad (3.2)$$

où v_t représente la vitesse, β est le coefficient de momentum, et $\nabla J(\theta_t)$ est le gradient de la fonction de coût.

Cette approche permet d'intégrer non seulement le gradient actuel, mais aussi ceux des mises à jour précédentes, bien que leur impact diminue à chaque étape par un facteur de γ . Ainsi, l'amplitude de la mise à jour augmente dans les régions douces, favorisant une convergence plus efficace.

Avantages de la Méthode avec Momentum

L'utilisation du momentum permet d'atténuer les oscillations indésirables et d'accélérer la convergence vers le minimum global. Cependant, il peut également survoler des minima locaux, entraînant un risque de ne pas converger vers la solution optimale.

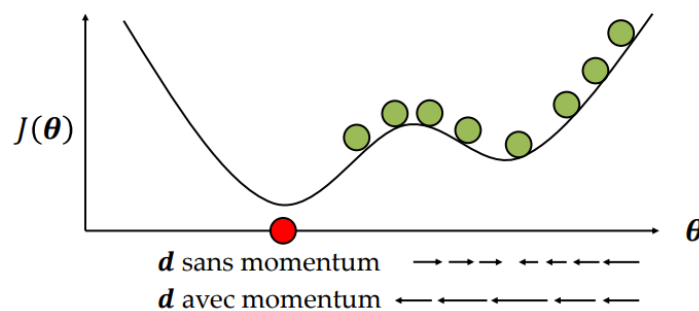


Figure 3.5: Descente du gradient sans momentum Vs Descente du gradient avec momentum

Inconvénients de la Méthode avec Momentum

Malgré ses avantages, la méthode de Descente de Gradient avec Momentum présente certains inconvénients. En particulier, elle peut survoler des minima locaux, ce qui signifie qu' **elle pourrait ne pas converger vers la solution optimale**. De plus, la performance de cette méthode est fortement dépendante du choix du coefficient de momentum β et du taux d'apprentissage η . Un mauvais choix de ces hyperparamètres peut entraîner des oscillations persistantes ou une divergence du processus d'optimisation.

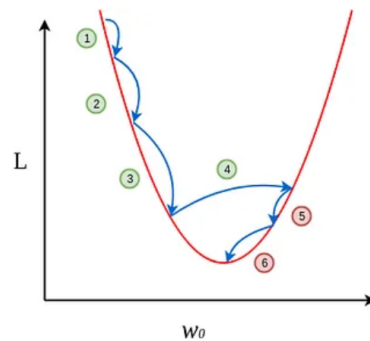


Figure 3.6: Inconvénient de Descente du gradient avec momentum

3.2.4 Descente de Gradient avec Momentum Accéléré de Nesterov

La méthode de Nesterov Accelerated Gradient (NAG) améliore la descente de gradient avec momentum **en anticipant le mouvement du gradient**, ce qui permet d'optimiser les mises à jour et de réduire les risques de dépassements (overshoots).

NAG se base sur une règle de mise à jour en deux étapes :

- (a) Une mise à jour partielle pour se rendre à un point anticipé,
- (b) Suivie d'une mise à jour finale qui utilise le gradient à ce point.

Cette approche aide à ajuster la direction des mises à jour et à **diminuer les oscillations indésirables**.

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla J(\theta_t - \eta \beta v_{t-1}) \quad (3.3)$$

$$\theta_{t+1} = \theta_t - \eta v_t \quad (3.4)$$

NAG présente des avantages significatifs en termes de vitesse de convergence, facilitant la navigation vers les minima globaux, comme le montre la figure 3.12.

