CHAPITRE RNN · FORMATION DEEP LEARNING PAR LA PRATIQUE

Intuition sur les RNN

Comprendre pas à pas ce que fait un réseau de neurones récurrent

Ce document suit, étape par étape, un petit réseau de neurones récurrent (RNN) qui lit la phrase « le chat dort » **mot par mot** et, à chaque mot, prédit le mot suivant. Chaque étape est expliquée avec des exemples chiffrés, vérifiés avec PyTorch. Trois questions y reçoivent une réponse :

- pourquoi un réseau classique **ne convient pas** aux séquences (étapes 1 et 2) ;
- comment un réseau peut garder une **mémoire** avec un nombre de paramètres fixe (étapes 3 à 5) ;
- pourquoi cette mémoire **s'efface** sur les longues séquences, et comment y remédier (étapes 8 à 10).

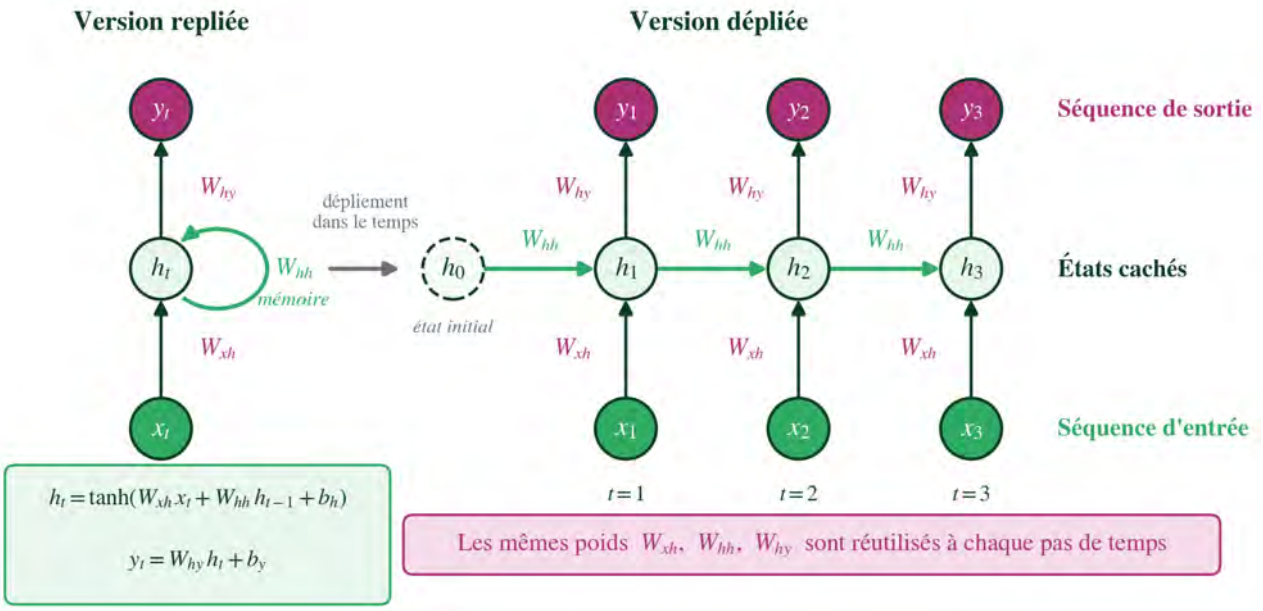

Figure 1 – Un réseau récurrent, replié (à gauche) puis déplié dans le temps (à droite)

> **Notations** : x_t est l'entrée au pas t, h_t l'état caché (la mémoire), y_t la sortie, h_0 l'état initial. Les trois matrices de poids W_xh, W_hh et W_hy sont **les mêmes à tous les pas de temps**.

Étape 0 – Qu'est-ce qu'une séquence ?

Une **séquence** est une suite d'éléments dont l'**ordre** porte du sens : les mots d'une phrase, les valeurs d'un capteur au fil du temps, les notes d'une mélodie. Chaque élément est lu l'un après l'autre, et on parle de **pas de temps** même quand il ne s'agit pas de temps (la position d'un mot dans une phrase, par exemple).

