RÉPUBLIQUE
FRANÇAISE

Liberté
Égalité
Fraternité

IGPDE

Institut de la Gestion publique
et du Développement économique

RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

INTRODUCTION AU DEEP LEARNING PAR LA PRATIQUE

FORMATEUR : SAMY AMRI

Sommaire

1. Introduction

2. Fondements

3. Introduction aux frameworks

4. Réseaux de neurones convolutifs (CNN)

5. Traitement du langage naturel (NLP)

6. Réseaux de neurones récurrents (RNN)

7. Transformers et attention

8. Problèmes courants et optimisation

9. Ressources complémentaires

1. Introduction

a. Intérêt de la formation

Le Deep Learning représente aujourd'hui l'une des branches les plus dynamiques et prometteuses de l'intelligence artificielle.

Cette discipline a révolutionné de nombreux domaines tels que **la vision par ordinateur, le traitement du langage naturel, la reconnaissance vocale, et bien d'autres.** Sa capacité à résoudre des problèmes complexes en apprenant directement à partir des données en fait un outil puissant pour l'innovation technologique.

Cette formation vous propose une immersion complète dans l'univers du Deep Learning, **depuis les fondamentaux théoriques jusqu'aux architectures avancées**.

Vous apprendrez à concevoir, implémenter et optimiser des réseaux de neurones profonds en utilisant les bibliothèques les plus populaires comme TensorFlow, PyTorch et Keras.

Pourquoi la formation ?

— Pour acquérir une compréhension solide des principes mathématiques et algorithmiques qui soustendent le Deep Learning.

— Pour maîtriser les frameworks les plus utilisés dans l'industrie et la recherche.

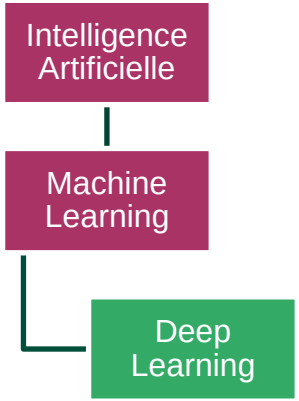
— Pour développer la capacité à concevoir des solutions innovantes basées sur les réseaux de neurones profonds.

— Pour vous préparer aux défis actuels et futurs de l'intelligence artificielle.

b. Deep Learning vs Machine Learning

Le Deep Learning est une sous-branche du Machine Learning, qui elle-même fait partie du domaine plus large de l'Intelligence Artificielle

Alors que le Machine Learning traditionnel s'appuie souvent sur l'extraction manuelle de caractéristiques (feature engineering), le Deep Learning se distingue par sa capacité à apprendre automatiquement des représentations hiérarchiques directement à partir des données brutes.



b. Deep Learning vs Machine Learning

La principale différence réside dans l'architecture et la profondeur des modèles.

Le terme "deep" (profond) fait référence à la présence de multiples couches cachées dans les réseaux de neurones, permettant d'abstraire et de modéliser des relations complexes dans les données.

Prenons le cas de la reconnaissance d'images :

- **En Machine Learning classique** : Un expert définit manuellement les caractéristiques pertinentes (contours, textures, etc.), puis un algorithme comme SVM ou Random Forest apprend à classifier les images.
- **En Deep Learning** : Un réseau de neurones profond apprend automatiquement à extraire des caractéristiques de plus en plus abstraites. Les premières couches détectent des contours simples, puis des textures, des motifs, et enfin des concepts de haut niveau comme des visages ou des objets.

Cette différence fondamentale explique pourquoi le Deep Learning excelle dans des domaines où la définition manuelle de caractéristiques est difficile ou inefficace, comme la vision par ordinateur, le traitement du langage naturel ou la reconnaissance vocale.

b. Deep Learning vs Machine Learning

Pour le reste, les méthodes déjà vues en Machine Learning sont parfaitement valables en Deep Learning.

Le but est toujours formellement de trouver modèle f_θ de paramètres θ qui minimise la perte moyenne.

Paramètres et hyperparamètres :

- Paramètres θ : poids et biais, appris par l'entraînement
- Hyperparamètres : choisis par vous (learning rate, nombre de couches, taille des lots, époques)
- Les hyperparamètres **se règlent sur la validation, jamais sur le test**

c. Histoire

Le Deep Learning a connu une histoire fascinante, marquée par des périodes d'enthousiasme suivies de "hivers de l'IA", avant de connaître sa renaissance spectaculaire dans les années 2010.

- **1943-1958** Premiers modèles théoriques de neurones artificiels (McCulloch & Pitts, Perceptron de Rosenblatt)
- **1969-1986** "Premier hiver de l'IA" suite aux limitations du perceptron simple
- **1986-1995** Algorithme de rétropropagation du gradient, premières applications pratiques
- **1995-2006** "Second hiver de l'IA", focus sur les SVM et autres méthodes
- **2006-2012** Renaissance avec le "Deep Learning" (Hinton, LeCun, Bengio)
- **2012-présen**t Explosion des applications avec AlexNet, DeepMind, GPT, etc.

c. Histoire

La résurgence du Deep Learning dans les années 2010 est principalement due à trois facteurs :

- **Disponibilité massive de données** (Big Data)
- **Puissance de calcul accrue** (GPU, TPU)
- **Avancées algorithmiques** (nouvelles fonctions d'activation, techniques de régularisation, architectures innovantes)

Le tournant majeur a eu lieu en 2012 avec la victoire écrasante du réseau convolutif AlexNet au concours ImageNet. Cette démonstration de la supériorité des réseaux de neurones profonds a déclenché une véritable révolution dans le domaine de l'IA, attirant l'attention des chercheurs, des entreprises et du grand public.

Liens utiles :

https://en.wikipedia.org/wiki/AlexNet?oldid=1211899570

https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

d. Exemples sectoriels

Le Deep Learning a transformé de nombreux secteurs et continue d'ouvrir de nouvelles possibilités d'innovation :

Secteur	Applications	Exemples concrets
Santé	Diagnostic médical, découverte de médicaments	Detection de cancer sur IRM, AlphaFold
Finance	Détection de fraude, trading algorithmique	Systèmes anti-blanchiment, prédiction de cours
Transports	Véhicules autonomes, optimisation logistique	Tesla Autopilot, optimisation de flottes
Industrie	Maintenance prédictive, contrôle qualité	Détection de défauts, optimisation de production
Médias	Génération de contenu, recommandation	DALL-E, systèmes de recommandation Netflix
Sciences	Modélisation climatique, physique des particules	DeepMind pour la prédiction de protéines

e. Présentation des outils

Plusieurs frameworks open-source dominent actuellement l'écosystème du Deep Learning :

Outil	Nature	Entraînement	Prise en main	Idéal pour
PyTorch	Bibliothèque complète de deep learning	Boucle écrite à la main, loss.backward(), très transparente	Progressive, les mécanismes sont visibles	Recherche, prototypage, comprendre les mécanismes
TensorFlow	Plateforme complète (calcul, entraînement, déploiement)	tf.GradientTape ou fit via Keras	Plus d'abstractions à connaître	Production à grande échelle, mobile
Keras	API de haut niveau (backends TensorFlow, PyTorch ou JAX depuis Keras 3)	compile puis fit, boucle prête à l'emploi	La plus rapide pour débuter	Débutants, prototypes rapides, code concis

f. Installation des environnements

À installer (une seule fois) :

1. **Python 3.11** : python.org > Downloads > version 3.11.x. Cochez **« Add python.exe to PATH »**.
2. **VS Code** : code.visualstudio.com.
3. **Extensions VS Code** (Ctrl+Maj+X) : **Python** et **Jupyter**

À récupérer

- Le dossier **kit** fourni par le formateur, à copier dans un chemin court, par exemple C:\Users\vous\kit.
- Il doit contenir notebooks\, data\ et check_env.py. Le dossier data doit se trouver **à côté** de notebooks.

Évitez les dossiers au chemin très long ou contenant des caractères spéciaux (&, accents).

f. Installation des environnements

Dans VS Code : **Fichier > Ouvrir un dossier** > choisir kit, puis **Terminal > Nouveau terminal**.

py -3.11 -m venv .venv
.\.venv\Scripts\Activate.ps1

Le terminal affiche (.venv) au début de la ligne.
Si l'activation est refusée, taper une fois :

Set-ExecutionPolicy -Scope CurrentUser RemoteSigned

puis relancer l'activation.

f. Installation des environnements

Avec (.venv) affiché (quelques minutes, 1 à 2 Go, internet nécessaire) :

python -m pip install --upgrade pip
pip install torch torchvision --index-url https://download.pytorch.org/whl/cpu
pip install tensorflow
pip install numpy pandas matplotlib scikit-learn ipykernel transformers

Vérification :

python -c "import tensorflow as tf, keras, torch; print(tf.__version__, keras.__version__, torch.__version__)«

Les trois numéros de version doivent s'afficher sans erreur.

2. Fondements

a. Algèbre linéaire

L'algèbre linéaire constitue le langage mathématique fondamental du Deep Learning.

Les réseaux de neurones manipulent essentiellement des vecteurs, des matrices et des tenseurs à travers diverses opérations linéaires.