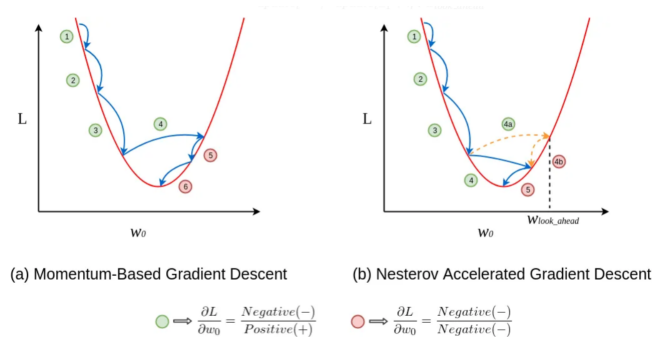


Figure 3.7: Descente de Gradient avec Momentum VS Descente de Gradient Accéléré de Nesterov

Inconvénients et Utilisation

Cependant, NAG est **sensible aux hyperparamètres**, tels que le taux d'apprentissage et le coefficient de momentum β . Un mauvais réglage de ces paramètres peut entraîner des oscillations ou une divergence.

Malgré ces inconvénients, NAG reste une méthode efficace pour des situations nécessitant une convergence rapide.

3.2.5 Adagrad (Adaptive Gradient Algorithm)

Adagrad est un algorithme d'optimisation qui ajuste le taux d'apprentissage de chaque paramètre en fonction de la fréquence d'apparition des gradients.

Sa mise à jour est donnée par :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla J(\theta_t)$$

où G_t est la somme accumulée des carrés des gradients.

Avantages d'Adagrad

Adagrad ajuste automatiquement le taux d'apprentissage pour chaque paramètre, **favorisant les caractéristiques rarement mises à jour** (avec des gradients moins fréquents) et réduisant le risque de surapprentissage des caractéristiques fréquentes.

Inconvénients d'Adagrad

À mesure que les gradients s'accumulent, le taux d'apprentissage diminue continuellement, ce qui peut rendre l'algorithme trop conservateur dans les dernières étapes de l'entraînement, **ralentissant la convergence**.

Dans certains cas, il peut être nécessaire d'introduire des techniques de régularisation supplémentaires pour contrer les effets de cette diminution excessive du taux d'apprentissage.

3.2.6 RMSprop (Root Mean Square Propagation)

RMSprop est une variante d'Adagrad qui surmonte la limitation liée à la diminution rapide du taux d'apprentissage. Contrairement à Adagrad, qui accumule tous les gradients passés, RMSprop utilise une moyenne mobile exponentielle du carré des gradients pour **normaliser les taux d'apprentissage**, assurant ainsi que le taux ne diminue pas trop rapidement au fil des itérations.

Mise à Jour des Paramètres

À chaque itération, le gradient est calculé et l'accumulateur de gradient est mis à jour selon la formule suivante :

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta)g_t^2$$

Ici, $E[g^2]_t$ représente la moyenne mobile du carré des gradients, β est le coefficient de décroissance, souvent fixé à 0.9, et g_t est le gradient calculé à l'itération t .

Cette approche donne plus de poids aux gradients récents tout en conservant une partie de l'information passée.

Ensuite, les paramètres du modèle sont mis à jour selon la règle suivante :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$

où : - θ_t est la valeur actuelle du paramètre, - η est le taux d'apprentissage, - ε est une petite constante ajoutée pour éviter la division par zéro.

Cette mise à jour assure que les grands gradients reçoivent des pas de mise à jour plus petits, tandis que les gradients plus petits peuvent encore faire progresser l'apprentissage, permettant ainsi **une convergence plus stable**.

Avantages de RMSprop

Cette méthode améliore l'adaptation aux différentes échelles de gradients, ce qui peut conduire à une convergence plus rapide dans certains contextes.