Type de données	Exemple de séquence
Texte	les mots d'une phrase, d'un avis, d'un courriel
Parole	le signal audio échantillonné
Séries temporelles	météo, cours de bourse, mesures de capteurs
Musique	une suite de notes
Biologie	une séquence d'ADN

Étape 1 – Pourquoi un réseau classique ne suffit pas

Trois difficultés distinguent les séquences des données vues jusqu'ici :

- **La longueur varie** : une phrase compte 3 mots ou 300. Un réseau dense attend une entrée de taille fixe.
- **L'ordre compte** : « le chien mord l'homme » et « l'homme mord le chien » contiennent les mêmes mots, mais pas le même sens. Un Bag of Words, qui compte les mots sans tenir compte de leur place, ne les distingue pas.
- **Les dépendances sont parfois lointaines** : dans « Les chats que mon voisin a adoptés la semaine dernière dorment », le verbe s'accorde avec « chats », situé 9 mots plus tôt.

Approche	Limite face à une séquence
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.
Bag of Words, TF-IDF	Perdent complètement l'ordre des mots.
CNN 1D	Ne regarde qu'une fenêtre de largeur fixe (k mots) : aucune dépendance au-delà.

> **Ce qu'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.

Étape 2 – L'idée clé : lire en gardant une mémoire

Quand vous lisez une phrase, vous ne repartez pas de zéro à chaque mot : vous mettez à jour un **résumé mental**. Voici ce que vous retenez au fil de « Le chat dort » :

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, le nouveau résumé dépend de **deux choses** : le mot qu'on vient de lire, et le résumé précédent. Un RNN fait exactement cela, avec un résumé qui est un **vecteur de nombres**, appelé **état caché** :

> **h_t = f(x_t , h_(t−1))**
>
> Le nouvel état h_t dépend de l'entrée x_t et de l'état précédent h_(t−1). L'état a **toujours la même taille**, quelle que soit la longueur de la phrase.

Étape 3 – Le schéma, élément par élément

3.1 Version repliée et version dépliée

À gauche de la figure 1, le réseau est **replié** : un seul neurone caché dont la sortie revient sur lui-même (la boucle W_hh). C'est cette boucle qui crée la mémoire. À droite, on **déplie** la boucle dans le temps : on obtient une chaîne, un exemplaire du même réseau par élément de la séquence, chaque exemplaire passant son état au suivant.

3.2 Le rôle de chaque élément

Les dimensions sont celles de l'exemple de l'étape 5 : un vocabulaire de 4 mots (entrée d = 4), un état caché de 2 valeurs (n = 2) et une sortie de 4 probabilités (m = 4).

Élément	Rôle	Dimension	Dans l'exemple
x_t	Entrée au pas t : le mot lu, encodé en vecteur (ici one-hot)	d	4 valeurs
h_t	État caché : résumé de ce qui a été lu jusqu'à t (la mémoire)	n	2 valeurs
h_0	État initial, avant toute lecture (souvent des zéros)	n	(0 ; 0)
y_t	Sortie au pas t	m	4 probabilités
W_xh	Transforme l'entrée en contribution à l'état	n × d	2×4 = 8 poids
W_hh	Réinjecte l'état précédent : c'est la boucle récurrente	n × n	2×2 = 4 poids
W_hy	Transforme l'état en sortie	m × n	4×2 = 8 poids
b_h, b_y	Biais de l'état et de la sortie	n et m	2 + 4 = 6

> **Poids partagés** : les mêmes W_xh, W_hh et W_hy servent à chaque pas de temps. Le réseau apprend **une seule manière de lire un mot et de mettre à jour sa mémoire**, qu'il applique à toutes les positions. Le nombre de paramètres (ici 8 + 4 + 8 + 6 = **26**) ne dépend donc pas de la longueur de la séquence : il est le même pour 3 mots ou 3 000.