Concept	Définition	Application en Deep Learning
Vecteurs	Tableaux 1D de valeurs	Représentation des caractéristiques d'entrée
Matrices	Tableaux 2D de valeurs	Poids des connexions entre couches
Tenseurs	Tableaux multidimensionnels	Données d'entrée (images, séquences, etc.)
Produit scalaire	$\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i$	Calcul d'une combinaison linéaire
Produit matriciel	$\mathbf{C} = \mathbf{A} \times \mathbf{B}$	Propagation avant dans un réseau
Transposition	$\mathbf{A}^T_{ij} = \mathbf{A}_{ji}$	Rétropropagation du gradient

a. Algèbre linéaire

Dans le script *algebre_lineaire.py*, le produit matriciel np.dot(x, W) représente l'opération fondamentale d'une couche entièrement connectée (Dense) dans un réseau de neurones.

Cette opération calcule la somme pondérée des entrées pour chaque neurone de la couche suivante.

L'ajout du biais + b permet au réseau d'apprendre un décalage pour chaque neurone, augmentant sa capacité de modélisation.

En plus des objets classiques, le **tenseur** est une structure de base : c'est un tableau à **n dimensions**. Il porte 3 attributs : **shape** (la forme), **dtype** (souvent float32, 4 octets par valeur) et **device** (le CPU ici).

- Scalaire : shape ()
- Vecteur : (n)
- Matrice : (m, n)
- Image couleur : (3, 32, 32) en PyTorch
- Lot de 64 images : (64, 3, 32, 32)

a. Algèbre linéaire

Ouvrir le TP1 *algebre_lineaire.ipynb* qui servira de fil rouge pour ce chapitre.

b. Calcul différentiel et descente de gradient

Le calcul différentiel est au coeur de l'apprentissage des réseaux de neurones. La descente de gradient, basée sur le calcul des dérivées partielles, permet d'optimiser les paramètres du réseau en minimisant une fonction de perte.

Concepts fondamentaux :

- **Dérivée** : Taux de variation d'une fonction par rapport à une variable
- **Dérivée partielle** : Dérivée par rapport à une variable en maintenant les autres constantes
- **Gradient** : Vecteur des dérivées partielles d'une fonction multivariable
- **Règle de chaîne** : Principe fondamental permettant la rétropropagation du gradient

La descente de gradient met à jour les paramètres θ selon la formule :

b. Calcul différentiel et descente de gradient

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta)$$

Où :

- θ_t représente les paramètres actuels
- η est le taux d'apprentissage (hyperparamètre)
- $\nabla_\theta J(\theta)$ est le gradient de la fonction de perte J par rapport aux paramètres θ

Cette règle généralise la suivante : si y = g(h(x)), alors dy/dx = g′(h(x)) · h′(x)

b. Calcul différentiel et descente de gradient

Dans le script calcul_diff.py, l'exemple illustre le principe fondamental de la descente de gradient sur une fonction simple à une variable.

Dans les réseaux de neurones, ce principe est appliqué à des fonctions multivariables (avec potentiellement des millions de paramètres), mais la logique reste la même : **calculer le gradient de la fonction de perte par rapport à chaque paramètre et ajuster ces paramètres dans la direction opposée au gradient pour minimiser la perte.**

La rétropropagation du gradient (backpropagation) est une application efficace de la règle de chaîne qui permet de calculer ces gradients couche par couche, en partant de la sortie et en remontant vers l'entrée du réseau.

c. Probabilités et statistiques

Les probabilités et statistiques jouent un rôle crucial dans les fondements théoriques du Deep Learning, notamment pour la conception des fonctions de perte, la modélisation des données et l'évaluation des performances.

Concepts probabilistes fondamentaux :

Distributions de probabilité : Modélisation de la répartition des données

Espérance et variance : Mesures statistiques fondamentales

Maximum de vraisemblance : Principe d'estimation des paramètres

Théorème de Bayes : Mise à jour des croyances avec nouvelles informations

c. Probabilités et statistiques

Concept statistique	Formule	Application en Deep Learning
Distribution normale	$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{\sigma^2}}$	Initialisation des poids, régularisation
Entropie croisée	$H(p,q) = \sum_x p(x) log q(x)$	Fonction de perte pour classification
Divergence KL	$D_{KL}(p\|\|q) = \sum_x p(x) log \frac{p(x)}{q(x)}$	Mesure de différence entre distributions
Maximum de vraisemblance	$\theta_{MLE} = argmax_\theta P(X\|\theta)$	Principe d'entraînement supervisé

c. Probabilités et statistiques

Dans le contexte du Deep Learning, l'entropie croisée mesure la différence entre la distribution des étiquettes réelles et les probabilités prédites par le modèle.

Une entropie croisée plus faible indique une meilleure correspondance entre les prédictions et la réalité.

Cette mesure est particulièrement adaptée aux problèmes de classification où la sortie du réseau est normalisée via une fonction softmax pour représenter une distribution de probabilités sur les différentes classes.

Dans le script *entropie_croisee.py*, on a un exemple de ce type de calcul.

Exercice : Comparer la fonction d'entropie croisée avec une perte d'entropie croisée binaire dans le script *entropie_croisee_binaire.py*, utilisée dans les problèmes de classification binaire.

c. Probabilités et statistiques

L'implémentation dans le scripit *entropie_croisee_binaire.py* calcule l'entropie croisée binaire, qui est adaptée aux problèmes où la cible est soit 0, soit 1. On observe que :

- Des prédictions proches des vraies valeurs (cas 1) donnent une perte faible
- Des prédictions incertaines (cas 2) donnent une perte intermédiaire
- Des prédictions erronées (cas 3) donnent une perte élevée

La visualisation montre comment la perte augmente à mesure que la prédiction s'éloigne de la valeur réelle avec une croissance exponentielle lorsqu'on s'approche des extrêmes.

d. Le neurone artificiel

Le neurone artificiel, inspiré du neurone biologique, est l'unité de calcul fondamentale des réseaux de neurones. Son fonctionnement peut être résumé en trois étapes principales :

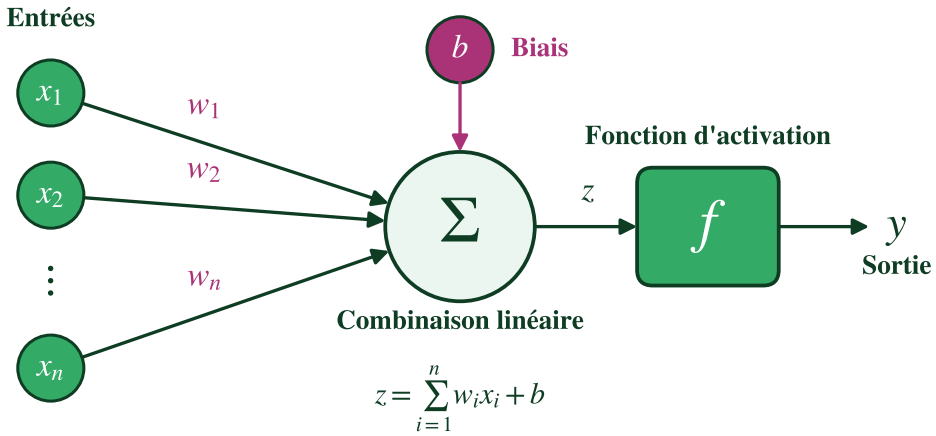
1. **Agrégation** : Combinaison linéaire des entrées pondérées
2. **Addition du biais** : Ajout d'un terme constant
3. **Activation** : Application d'une fonction non-linéaire

Mathématiquement, pour un neurone avec n entrées x = [x1, x2, . . . , xn], des poids w = [w1,w2, . . . ,wn] et un biais b, la sortie y est calculée comme suit :

$$y = f(\sum_{i=1}^{n} wixi + b) = f(w.x + b)$$

où f est la fonction d'activation.

d. Le neurone artificiel



$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w} \cdot \mathbf{x} + b \qquad y = f(z)$$

d. Le neurone artificiel

Le neurone artificiel est une unité de calcul qui transforme ses entrées en une sortie unique via une combinaison linéaire suivie d'une non-linéarité. Voir le script *neurone.py.*

Cette non-linéarité, apportée par la fonction d'activation, est cruciale car elle permet au réseau de modéliser des relations complexes entre les entrées et les sorties.

Sans ces non-linéarités, même un réseau profond se réduirait à une simple transformation linéaire.

Les fonctions d'activation comme la sigmoïde et la tangente hyperbolique ont l'avantage d'être bornées, ce qui facilite la convergence pendant l'apprentissage. Cependant, elles souffrent du problème de gradient évanescent pour les réseaux profonds. La fonction ReLU (Rectified Linear Unit) a gagné en popularité car elle n'est pas sujette à ce problème et permet un calcul plus rapide, bien qu'elle puisse mener au problème des "neurones morts" si le gradient devient nul.

e. Fonctions d'activation

Fonction	Équation	Avantages	Inconvénients
Sigmoïde	$\sigma(x) = 1 / (1 + e^{-x})$	Sortie bornée [0, 1], différentiable	Gradient évanescent, sortie non centrée
Tanh	$\tanh(x) = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$	Sortie bornée [−1, 1] et centrée	Gradient évanescent aux extrêmes
ReLU	$\max(0, x)$	Calcul rapide, prévient le gradient évanescent	Neurones « morts » (gradient = 0)
Leaky ReLU	$\max(\alpha x, x)$, $\alpha \approx 0{,}01$	Évite le problème des neurones morts	Peu d'avantages empiriques par rapport à ReLU
ELU	x si $x > 0$ $\alpha(e^{x} - 1)$ si $x \leq 0$	Sortie moyenne proche de zéro	Coût computationnel plus élevé
Softmax	$\text{softmax}(x_i) = \exp(x_i) / \Sigma_j \exp(x_j)$	Normalisation en distribution de probabilité	Sensible aux valeurs extrêmes

e. Fonctions d'activation

Les fonctions d'activation sont cruciales car elles déterminent la capacité du réseau à apprendre des patterns complexes.