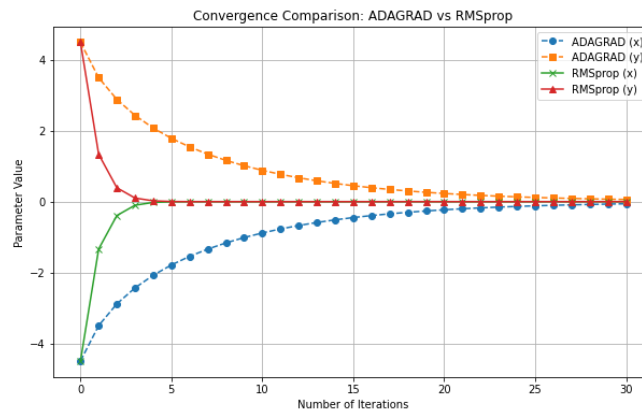


Figure 3.8: AdaGrad VS RMSprop

3.2.7 Adam (Adaptive Moment Estimation)

Adam est un algorithme d'optimisation qui combine deux méthodologies de descente de gradient : **RMSProp et Momentum**.

Comme RMSProp, Adam utilise le carré du gradient pour ajuster le taux d'apprentissage de manière adaptative (taux d'apprentissage adaptatif), et comme Momentum, il suit une moyenne mobile des gradients (momentum).

Cela fait d'Adam **un algorithme adaptatif par rapport aux moments**.

Mise à jour des moments

Les mises à jour des moments dans Adam se font comme suit :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Les paramètres sont ensuite mis à jour en fonction des estimations corrigées des moments :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$

Où m_t représente la moyenne mobile du premier moment (momentum) et v_t la moyenne mobile du second moment (vitesse), corrigés pour compenser les biais initiaux.

Avantages et performance d'Adam

Adam présente des avantages tels que des besoins en mémoire relativement faibles et une capacité à ajuster dynamiquement la taille des mises à jour des paramètres en fonction des gradients, ce qui le rend adapté aux problèmes comportant des gradients bruités ou épars. Grâce à sa combinaison des concepts de momentum et de RMSprop, Adam permet souvent **une convergence rapide avec de bonnes performances**.

Cependant, l'efficacité d'Adam **dépend fortement du choix des hyperparamètres**, ce qui reste une étape critique dans son utilisation.

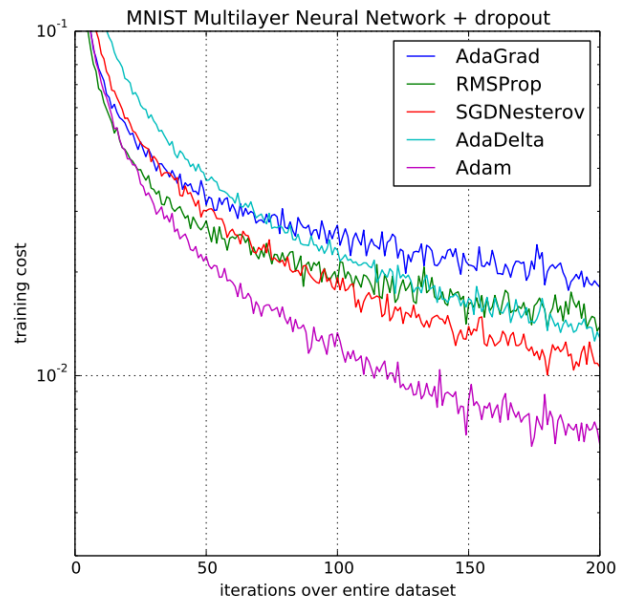


Figure 3.9: Comparaison des algorithmes d'optimisation (MNIST)

3.3 Sélection d'Optimiseurs en Fonction du Problème

Le choix de l'optimiseur dépend souvent des caractéristiques du problème spécifique et de la nature des données. Voici quelques considérations courantes lors de la sélection de l'optimiseur :

- Ensembles de données peu volumineux : **SGD avec Momentum** peut être préféré pour des ensembles de données plus petits, offrant un équilibre entre stabilité de convergence et efficacité computationnelle.
- Grandes quantités de données : Pour des ensembles de données massifs, **Adam** est souvent un choix efficace en raison de son adaptation dynamique des taux d'apprentissage.
- Problèmes de classification d'images (CNN) : **Adam** se comporte généralement bien sur des tâches de classification d'images utilisant des architectures de réseaux de neurones convolutifs (CNN).
- Réseaux récurrents (RNN) : Les méthodes d'adaptation des taux d'apprentissage, comme **Adam et RMSprop**, sont souvent utilisées pour l'entraînement de réseaux récurrents (RNN) en raison de la nature séquentielle des données.