Étape 4 – Les équations

À chaque pas de temps t, le réseau applique les deux mêmes équations :

> **h_t = tanh(W_xh x_t + W_hh h_(t−1) + b_h)**
>
> **y_t = softmax(W_hy h_t + b_y)**

Partie de l'équation	Rôle
W_xh x_t	ce que le mot courant apporte à l'état
W_hh h_(t−1)	ce que l'on garde du passé : c'est la mémoire
b_h	un décalage appris, comme dans toute couche dense
tanh	comprime le résultat entre −1 et 1 et apporte la non-linéarité (sans elle, tout se réduirait à un calcul linéaire)
softmax(W_hy h_t + b_y)	transforme l'état en probabilités sur les mots possibles : positives, de somme 1

Au premier pas, il n'y a pas d'état précédent : on utilise l'état initial h_0, en général un vecteur de zéros. La sortie y_t n'est pas toujours nécessaire à chaque pas : pour classer un avis, on n'utilise que le dernier état.

Étape 5 – L'exemple pas à pas : lire « le chat dort »

5.1 Mise en place

- **Tâche** : après chaque mot lu, prédire le mot suivant.
- **Vocabulaire** : le, chat, dort, mange. Chaque mot est un vecteur **one-hot** : le = (1, 0, 0, 0), chat = (0, 1, 0, 0), dort = (0, 0, 1, 0), mange = (0, 0, 0, 1).
- **Dimensions** : entrée d = 4, état caché n = 2, sortie m = 4. Biais nuls, h_0 = (0 ; 0).
- **Les poids sont choisis à la main** : le réseau n'est pas entraîné. L'exemple montre le mécanisme, pas la qualité des prédictions.

W_xh (2 × 4) : chaque colonne est ce qu'apporte un mot à l'état quand il est lu.

État \ mot	le	chat	dort	mange
h_1	0,5	−0,3	0,2	0,1
h_2	0,1	0,8	−0,4	0,6

W_hh (2 × 2) : comment l'état précédent influence le nouvel état.

	h_1 (t−1)	h_2 (t−1)
h_1	0,5	0,1
h_2	−0,2	0,4

W_hy (4 × 2) : comment l'état se transforme en scores sur les mots.

Mot	h_1	h_2
le	0,2	−0,1
chat	0,4	0,3
dort	−0,3	0,5
mange	0,1	0,2

5.2 Pas 1 : lire « le »

Étape de calcul	Résultat
W_xh x_1 : la colonne « le » de W_xh	(0,5 ; 0,1)
W_hh h_0 : l'état initial est nul	(0 ; 0)
z_1 = somme des deux (+ biais nul)	(0,5 ; 0,1)
h_1 = tanh(z_1)	(0,462 ; 0,100)
u_1 = W_hy h_1 : un score par mot	(0,082 ; 0,215 ; −0,089 ; 0,066)
exponentielles de u_1 (somme 4,309)	(1,086 ; 1,240 ; 0,915 ; 1,068)
y_1 = softmax(u_1) : probabilités de (le, chat, dort, mange)	(0,252 ; 0,288 ; 0,212 ; 0,248)

Après « le », le mot le plus probable est « chat » (0,288).

5.3 Pas 2 : lire « chat »

C'est ici que la **mémoire** intervient : la contribution de l'état précédent n'est plus nulle.

Étape de calcul	Résultat
W_xh x_2 : la colonne « chat » de W_xh	(−0,3 ; 0,8)
W_hh h_1 = (0,5×0,462 + 0,1×0,100 ; −0,2×0,462 + 0,4×0,100)	(0,241 ; −0,053)
z_2 = somme des deux	(−0,059 ; 0,747)
h_2 = tanh(z_2)	(−0,059 ; 0,634)
u_2 = W_hy h_2	(−0,075 ; 0,167 ; 0,334 ; 0,121)
exponentielles de u_2 (somme 4,634)	(0,928 ; 1,181 ; 1,397 ; 1,128)
y_2 = softmax(u_2)	(0,200 ; 0,255 ; 0,301 ; 0,243)

Après « le chat », le mot le plus probable est « dort » (0,301). Les calculs sont faits avec toute la précision ; les valeurs affichées sont arrondies à 3 décimales (d'où un écart de 0,001 possible si on refait le calcul à la main avec les valeurs arrondies).