La fonction ReLU est devenue standard dans les couches cachées en raison de sa simplicité computationnelle et de sa capacité à éviter le gradient évanescent.

La fonction softmax est principalement utilisée dans la couche de sortie pour les problèmes de classification multi-classes, car elle produit une distribution de probabilité normalisée.

Le choix de la fonction d'activation dépend du contexte :

- **Couches cachées** : ReLU ou ses variantes (Leaky ReLU, ELU)
- **Sortie binaire** : Sigmoid
- **Sortie multi-classes** : Softmax
- **Régression** : Aucune activation (linéaire)

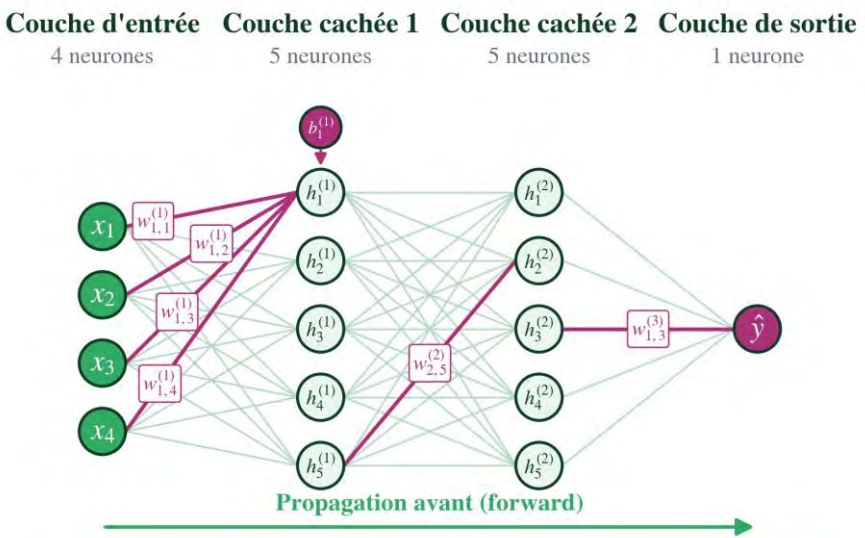
Le script *fonction_activ.py* permet de les visualiser.

f. Architecture des réseaux feedforward

Les réseaux de neurones feedforward, également appelés perceptrons multicouches (MLP), sont organisés en couches successives où l'information circule uniquement de l'entrée vers la sortie, sans connexions récurrentes.

1. **Couche d'entrée** : Reçoit les données brutes
2. **Couches cachées** : Effectuent des transformations successives
3. **Couche de sortie** : Produit la prédiction finale

f. Architecture des réseaux feedforward

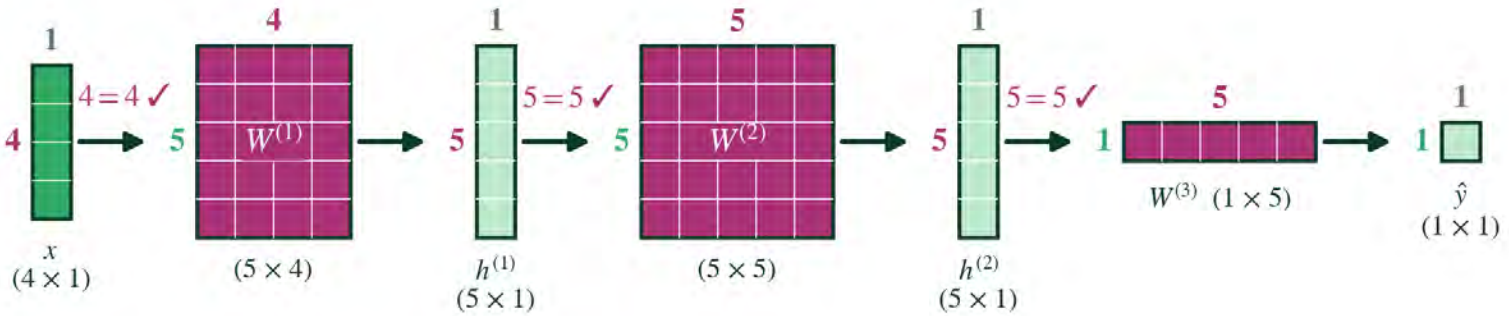


$$h^{(1)} = f(W^{(1)}x + b^{(1)}) \qquad h^{(2)} = f(W^{(2)}h^{(1)} + b^{(2)}) \qquad \hat{y} = g(W^{(3)}h^{(2)} + b^{(3)})$$

$w_{j,i}^{(l)}$: poids de la connexion du neurone i de la couche $l-1$ vers le neurone j de la couche l

f. Architecture des réseaux feedforward

Chaque étape : $h = f(W \cdot v + b)$ — **le nombre de colonnes de W doit être égal au nombre de lignes du vecteur qui entre**

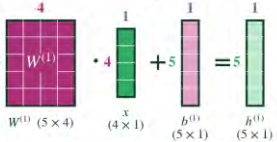


Les nombres magenta doivent coïncider ; les nombres verts donnent la taille du résultat (le nombre de neurones de la couche).

f. Architecture des réseaux feedforward

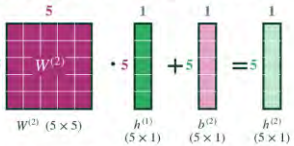
Règle d'or : $(m\times n)\cdot(n\times p)\to(m\times p)$: les dimensions « du milieu » (en magenta) doivent être égales

① Couche cachée 1 : $h^{(1)}=f(W^{(1)}x+b^{(1)})$



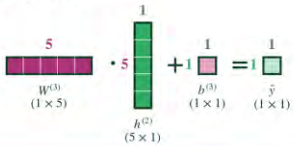
Vérification
$(5\times4)\cdot(4\times1)\to(5\times1)$
colonnes de $W^{(1)}$ = lignes de x : $4=4$ ✓
résultat : 5 neurones ✓

② Couche cachée 2 : $h^{(2)}=f(W^{(2)}h^{(1)}+b^{(2)})$



Vérification
$(5\times5)\cdot(5\times1)\to(5\times1)$
colonnes de $W^{(2)}$ = lignes de $h^{(1)}$: $5=5$ ✓
résultat : 5 neurones ✓

③ Sortie : $\hat{y}=g(W^{(3)}h^{(2)}+b^{(3)})$



Vérification
$(1\times5)\cdot(5\times1)\to(1\times1)$
colonnes de $W^{(3)}$ = lignes de $h^{(2)}$: $5=5$ ✓
résultat : 1 neurone ✓

Avec un lot de n exemples (une ligne par exemple, X de forme $n\times4$) : $H^{(1)}=f(XW^{(1)T}+b^{(1)})$
$(n\times4)\cdot(4\times5)\to(n\times5)$ $(n\times5)\cdot(5\times5)\to(n\times5)$ $(n\times5)\cdot(5\times1)\to(n\times1)$
Paramètres : $(5\times4+5)+(5\times5+5)+(1\times5+1)=25+30+6=\mathbf{61}$

f. Architecture des réseaux feedforward

Un réseau feedforward propage l'information de la couche d'entrée vers la couche de sortie à travers **une série de transformations non linéaires**.

Chaque couche extrait des caractéristiques de plus en plus abstraites des données d'entrée.

Dans l'exemple du script *feedforward.py*, nous avons implémenté un réseau à deux couches cachées, mais cette architecture peut être généralisée à un nombre arbitraire de couches. Le choix de l'initialisation des poids est crucial pour la convergence du réseau.

L'initialisation de He (pour ReLU) et Xavier/Glorot (pour sigmoid/tanh) sont des techniques populaires qui aident à maintenir la variance du signal à travers les couches, évitant ainsi les problèmes de gradient évanescent ou explosif.

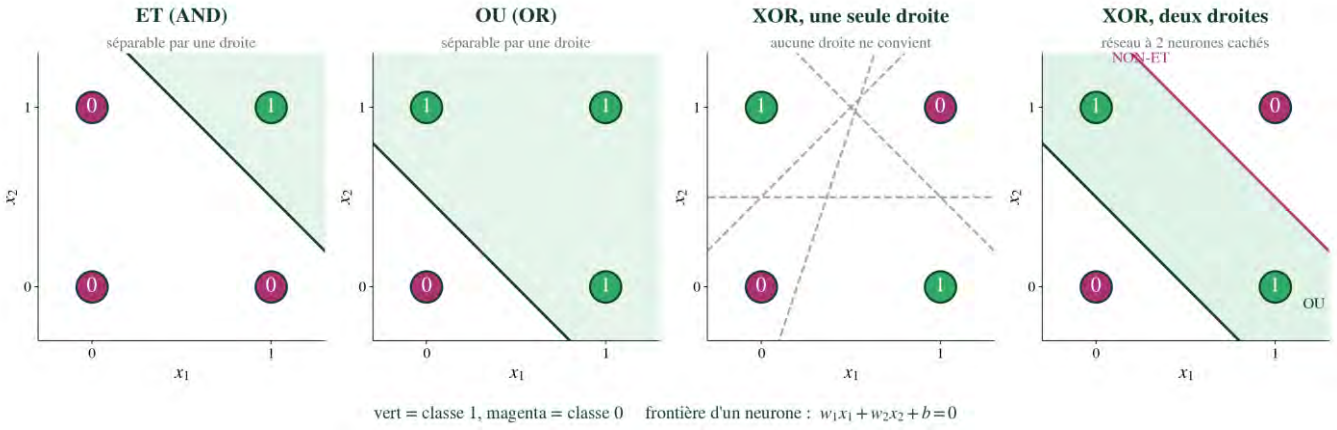
Notez que cette implémentation ne comprend pas l'apprentissage par rétropropagation, que nous aborderons dans la section suivante.

g. TP2_perceptron

Essayer de compléter le *TP2_perceptron.ipynb* intégralement tout en analysant en détail les codes fournis.

g. TP2

Explication : Le perceptron simple ne peut pas apprendre la fonction XOR car cette fonction **n'est pas linéairement séparable**. **Il est impossible de tracer une ligne droite qui sépare parfaitement les classes dans l'espace bidimensionnel**. Cette limitation historique du perceptron a mené au développement des réseaux multicouches, qui peuvent résoudre des problèmes non linéairement séparables grâce aux couches cachées et aux fonctions d'activation non linéaires.



h. Algorithme de rétropropagation

La rétropropagation (backpropagation) est l'algorithme fondamental qui permet d'entraîner les réseaux de neurones multicouches. Il utilise la règle de chaîne pour calculer efficacement les gradients de la fonction de perte par rapport à tous les paramètres du réseau.