3.4 Régularisation et Optimisation

L'entraînement des réseaux de neurones peut conduire à un surapprentissage (*overfitting*), surtout lorsque les données sont complexes ou en grand nombre. Pour pallier cela, diverses méthodes de régularisation sont utilisées. Dans cette section, nous abordons deux méthodes de régularisation populaires : la régularisation L2 et le Dropout.

3.4.1 Régularisation L2

La régularisation L2, également connue sous le nom de *ridge regression*, consiste à ajouter une pénalité sur la norme des poids afin de **réduire la complexité du modèle**. Cette méthode permet de contrôler la magnitude des poids du réseau en minimisant la somme des carrés des poids. La fonction de perte avec régularisation L2 s'écrit comme suit :

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{original}} + \lambda \sum_{i=1}^n w_i^2 \quad (3.5)$$

Où :

- $\mathcal{L}_{\text{total}}$ est la fonction de perte totale.
- $\mathcal{L}_{\text{original}}$ est la fonction de perte originale sans régularisation.
- λ est le coefficient de régularisation qui contrôle l'intensité de la pénalisation.
- w_i représente les poids du réseau.

Le terme de régularisation **encourage les poids à être plus petits**, ce qui réduit la variance du modèle et aide à prévenir le surapprentissage.

3.4.2 Dropout

Le *Dropout* est une autre technique de régularisation qui consiste à **ignorer aléatoirement certains neurones** pendant la phase d'entraînement afin de rendre le réseau plus robuste et moins dépendant de neurones spécifiques.

Lors de chaque itération d'entraînement, un sous-ensemble de neurones est désactivé avec une probabilité p prédéfinie, selon la formule suivante :

$$h_i^{(l)} = \begin{cases} 0 & \text{avec probabilité } 1 - p \\ \frac{h_i^{(l)}}{p} & \text{avec probabilité } p \end{cases} \quad (3.6)$$

Où :

- $h_i^{(l)}$ est l'activation du neurone i à la couche l .
- p est le taux de *Dropout*, généralement choisi entre 0.5 et 0.8.

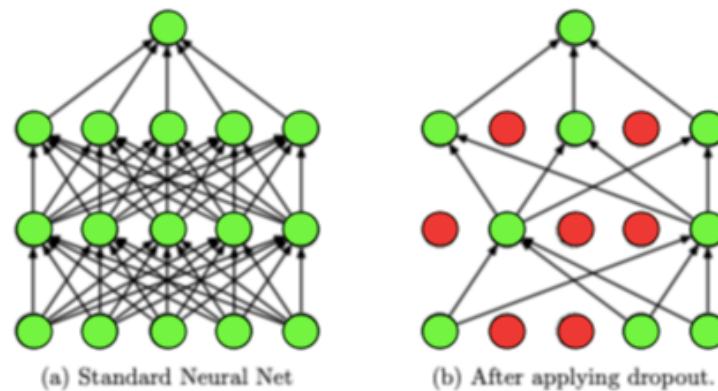


Figure 3.10: Application de Dropout

Le *Dropout* aide à **prévenir la co-adaptation excessive des neurones** et améliore la généralisation du réseau. Il est souvent utilisé conjointement avec d'autres techniques de régularisation pour obtenir de meilleurs résultats.

3.5 Hyperparamètres et Stratégies de Recherche

3.5.1 Hyperparamètres et leur Importance

Les **hyperparamètres** sont essentiels pour la performance des réseaux de neurones et doivent être définis avant l'entraînement. Les principaux hyperparamètres incluent :

- **Taux d'apprentissage** : Influence la vitesse d'ajustement des poids. Un taux trop élevé peut provoquer une divergence, tandis qu'un taux trop faible ralentit la convergence.