5.4 Pas 3 : lire « dort »

Étape de calcul	Résultat
W_xh x_3 : la colonne « dort » de W_xh	(0,2 ; −0,4)
W_hh h_2 = (0,5×(−0,059) + 0,1×0,634 ; −0,2×(−0,059) + 0,4×0,634)	(0,034 ; 0,265)
z_3 = somme des deux	(0,234 ; −0,135)
h_3 = tanh(z_3)	(0,230 ; −0,134)
u_3 = W_hy h_3	(0,059 ; 0,052 ; −0,136 ; −0,004)
exponentielles de u_3 (somme 3,983)	(1,061 ; 1,053 ; 0,873 ; 0,996)
y_3 = softmax(u_3)	(0,266 ; 0,264 ; 0,219 ; 0,250)

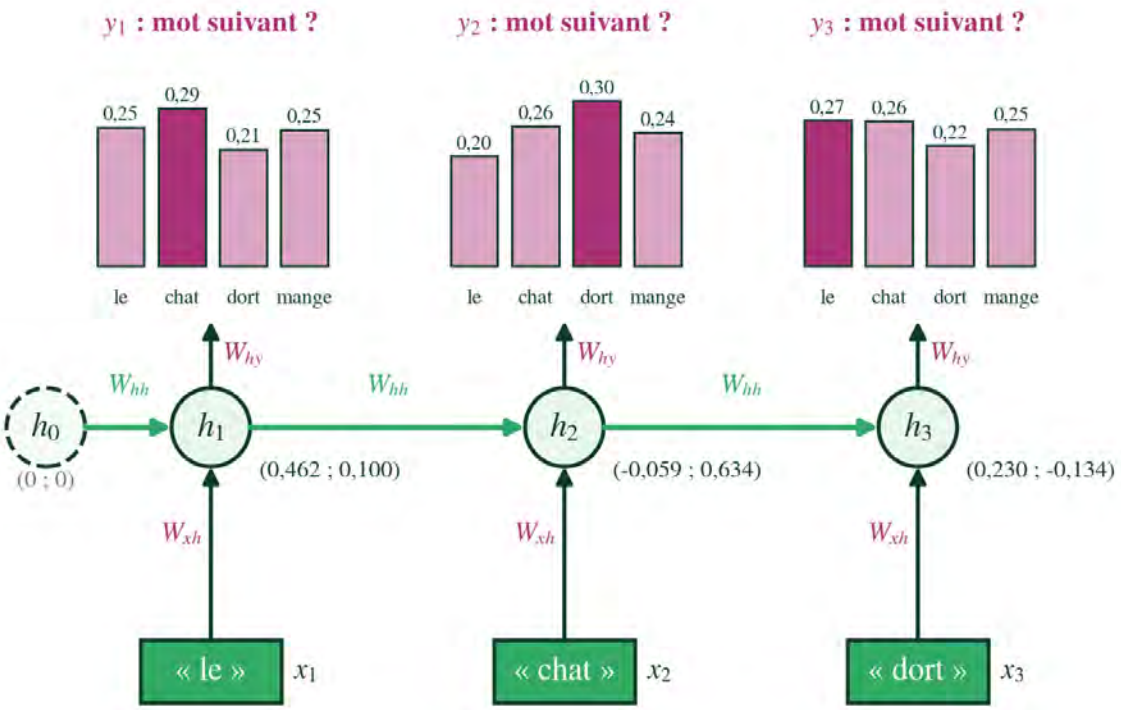

Figure 2 – Les trois pas de l'exemple : l'état caché et les probabilités du mot suivant