Étapes de la rétropropagation :

1. **Propagation avant** : Calcul des activations couche par couche
2. **Calcul de la perte** : Évaluation de l'erreur entre prédiction et vérité terrain
3. **Propagation arrière** : Calcul des gradients en remontant le réseau
4. **Mise à jour des paramètres** : Ajustement des poids et biais selon les gradients

h. Algorithme de rétropropagation

L'algorithme de rétropropagation fonctionne en deux phases principales :

Phase 1 - Propagation avant : Les données traversent le réseau couche par couche, chaque couche appliquant une transformation linéaire suivie d'une fonction d'activation non linéaire.

Phase 2 - Propagation arrière : Le gradient de la fonction de perte est calculé en remontant le réseau, en utilisant la règle de chaîne pour décomposer le calcul complexe en étapes simples.

La clé de la rétropropagation réside dans la règle de chaîne du calcul différentiel : $\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j}{\partial w_{ij}^{(l)}}$

∂L où L est la fonction de perte, w(l)ij est le poids connectant le neurone i de la couche l−1 au neurone j de la couche l, et z(l) j est l'entrée nette du neurone j dans la couche l.

h. Algorithme de rétropropagation

Le réseau

- 1 neurone caché (sigmoïde)
- 1 neurone de sortie linéaire
- Perte : $L = (\hat{y} - y)^2$

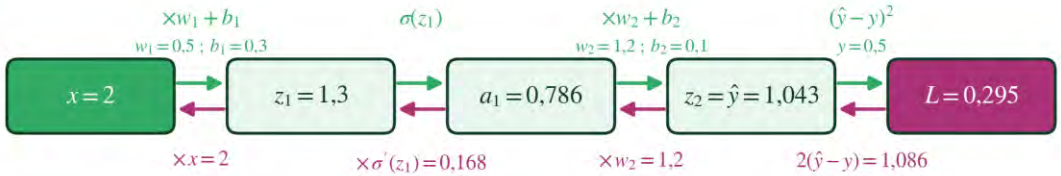
Les valeurs

- Entrée $x = 2$, cible $y = 0{,}5$
- $w_1 = 0{,}5$; $b_1 = 0{,}3$
- $w_2 = 1{,}2$; $b_2 = 0{,}1$

Étape	Calcul	Valeur
z_1	$w_1x + b_1 = 0{,}5 \times 2 + 0{,}3$	**1,3**
a_1	$\sigma(z_1) = \sigma(1{,}3)$	**0,786**
$z_2 = \hat{y}$	$w_2a_1 + b_2 = 1{,}2 \times 0{,}786 + 0{,}1$	**1,043**
L	$(\hat{y} - y)^2 = (1{,}043 - 0{,}5)^2$	**0,295**

h. Algorithme de rétropropagation

Propagation avant : on calcule les valeurs



Propagation arrière : on multiplie les dérivées locales

$$\frac{\partial L}{\partial w_1} = 0,439 \qquad \frac{\partial L}{\partial z_1} = 0,219 \qquad \frac{\partial L}{\partial a_1} = 1,303 \qquad \frac{\partial L}{\partial z_2} = 1,086$$

$$= \frac{\partial L}{\partial b_1} \text{ car } \frac{\partial z_1}{\partial b_1} = 1 \qquad = \frac{\partial L}{\partial b_2} \text{ car } \frac{\partial z_2}{\partial b_2} = 1$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} = 1,086 \times 1 \times 1,2 \times 0,168 \times 2 \approx 0,439$$

h. Algorithme de rétropropagation

Gradients intermédiaires

- $\partial L/\partial z_2 = 2(\hat{y} - y) = 1{,}086$
- $\partial L/\partial a_1 = 1{,}086 \times w_2 = 1{,}303$
- $\partial L/\partial z_1 = 1{,}303 \times \sigma'(z_1) = 0{,}219$
- (avec $\sigma'(z_1) = a_1(1 - a_1) = 0{,}168$)

Paramètre	Règle de la chaîne	Calcul	Gradient
w_2	$\partial L/\partial z_2 \times a_1$	$1{,}086 \times 0{,}786$	**0,853**
b_2	$\partial L/\partial z_2 \times 1$	1,086	**1,086**
w_1	$\partial L/\partial z_1 \times x$	$0{,}219 \times 2$	**0,439**
b_1	$\partial L/\partial z_1 \times 1$	0,219	**0,219**

h. Algorithme de rétropropagation

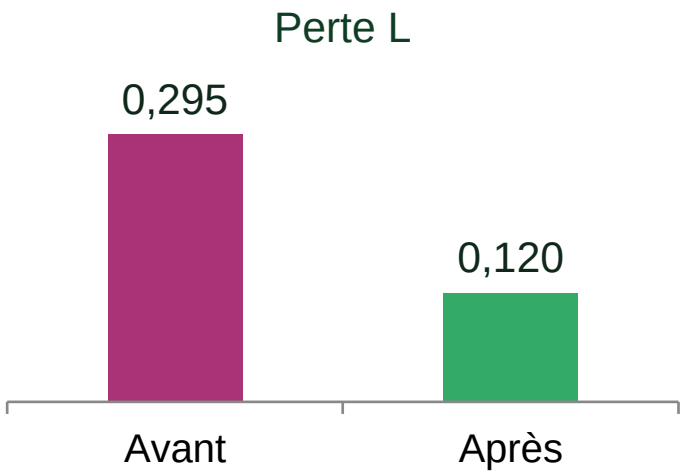
$$p \leftarrow p - \eta \cdot \partial L/\partial p$$

Le gradient indique le sens où la perte augmente : on avance dans le sens opposé, d'un pas proportionnel au taux d'apprentissage $\eta = 0{,}1$.

Paramètre	Avant	Gradient	Calcul	Après
w_1	0,500	0,439	$0{,}5 - 0{,}1 \times 0{,}439$	**0,456**
b_1	0,300	0,219	$0{,}3 - 0{,}1 \times 0{,}219$	**0,278**
w_2	1,200	0,853	$1{,}2 - 0{,}1 \times 0{,}853$	**1,115**
b_2	0,100	1,086	$0{,}1 - 0{,}1 \times 1{,}086$	**−0,009**

h. Algorithme de rétropropagation

	Avant	Après
z_1	1,300	1,190
a_1	0,786	0,767
Prédiction ŷ	1,043	0,846
Perte L	**0,295**	**0,120**



- **Une seule mise à jour réduit la perte d'environ 59 % ici.**
- L'entraînement répète ce cycle des milliers de fois : propagation avant, propagation arrière, mise à jour.
- Taux d'apprentissage : trop petit, la perte baisse très lentement ; trop grand, le pas dépasse le minimum et la perte peut augmenter, voire diverger.

i. Fonctions de perte

Le choix de la fonction de perte est crucial car elle définit l'objectif d'optimisation du modèle

Différentes tâches nécessitent différentes fonctions de perte :

Type de tâche	Fonction de perte	Formule mathématique	Activation de sortie
Classification binaire	Cross-entropie binaire	$-(1/m)\ \Sigma_{i=1\ldots m}\ [\ y_i\ \log(\hat{y}_i) + (1 - y_i)\ \log(1 - \hat{y}_i)\ ]$	Sigmoïde
Classification multi-classes	Cross-entropie catégorielle	$-(1/m)\ \Sigma_{i=1\ldots m}\ \Sigma_{j=1\ldots C}\ y_{ij}\ \log(\hat{y}_{ij})$	Softmax
Régression	Erreur quadratique moyenne	$(1/2m)\ \Sigma_{i=1\ldots m}\ (y_i - \hat{y}_i)^2$	Aucune (linéaire)
Régression robuste	Erreur absolue moyenne	$(1/m)\ \Sigma_{i=1\ldots m}\ \lvert\ y_i - \hat{y}_i\ \rvert$	Aucune (linéaire)

i. Métriques d'évaluation

Une seule métrique ne suffit pas : chaque prédiction tombe dans l'un des quatre cas.

- VP (vrai positif) : le modèle dit « positif » et a raison
- FP (faux positif) : fausse alerte, il dit « positif » à tort
- FN (faux négatif) : cas manqué, il dit « négatif » à tort
- VN (vrai négatif) : le modèle dit « négatif » et a raison

	Prédit positif	Prédit négatif
Réellement positif	VP	FN
Réellement négatif	FP	VN

Exemple fil rouge (100 exemples) : VP = 40 · FP = 10 · FN = 20 · VN = 30

i. Métriques d'évaluation : Exactitude (Accuracy)

Proportion de prédictions correctes.