- **Taille des mini-lots** : Définit combien d'exemples sont utilisés avant la mise à jour des poids. Des lots plus petits accélèrent l'entraînement mais ajoutent du bruit, tandis que des lots plus grands stabilisent mais nécessitent plus de mémoire.
- **Nombre d'epochs** : Nombre de passes sur les données d'entraînement. Trop d'epochs cause du surapprentissage, tandis qu'un nombre trop faible empêche le modèle d'apprendre suffisamment.
- **Fonctions d'activation** : Déterminent comment les informations sont transmises dans le réseau, avec des choix comme ReLU, Sigmoid et Tanh ayant des impacts divers.

3.5.2 Stratégies de Recherche des Hyperparamètres

La recherche des hyperparamètres optimaux est un processus clé pour maximiser la performance d'un modèle de réseau de neurones. Les stratégies les plus courantes sont :

1. Recherche par Grille (*Grid Search*) : La recherche par grille consiste à définir un espace de recherche, c'est-à-dire un ensemble prédéfini de valeurs pour chaque hyperparamètre, et à évaluer systématiquement **toutes les combinaisons possibles**.

Bien que cette méthode soit exhaustive, elle est souvent **coûteuse en termes de calcul**, surtout lorsque le nombre d'hyperparamètres est élevé.

$$\text{Nombre de combinaisons} = \prod_{i=1}^n \text{valeurs possibles pour l'hyperparamètre } i \quad (3.7)$$

2. Recherche Aléatoire (*Random Search*) : Contrairement à la recherche par grille, la recherche aléatoire sélectionne de manière **aléatoire des combinaisons d'hyperparamètres** dans l'espace de recherche défini. Cela peut être plus efficace car il n'est pas nécessaire de tester toutes les combinaisons.

En fait, il a été montré que la recherche aléatoire peut parfois être **plus performante** que la recherche par grille, car elle explore plus efficacement l'espace des hyperparamètres.

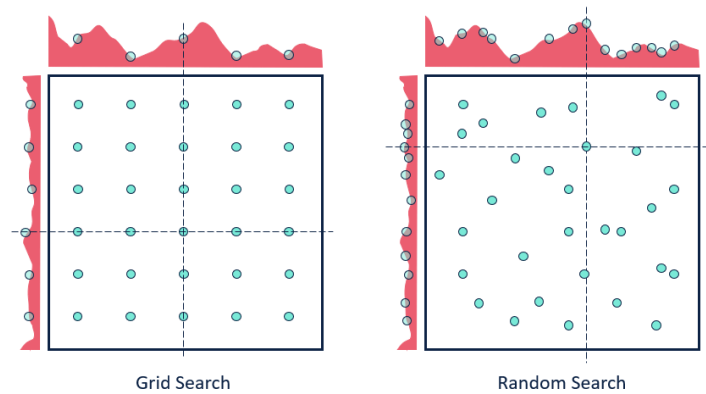


Figure 3.11: Recherche par Grille VS Recherche Aléatoire

3. Optimisation Bayésienne : L'optimisation bayésienne est **une méthode probabiliste** efficace pour rechercher les hyperparamètres optimaux d'un modèle d'apprentissage automatique. Contrairement aux approches classiques comme la recherche par grille ou la recherche aléatoire, cette méthode construit un modèle probabiliste basé sur les résultats des itérations précédentes et utilise ces informations pour orienter les prochaines configurations à tester.

La fonction objectif $\mathcal{L}(x)$, où x représente l'ensemble des hyperparamètres, est modélisée à l'aide d'un processus gaussien.

- À chaque étape, l'optimisation bayésienne sélectionne les hyperparamètres qui maximisent une fonction d'acquisition, combinant l'exploration de nouvelles régions et l'exploitation des configurations prometteuses déjà testées.

Cette approche permet de **réduire le nombre de tests inutiles** et d'optimiser efficacement le temps d'entraînement des modèles, tout en prenant en compte le dilemme entre exploration et exploitation.