Dans notre exemple : **(40 + 30) / 100 = 0,70**

Le piège : les classes déséquilibrées

- 100 patients : 95 sains et 5 malades
- Un modèle qui répond toujours « sain » obtient 95 % d'accuracy
- Mais il ne détecte aucun malade (rappel = 0)
- Fiable seulement si les classes sont à peu près équilibrées

Accuracy = (VP + VN) / (VP + VN + FP + FN)

i. Métriques d'évaluation : Précision (Precision)

Parmi les cas déclarés positifs, quelle proportion l'est vraiment ?

Dans notre exemple : **40 / (40 + 10) = 0,80**

La question à se poser

- « Quand le modèle déclenche une alerte, peut-on lui faire confiance ? »
- Précision élevée : peu de fausses alertes
- À privilégier quand une fausse alerte coûte cher (spam, blocage de paiement)

Précision = VP / (VP + FP)

i. Métriques d'évaluation : Rappel (Recall)

Parmi les cas réellement positifs, quelle proportion le modèle détecte-t-il ?

Dans notre exemple : **40 / (40 + 20) = 0,67**

Le compromis précision / rappel

- Baisser le seuil de décision (0,5 → 0,3) : plus de cas déclarés positifs
- Le rappel augmente, la précision baisse
- Rappel élevé : peu de cas manqués
- À privilégier quand rater un cas coûte cher (dépistage, fraude)

Rappel = VP / (VP + FN)

i. Métriques d'évaluation : F1-score

Moyenne harmonique de la précision et du rappel.

Dans notre exemple : **2 × (0,80 × 0,67) / (0,80 + 0,67) ≈ 0,73**

Pourquoi une moyenne harmonique ?

- Elle est tirée vers la plus petite des deux valeurs
- Précision 1,0 et rappel 0,1 : moyenne simple 0,55, mais F1 ≈ 0,18
- Élevé seulement si la précision et le rappel le sont
- Utile quand les classes sont déséquilibrées

i. Métriques d'évaluation : AUC-ROC (Classification binaire)

La courbe ROC montre le compromis pour tous les seuils de décision :

Abscisse : taux de faux positifs, FPR = FP / (FP + VN)

Ordonnée : taux de vrais positifs, TPR = VP / (VP + FN), c'est le rappel

L'AUC est l'aire sous la courbe, entre 0 et 1

Intuition : l'AUC est la probabilité qu'un exemple positif tiré au hasard ait un score plus élevé qu'un exemple négatif tiré au hasard.

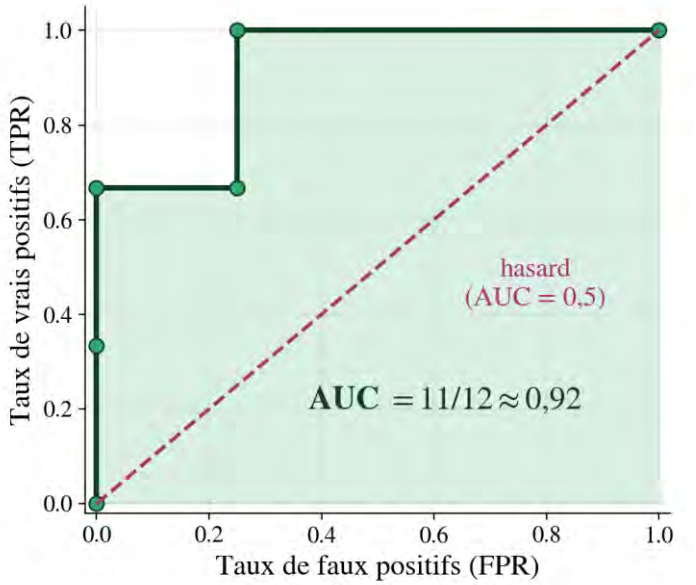
AUC	Interprétation
1,0	Classement parfait
0,5	Équivalent au hasard (la diagonale)
< 0,5	Pire que le hasard : prédictions inversées

i. Métriques d'évaluation : AUC-ROC (Classification binaire)

3 positifs (scores 0,9 ; 0,8 ; 0,4) et 4 négatifs (scores 0,7 ; 0,3 ; 0,2 ; 0,1)

Seuil	FPR	TPR
0,9	0	0,33
0,8	0	0,67
0,7	0,25	0,67
0,4	0,25	1,00
0,1	1,00	1,00

3 × 4 = 12 paires (un positif, un négatif) : le positif a le meilleur score dans 11 d'entre elles, donc AUC = 11 / 12 ≈ 0,92.



i. Métriques d'évaluation

Situation	Métrique à privilégier	Raison
Classes équilibrées, erreurs de coût égal	**Accuracy**	Simple et lisible
Une fausse alerte coûte cher (spam, blocage d'un paiement)	**Précision**	Éviter les faux positifs
Rater un cas coûte cher (dépistage médical, fraude)	**Rappel**	Éviter les faux négatifs
Classes déséquilibrées, un seul chiffre	**F1-score**	Équilibre précision et rappel
Comparer des modèles sur tous les seuils	**AUC-ROC**	Indépendant du seuil

Bonne pratique : ne jamais se limiter à un seul chiffre. On regarde la matrice de confusion, puis la précision, le rappel et le F1, et on choisit selon le coût métier des erreurs.

j. Initialisation des poids

L'initialisation des poids est un aspect critique qui peut considérablement affecter la convergence et les performances du modèle. Une mauvaise initialisation peut conduire à des gradients qui s'annulent ou explosent.

Ci-dessous : n_{in} : nombre d'entrées de la couche et n_{out} : nombre de sorties

Méthode	Distribution	Cas d'usage optimal
Xavier / Glorot	$N(0, \sqrt{(2 / (n_{in} + n_{out}))})$	Fonctions d'activation symétriques (tanh, sigmoïde)
He / MSRA	$N(0, \sqrt{(2 / n_{in})})$	Fonctions ReLU et variantes
LeCun	$N(0, \sqrt{(1 / n_{in})})$	Réseaux très profonds avec normalisation
Orthogonale	Matrices orthogonales	RNN et réseaux récurrents

k. Sous et sur apprentissage (overfitting)

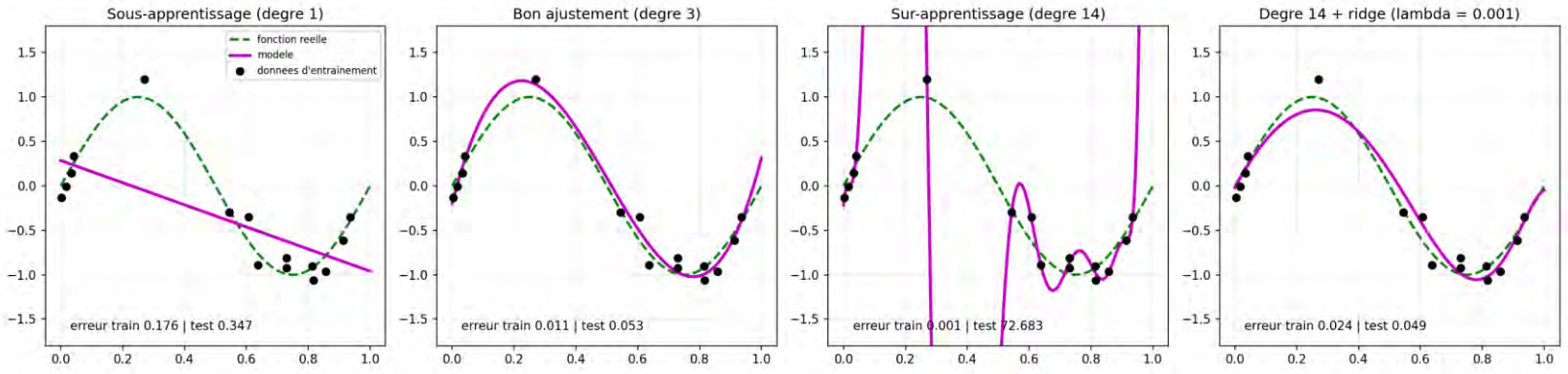
Un modèle doit **généraliser** : bien prédire des données jamais vues, pas seulement mémoriser. On sépare donc les données comme en ML :

- Train (≈ 70-80 %) : ajuster les paramètres
- Validation (≈ 10-15 %) : régler les hyperparamètres, arrêter l'entraînement
- Test (≈ 10-15 %) : mesure finale, utilisée une seule fois

Sous- et surapprentissage :

- Sous-apprentissage : mauvais partout, modèle trop simple
- Surapprentissage : excellent sur le train, mauvais sur le test : le modèle a mémorisé
- Objectif : bonne performance avec un petit écart train / test

k. Sous et sur apprentissage (overfitting)



I. Régularisation : limiter le sur-apprentissage

Principe :

La régularisation contraint le modèle pour qu'il généralise mieux, au prix d'une légère baisse de la performance sur les données d'entraînement.

On ajoute à la perte un terme qui pénalise la complexité :L_totale = L_données + λ · Pénalité(w)

Le coefficient λ règle l'équilibre :

trop petit, il ne corrige rien (sur-apprentissage) ;

trop grand, il écrase le modèle (sous-apprentissage).

On le choisit sur le jeu de validation.

Un exemple se trouve dans le script *regul.py*

3. Introduction aux frameworks

a. Pytorch

Faire le TP *3a_decouverte_pytorch.ipynb afin de découvrir Pytorch.*

b. Keras / Tensorflow

Faire le TP 3b_decouverte_keras.ipynb *afin de découvrir Keras / Tensorflow.*

4. Réseaux de neurones convolutifs (CNN)

a. Pourquoi des réseaux convolutifs ?

Les CNN se distinguent des réseaux de neurones traditionnels par leur capacité à préserver la structure spatiale des données d'entrée (images, mais aussi signaux et séries).

- Un perceptron multicouche aplatit l'image en une longue liste de pixels : il perd la notion de voisinage.
- Il compte aussi un nombre énorme de paramètres, qui croît avec la taille de l'image.
- Un CNN exploite le fait que les pixels voisins sont liés et qu'un même motif (un bord, une texture) peut apparaître n'importe où.

Un exemple chiffré : image 32×32×3, 32 cartes de sortie de 28×28

Couche	Paramètres
Dense : 3 072 entrées vers 25 088 sorties	**77 095 424**
Convolution : 32 filtres 5×5×3	**2 432**

Environ 31 700 fois moins de paramètres, pour une sortie de même taille.

b. Trois idées clés

1. Connectivité locale

Chaque neurone n'est connecté qu'à une petite région de la couche précédente (son champ réceptif), contrairement aux couches entièrement connectées.

Exemple : un neurone de Conv1 voit 5×5×3 = 75 valeurs, et non les 3 072 pixels.

2. Partage de paramètres

Les mêmes poids (un filtre) sont utilisés sur toute l'image : le même détecteur balaie chaque position.

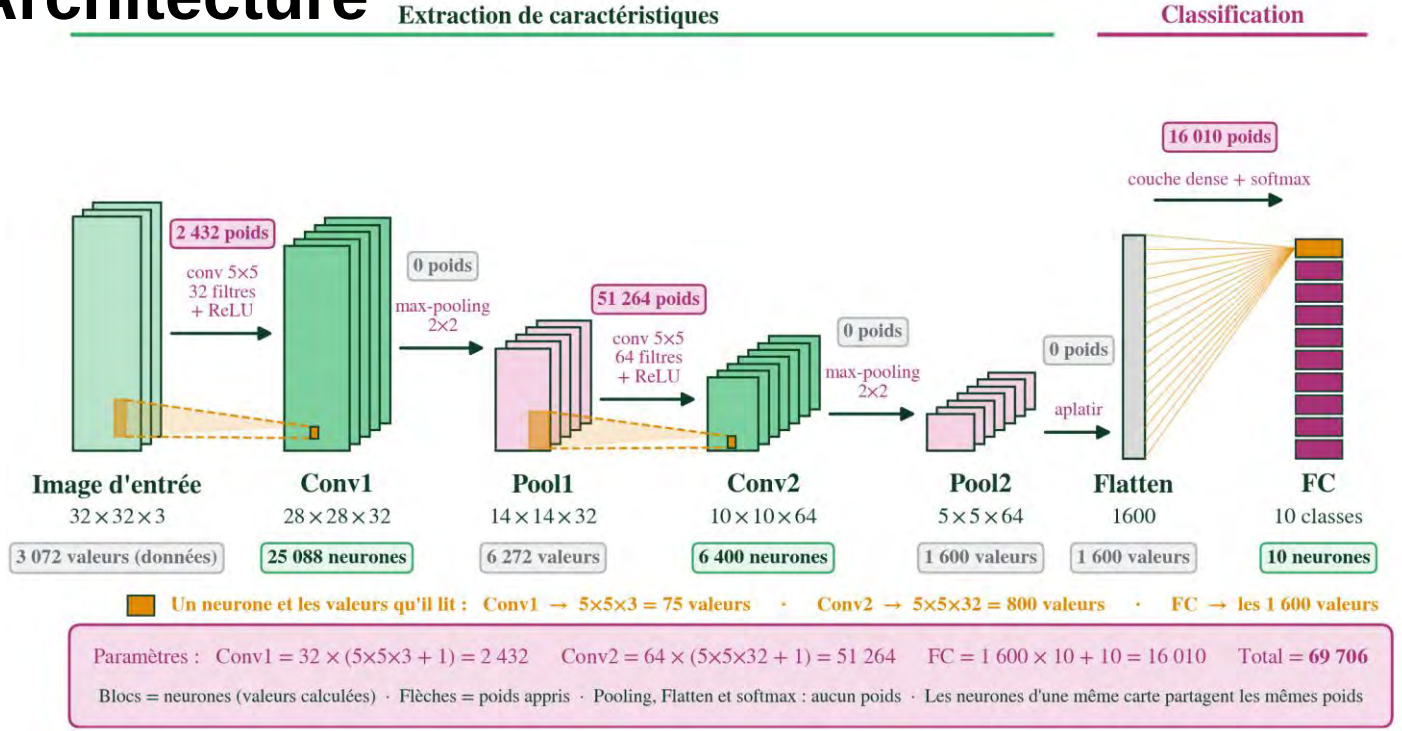
Exemple : 32 filtres × (5×5×3 + 1) = 2 432 paramètres, quelle que soit la taille de l'image.

3. Equivariance à la translation

Si un motif se déplace dans l'image, sa détection se déplace de la même façon dans la carte de caractéristiques : la même caractéristique est détectée quelle que soit sa position.

Le pooling ajoute une invariance approchée aux petits décalages.

c. Architecture



d. Les briques d'un CNN

Brique	Rôle	Paramètres appris
Convolution	Extrait des caractéristiques locales (bords, textures, motifs) avec des filtres appris	Oui : C_sortie × (k × k × C_entrée + 1)
Activation (ReLU)	Introduit la non-linéarité : sans elle, les couches s'écrasent en une seule opération linéaire	Non
Pooling	Réduit la taille des cartes (ici ÷ 2) et apporte une invariance aux petits décalages	Non
Aplatissement (Flatten)	Transforme les cartes 5×5×64 en un vecteur de 1 600 valeurs	Non
Couche dense (FC)	Classification finale (10 classes), suivie d'une softmax pour obtenir des probabilités	Oui : entrées × sorties + sorties

e. Les filtres et feature maps

Les filtres (ou noyaux de convolution) sont les éléments fondamentaux des CNN.

Chaque filtre est une petite matrice de poids qui glisse sur l'image d'entrée pour détecter des motifs spécifiques.

Mécanisme de la convolution :

Pour une image d'entrée I et un filtre F de taille k × k, la convolution produit une feature map O où chaque élément est calculé comme :

$$O(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(i+m, j+n) * F(m,n) + b$$

où b est le biais associé au filtre.

e. Les filtres et feature maps

Taille du filtre	Utilisation typique	Caractéristiques détectées
1×1	Réduction / augmentation du nombre de canaux	Combinaisons linéaires de canaux
3×3	Extraction de caractéristiques locales	Contours, textures simples
5×5	Caractéristiques moyennes	Formes, motifs plus complexes
7×7	Caractéristiques globales	Structures larges, contexte

f. La convolution

$$O(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(i+m, j+n) * F(m,n) + b$$

Pour une image N × N, un filtre k × k, un pas de 1 et sans remplissage : 0 ≤ i, j ≤ N − k, donc une carte de (N − k + 1) × (N − k + 1) cases.

Terme	Signification	Dans l'exemple : image 6×6, filtre 3×3, O(0, 1)
O(i, j)	valeur de la case (i, j) de la carte (feature map)	O(0, 1) : 2e case de la 1re ligne de la carte
(i, j)	coin supérieur gauche de la fenêtre dans l'image	i = 0, j = 1 : la fenêtre commence à la colonne 1
Σ Σ (m, n de 0 à k − 1)	on parcourt les k × k cases de la fenêtre	k = 3 : 9 termes à additionner
I(i + m, j + n)	pixel de l'image situé sous la case (m, n) du filtre	colonnes 1, 2, 3 : 10, 10, 0 sur chaque ligne
F(m, n)	poids du filtre en (m, n)	1, 0, −1 sur chaque ligne
× puis Σ	multiplier terme à terme, puis additionner	3 × (10×1 + 10×0 + 0×(−1)) = 30
+ b	biais du filtre, ajouté une seule fois	**b = 0, donc O(0, 1) = 30**

g. TP4_visualisation_CNN

Essayer de compléter le *TP4_visualisation_CNN.ipynb* intégralement tout en analysant en détail les codes fournis.

5. Traitement du langage naturel (NLP)

a. Représentation du texte

Le traitement du langage naturel en Deep Learning nécessite de convertir le texte en représentations numériques que les réseaux de neurones peuvent manipuler.

Plusieurs approches existent, chacune avec ses avantages et inconvénients.

Méthode	Description	Avantages	Inconvénients
One-hot encoding	Vecteur binaire avec un 1 à la position du mot	Simple, pas d'ambiguïté	Haute dimensionnalité, pas de relation sémantique
Bag of Words	Comptage des occurrences	Capture la fréquence	Perd l'ordre des mots
TF-IDF	Fréquence pondérée par la rareté du terme	Importance relative des mots	Pas de sémantique
Word Embeddings	Vecteurs denses de dimension fixe	Capture la sémantique, dimension réduite	Nécessite beaucoup de données d'entraînement

a. Représentation du texte

Les représentations traditionnelles comme le one-hot encoding souffrent de la "malédiction de la dimensionnalité" et ne capturent pas les relations sémantiques entre les mots.

Par exemple, "chat« et "chien" sont traités comme complètement indépendants, alors qu'ils partagent des propriétés sémantiques (animaux domestiques).