3.5.3 Impact des Hyperparamètres sur la Performance du Modèle

Le choix des hyperparamètres est crucial pour la performance du modèle. Un taux d'apprentissage trop élevé empêche la convergence, tandis qu'un taux trop bas ralentit l'entraînement. De même, la taille des mini-lots affecte la vitesse d'entraînement et la stabilité des gradients.

Les courbes d'apprentissage aident à ajuster ces paramètres, une divergence entre la courbe d'entraînement et de validation indiquant souvent un surapprentissage, nécessitant des ajustements comme la réduction du nombre d'epochs ou l'ajout de régularisation.

3.6 Apprentissage par Transfert

L'apprentissage par transfert est une technique essentielle de l'apprentissage automatique qui permet d'adapter un modèle préalablement formé sur une tâche source à une tâche cible, souvent lorsque celle-ci dispose de peu de données étiquetées.

3.6.1 Concept de l'Apprentissage par Transfert

Ce processus repose sur l'idée que les représentations et les motifs appris sur la tâche source peuvent être bénéfiques pour améliorer les performances sur la tâche cible. Le processus se

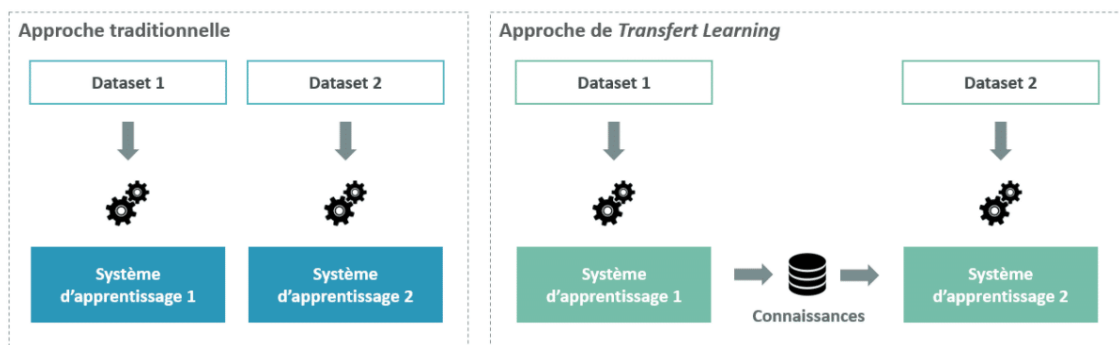


Figure 3.12: Apprentissage par Transfert

déroule en trois étapes principales :

- Formation sur une Tâche Source :** Un modèle est entraîné sur un ensemble de données de la tâche source, minimisant une fonction de perte spécifique.
- Transfert vers une Tâche Cible :** Le modèle pré-entraîné est adapté à l'aide de données étiquetées de la tâche cible, souvent en ajustant les poids du modèle par un processus de fine-tuning.
- Formalisation Mathématique :** Les représentations apprises sont formalisées par des fonctions de représentation, indiquant que les caractéristiques communes entre les tâches permettent une adaptation efficace.

3.6.2 Types d'Apprentissage par Transfert

Une fois que vous avez choisi un modèle pré-entraîné, plusieurs approches d'apprentissage par transfert (TL) peuvent être appliquées pour adapter ce modèle à votre tâche spécifique. Voici les principaux types d'apprentissage par transfert que vous pouvez envisager :

1. Transfert de Domaine

Après avoir chargé un modèle pré-entraîné, vous pouvez l'adapter à un domaine différent. Cela implique souvent de réentraîner le modèle sur des données du nouveau domaine pour améliorer ses performances.

- **Exemple :** Adapter un modèle de classification d'images pré-entraîné sur ImageNet pour classer des images médicales.

2. Transfert de Tâche

Cette approche consiste à adapter un modèle pré-entraîné à une tâche différente mais connexe. Vous pouvez ajuster les couches finales pour qu'elles correspondent à votre nouvelle tâche.