Les word embeddings résolvent ces problèmes en apprenant des représentations denses où des mots sémantiquement similaires ont des vecteurs proches dans l'espace vectoriel. Cette proximité peut être mesurée par **la similarité cosinus** ou **la distance euclidienne**.

b. Word Embeddings

Les word embeddings modernes utilisent des techniques d'apprentissage non supervisé pour créer des représentations vectorielles riches en information sémantique.

Word2Vec utilise l'hypothèse distributionnelle : "des mots qui apparaissent dans des contextes similaires ont des significations similaires".

L'entraînement optimise les vecteurs pour que la probabilité des mots contextuels soit maximisée, créant naturellement des représentations où des mots sémantiquement liés ont des vecteurs proches.

Les opérations vectorielles sur ces embeddings capturent des relations sémantiques complexes (analogies comme « Paris - France + Tunis Tunisie").

Lien du papier : https://arxiv.org/pdf/1301.3781

c. TP5_word_Embeddings

Essayer de compléter le *TP5_word_Embeddings.ipynb* intégralement tout en analysant en détail les codes fournis.

6. Réseaux de neurones récurrents (RNN)

a. Pourquoi le RNN ?

Beaucoup de données sont des **séquences** : l'ordre des éléments porte du sens.

Longueur variable : une phrase a 3 mots ou 300 ; un réseau classique attend une entrée de taille fixe.

L'ordre compte : « le chien mord l'homme » et « l'homme mord le chien » ont les mêmes mots mais pas le même sens. Un Bag of Words ne les distingue pas.

Dépendances à distance : dans « Les chats que mon voisin a adoptés la semaine dernière dorment », le verbe s'accorde avec « chats », 9 mots plus tôt.

Type de données	Exemple
Texte	phrases, avis, courriels
Parole	signal audio échantillonné
Séries temporelles	météo, cours de bourse, capteurs
Musique	suite de notes
Biologie	séquence d'ADN

a. Pourquoi le RNN ?

Les modèles déjà vus ne suffisent pas.

Réseau dense (MLP)

Entrée de taille fixe, et un poids différent pour chaque position : ce qui est appris sur le mot n°3 ne sert pas pour le mot n°7.

CNN 1D

Ne regarde qu'une fenêtre de largeur fixe (k mots) : aucune dépendance au-delà.

Il faut un modèle qui lit la séquence élément par élément, avec le même mécanisme à chaque position, et qui garde une trace de ce qu'il a déjà lu.

b. Idée clé : lire en gardant une mémoire

Lire une phrase mot à mot, en mettant à jour un résumé mental :

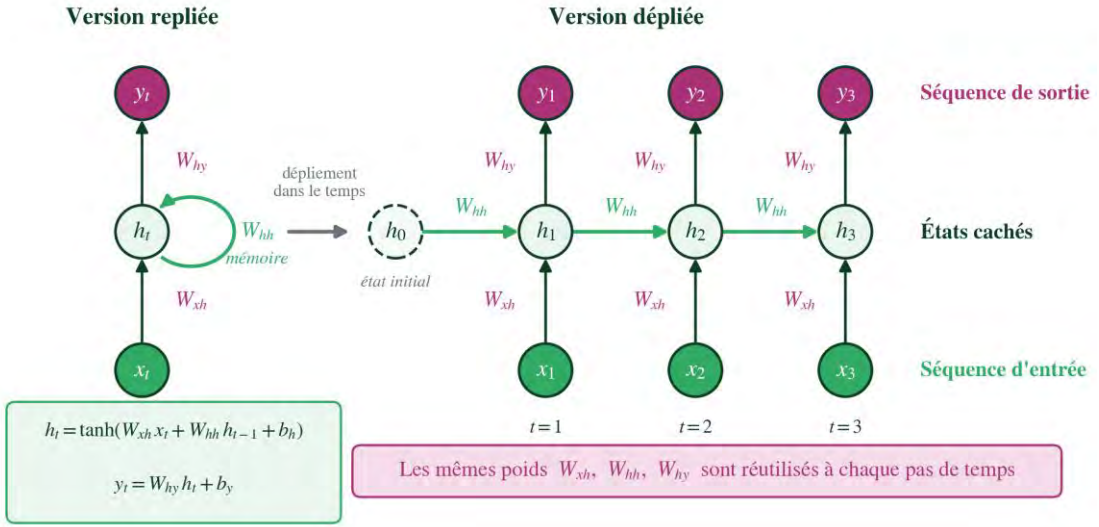
Mot lu	Ce que je retiens (mon résumé)
« Le »	un nom va suivre
« chat »	le sujet est un chat (masculin, singulier)
« dort »	la phrase est complète : un chat qui dort

- À chaque mot, mon nouveau résumé dépend **du mot lu** et **de mon résumé précédent**.
- Le résumé a toujours la même taille, quelle que soit la longueur de la phrase.

c. Le principe

h_t = f(x_t , h_(t−1))

- x_t : l'élément lu au pas t
- h_(t−1) : la mémoire avant de lire x_t
- h_t : la nouvelle mémoire, un vecteur de nombres



d. Entrainer un RNN : la rétropropagation dans le temps

- On **déplie** le réseau sur la longueur T de la séquence : on obtient un réseau profond de T couches qui partagent les mêmes poids.
- La perte totale est la somme des pertes de chaque pas :
  - **L = L_1 + … + L_T**.
- On rétropropage du dernier pas vers le premier. Le gradient d'un poids partagé est la **somme de ses contributions à chaque pas**.
- Puis mise à jour habituelle : poids ← poids − taux × gradient.

Remonter d'un pas dans le temps

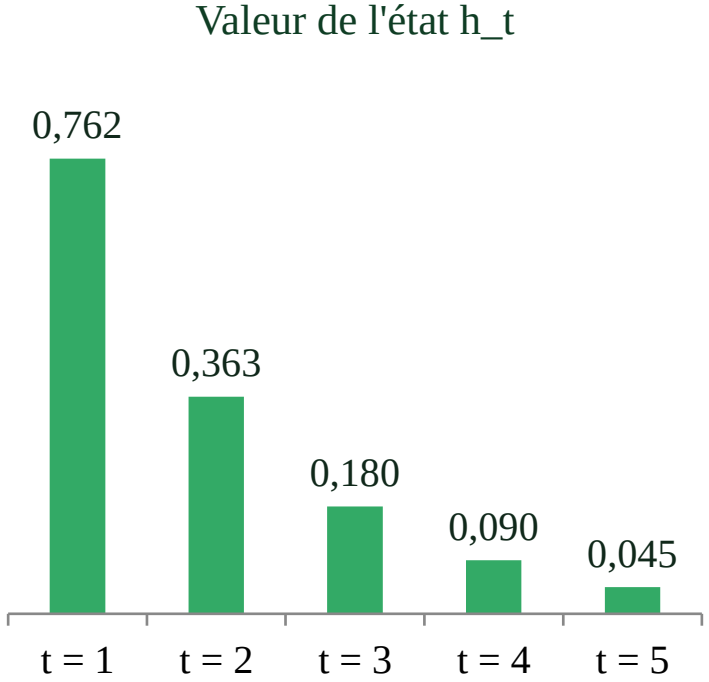
$$\frac{\partial h_t}{\partial h_{t-1}} = \mathrm{diag}\left(1 - h_t^2\right) W_{hh}$$

- Pour remonter d'un pas, on multiplie par cette matrice.
- Pour remonter de **k pas**, on la multiplie **k fois**.
- Pour « le chat dort » (T = 3), le gradient atteint le premier mot après 2 multiplications.

e. Le problème des dépendances longues

- Cas le plus simple : un état à une seule valeur, **h_t = tanh(x_t + 0,5 · h_(t−1))**
- On présente une information au premier pas (x_1 = 1), puis rien (x_t = 0).
- L'information du premier pas est **divisée par environ 2 à chaque pas** : après 5 pas, il en reste 6 %.
- Un RNN simple a donc une **mémoire courte** : il peine à relier deux éléments éloignés.

Cette dilution est la limite du RNN simple, qui conduit au problème des dépendances longues.



e. Le problème des dépendances longues

- Facteur < 1 : le gradient **s'évanouit**. Le début de la séquence n'influence plus l'apprentissage.
- Facteur > 1 : le gradient **explose**. L'apprentissage devient instable.

Pas remontés k	W_hh = 0,5	W_hh = 1,5
5	0,031	7,6
10	0,00098	57,7
20	**0,00000095**	**3 325**

f. LSTM et GRU : une mémoire contrôlée par des portes

Les réseaux **LSTM (Long Short-Term Memory)** et **GRU (Gated Recurrent Unit)** sont des variantes avancées des RNN conçues pour résoudre le problème du gradient évanescent et permettre l'apprentissage de dépendances à long terme.

- Les LSTM résolvent le problème du gradient évanescent grâce à leur architecture de portes qui permet un flux direct d'information à travers l'état de cellule. Les trois portes principales sont :
  - Porte d'oubli : Décide quelles informations supprimer de l'état de cellule
  - Porte d'entrée : Décide quelles nouvelles informations stocker
  - Porte de sortie : Détermine quelles parties del'état de cellule utiliser pour la sortie
- Les GRU simplifient cette architecture en fusionnant certaines portes, réduisant le nombre de paramètres tout en conservant la capacité de mémorisation à long terme. En pratique, les GRU sont souvent aussi performants que les LSTM sur de nombreuses tâches.

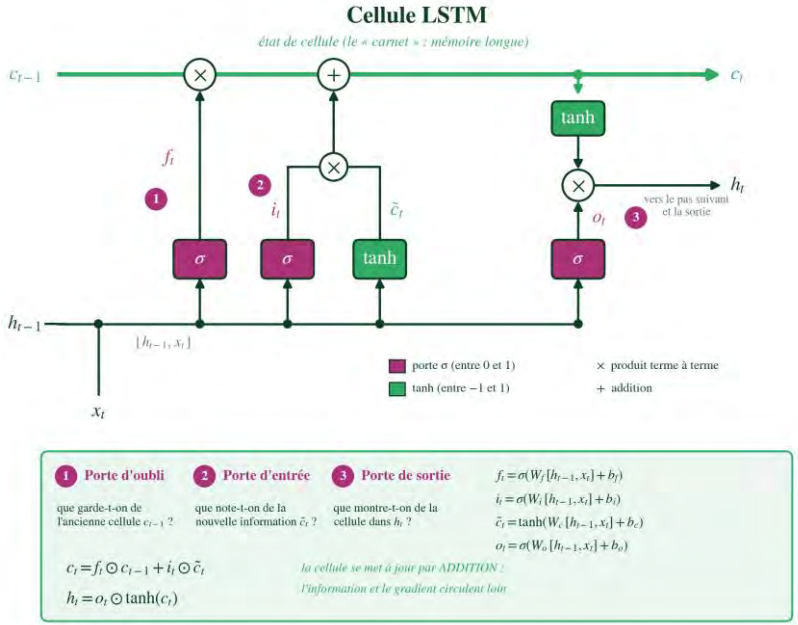
f. LSTM et GRU : une mémoire contrôlée par des portes

« Les LSTM ajoutent une piste de mémoire distincte, appelée état de cellule, qui fonctionne en parallèle de l'état caché standard. La différence est structurelle : **l'information dans l'état de cellule circule via des additions plutôt que des multiplications répétées, elle ne se dégrade donc pas de la même manière**.

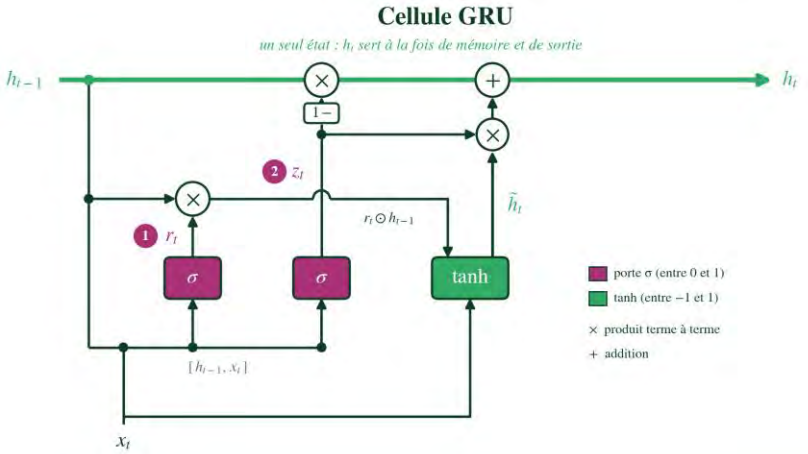
Pour contrôler ce qui entre et sort de cette mémoire, les LSTM utilisent des portes. **Chaque porte est un petit réseau de neurones qui apprend à produire des valeurs entre 0 et 1, agissant comme un filtre**. Une valeur proche de 0 bloque l'information ; une valeur proche de 1 la laisse passer. Le réseau apprend pendant l'entraînement quels motifs valent la peine d'être mémorisés sur de longues séquences et lesquels peuvent être éliminés.

Ce mécanisme de portes distingue les LSTM des RNN classiques. Au lieu de forcer toute l'information à suivre le même chemin, les LSTM peuvent préserver sélectivement le contexte sur des centaines de pas de temps. » (Source Datacamp)

f. LSTM et GRU : une mémoire contrôlée par des portes



f. LSTM et GRU : une mémoire contrôlée par des portes



1 Porte de réinitialisation

que ressort-on de l'ancien état pour proposer un candidat $\tilde{h}_t$?

2 Porte de mise à jour

quelle part de l'ancien état remplace-t-on par le candidat ?

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$
$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$
$$\tilde{h}_t = \tanh(W[r_t \odot h_{t-1}, x_t] + b)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

une seule porte, z_t, règle le mélange entre l'ancien état $(1 - z_t)$ et le nouveau (z_t)

pas de cellule séparée : 2 portes au lieu de 3

g. TP6_RNN

Essayer de compléter le *TP6_RNN.ipynb* intégralement tout en analysant en détail les codes fournis.

7. Transformers et Attention (2017)

a. Pourquoi aller au-delà des RNN ?

Les RNN ont fait progresser le traitement du texte, mais ils butent sur trois limites :

Limite des RNN	Ce que cela implique	Réponse de l'attention
Lecture séquentielle	Pour calculer h_t, il faut avoir calculé h_(t−1) : impossible de traiter tous les mots en parallèle, l'entraînement est lent.	Tous les mots sont lus en même temps.
Mémoire qui s'efface	Même avec un LSTM, l'information d'un mot lointain traverse tous les pas intermédiaires : un chemin de longueur n.	Chaque mot accède directement à n'importe quel autre : un chemin de longueur 1.
Goulot d'étranglement	En traduction (encodeur-décodeur), toute la phrase source est résumée dans un seul vecteur de taille fixe.	On garde un vecteur par mot, pas un seul résumé.

b. Exemple concret

Un problème concret : à quoi renvoie « il » ?

Phrase	Mot à interpréter	Il renvoie à
« L'animal n'a pas traversé la rue parce qu'il était trop fatigué »	**il**	**l'animal**
« L'animal n'a pas traversé la rue parce qu'elle était trop large »	**elle**	**la rue**

- Les deux phrases ne diffèrent que par un mot, et pourtant **la référence change**.
- Pour interpréter « il » ou « elle », le modèle doit aller **chercher dans la phrase le mot pertinent**, quelle que soit sa distance.
- Il doit aussi savoir **quels mots ignorer** : « traversé » ou « parce que » n'aident pas à résoudre la référence.

Comment un mot peut-il « regarder » les autres mots et choisir lesquels comptent ? C'est ce que fait l'attention.

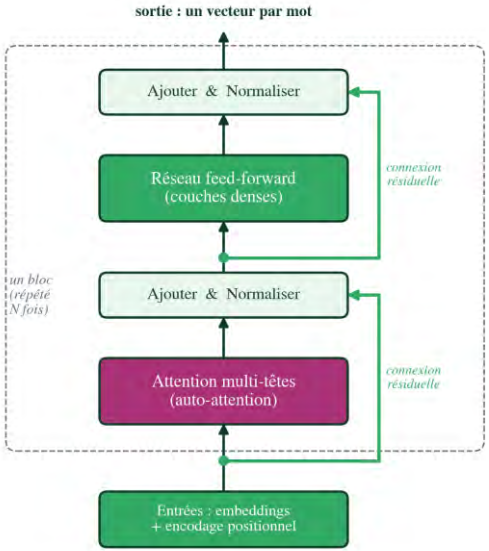
c. Principes fondamentaux des Transformers

Les Transformers représentent un changement de paradigme fondamental dans le traitement séquentiel :

- **Mécanisme d'attention** : Chaque position peut directement accéder à toutes les autres positions, éliminant les problèmes de gradient évanescent des RNN. L'attention pondérée permet de se concentrer sur les éléments pertinents.
- **Parallélisation** : Contrairement aux RNN qui traitent séquentiellement, les Transformers peuvent traiter tous les tokens simultanément, accélérant drastiquement l'entraînement sur GPU.
- **Multi-head attention** : Permet au modèle de capturer différents types de relations (syntaxiques, sémantiques) simultanément en utilisant plusieurs "têtes" d'attention parallèles.
- **Encodage positionnel** : Compense l'absence d'ordre séquentiel naturel en injectant des informations de position via des fonctions sinusoïdales.

Cette architecture est devenue la base de modèles révolutionnaires comme BERT, GPT, et leurs successeurs.

d. Le bloc Transformer



Partie	Rôle
Attention multi-têtes	les mots échangent de l'information entre eux
Ajouter & Normaliser	recopie l'entrée (connexion résiduelle) et stabilise les valeurs (normalisation)
Feed-forward	transforme chaque mot séparément, avec le même petit réseau pour tous les mots
Répété N fois	les blocs s'empilent : 6 dans le Transformer d'origine, 12 dans BERT-base

8. Retour sur les problèmes courants et techniques d’optimisation

a. TP7_diagnostic_optimisation

Voir le document : *Diagnostic_et_optimisation_des_modeles.docx* et le *TP7_diagnostic_optimisation.ipynb*.

9. Ressources complémentaires

a. Liste

- https://course.fast.ai/ site de reference pour apprendre le deep learning en codant

- Chaines Youtube :
  - https://www.youtube.com/watch?v=XUFLq6dKQok&list=PLO_fdPEVlfKoanjvTJbIbd9V5d9Pzp8Rw (Machine Learnia – introduction au dep learning “à la main”
  - https://www.youtube.com/@freecodecamp (Freecode camp avec plusieurs formations completes au Deep Learning et bien plus encore)
  - https://www.youtube.com/@lexfridman/videos
  - https://www.youtube.com/@3blue1brown/playlists
  - https://www.youtube.com/watch?v=CsQNF9s78Nc&list=PLxzM9a5lhAun7pZED3Q53b9Zd8YP5rack&index=13

- Deep Learning from Scratch (O’reilly), disponible version papier ou numérique