- **Exemple :** Utiliser un modèle pré-entraîné pour la classification d'images et l'ajuster pour la détection d'objets.

3. Transfert de Représentation

Ici, vous extrayez des caractéristiques à partir des couches d'un modèle pré-entraîné et les utilisez dans un nouveau modèle. Cela est particulièrement utile lorsque vous avez des données limitées.

- **Exemple :** Utiliser les couches convolutives d'Inception V3 pour extraire des caractéristiques, puis les alimenter dans un modèle de classification personnalisé.

4. Fine-Tuning

Le fine-tuning implique de continuer l'entraînement d'un modèle pré-entraîné sur votre propre ensemble de données avec un taux d'apprentissage réduit. Cela permet au modèle de mieux s'adapter à votre tâche spécifique.

- **Exemple :** Prendre un modèle BERT pré-entraîné et l'affiner sur un ensemble de données étiquetées pour une tâche de classification de texte.

5. Transfert de Connaissances

Cette approche utilise des mécanismes de distillation de connaissances pour transférer l'apprentissage d'un modèle plus grand (modèle enseignant) vers un modèle plus petit (modèle étudiant).

- **Exemple :** Entraîner un modèle étudiant pour qu'il imite le comportement d'un modèle enseignant, permettant ainsi d'obtenir des performances similaires avec moins de ressources.

6. Transfert Multi-Task

Cette méthode utilise un modèle qui apprend simultanément plusieurs tâches connexes, ce qui permet de partager des informations entre les tâches et d'améliorer la performance globale.

- **Exemple :** Un modèle qui effectue à la fois la classification et la segmentation d'images en partageant des couches intermédiaires.

7. Transfert par Réseaux Antagonistes

Cette approche utilise des techniques telles que les GANs (Generative Adversarial Networks) pour transférer des connaissances entre les modèles.

- **Exemple :** Entraîner un modèle à générer des images de haute qualité à partir d'images de faible qualité, puis utiliser ce modèle pour des tâches de classification.

En conclusion, l'apprentissage par transfert optimise l'apprentissage dans des situations de données limitées, exploitant les connaissances acquises pour améliorer la performance des modèles dans des domaines variés.

4 Conclusion

En conclusion, l'optimisation des réseaux de neurones, bien qu'elle repose initialement sur des concepts de base comme la descente de gradient, a évolué pour intégrer des méthodes plus sophistiquées comme SGD avec Momentum, NAG, Adagrad, RMSprop et Adam. Chacun de ces algorithmes présente des avantages et des inconvénients en fonction des spécificités des données et du problème à résoudre.

Le choix de l'optimiseur doit être guidé par la nature du problème et la structure des données. Les méthodes plus simples comme la descente de gradient stochastique sont utiles pour des ensembles de données de taille modeste, tandis que des techniques avancées comme Adam ou RMSprop sont plus adaptées aux grands volumes de données ou aux réseaux complexes comme les CNN et RNN.

Ainsi, l'optimisation joue un rôle crucial dans la performance des modèles de deep learning, et la maîtrise des différents optimiseurs permet d'améliorer la convergence, la stabilité et l'efficacité des modèles.

En outre, il ne faut pas négliger l'impact de la régularisation dans le processus d'optimisation. Des techniques telles que la régularisation L2 ou le dropout sont souvent essentielles pour éviter le surapprentissage et améliorer la capacité du modèle à généraliser sur des données inconnues. Le réglage des hyperparamètres, notamment le taux d'apprentissage, les coefficients de régularisation et la taille des lots, joue également un rôle clé dans la réussite de l'entraînement des réseaux de neurones.

Finalement, une bonne optimisation va au-delà du simple choix d'un algorithme. Elle exige une combinaison de régularisation, de gestion de l'initialisation des poids, ainsi que l'ajustement minutieux des hyperparamètres pour assurer une convergence rapide et stable. Une compréhension approfondie des différentes techniques permet d'optimiser les performances du modèle tout en assurant sa robustesse face à de nouvelles données.