5.5 Ce que l'on observe

- **L'état h_2 contient une trace de « le »**. Pour le voir, lisons « chat » **sans** avoir lu « le » (h_0 = 0) : on obtient h = (−0,291 ; 0,664) au lieu de (−0,059 ; 0,634), et la probabilité de « dort » passe de 0,301 à 0,331. La différence vient uniquement de la mémoire du mot précédent.
- **Les mêmes poids sont utilisés aux trois pas** : W_xh, W_hh et W_hy n'ont pas changé d'un pas à l'autre. Seuls changent l'entrée et l'état.
- **Les probabilités restent proches de 0,25** (quatre mots équiprobables) car les poids ne sont pas entraînés. Après un entraînement sur de nombreuses phrases, les poids seraient ajustés pour que, après « le chat », la probabilité de « dort » ou « mange » soit nettement plus élevée que celle de « le ».
- **Ici, la sortie est utilisée à chaque pas** (prédire le mot suivant). Pour classer l'avis d'un client, on n'utiliserait que la sortie du dernier pas.

Étape 6 – Les grandes familles de tâches

Selon le problème, on utilise les entrées et les sorties à des moments différents. Dans notre exemple (prédire le mot suivant), on lit une sortie à chaque pas :

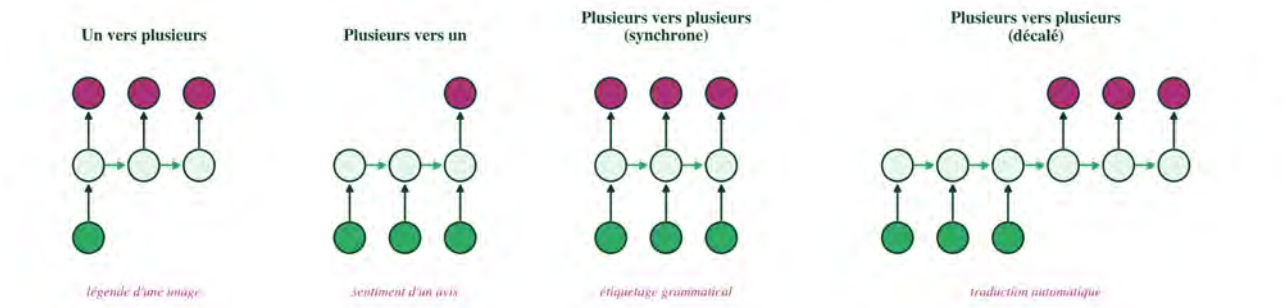

Figure 3 – Les quatre familles de tâches

Famille	Entrée vers sortie	Exemple
Un vers plusieurs	une image vers une suite de mots	légende d'une image
Plusieurs vers un	une suite de mots vers une étiquette	sentiment d'un avis (positif ou négatif)
Plusieurs vers plusieurs, synchrone	une sortie par élément lu	étiquetage grammatical, mot suivant
Plusieurs vers plusieurs, décalé	une suite vers une autre, de longueur différente	traduction automatique

Étape 7 – Entraîner un RNN : la rétropropagation dans le temps

L'entraînement suit le principe habituel (passe avant, perte, gradients, mise à jour), avec une particularité : le réseau est **déplié** sur toute la séquence.

- Le réseau déplié est un réseau **profond de T couches** (T = nombre de pas), dont toutes les couches 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**. Comme un poids est utilisé à chaque pas, son gradient est la **somme de ses contributions** à tous les pas.
- Puis la mise à jour habituelle : poids ← poids − taux d'apprentissage × gradient.

Pour remonter d'un pas de temps, on multiplie le gradient par la matrice :

> **$\partial h_t / \partial h_{(t-1)} = \mathrm{diag}(1 - h_t^2) \cdot W_{hh}$**
>
> Remonter de k pas revient à multiplier k fois par une matrice de ce type. Pour « le chat dort » (T = 3), le gradient atteint le premier mot après 2 multiplications.

Étape 8 – Pourquoi la mémoire s'efface

8.1 L'information se dilue

Prenons le cas le plus simple : un état à une seule valeur, avec $h_t = \tanh(x_t + 0{,}5 \cdot h_{(t-1)})$. On présente une information au premier pas ($x_1 = 1$), puis plus rien ($x_t = 0$).

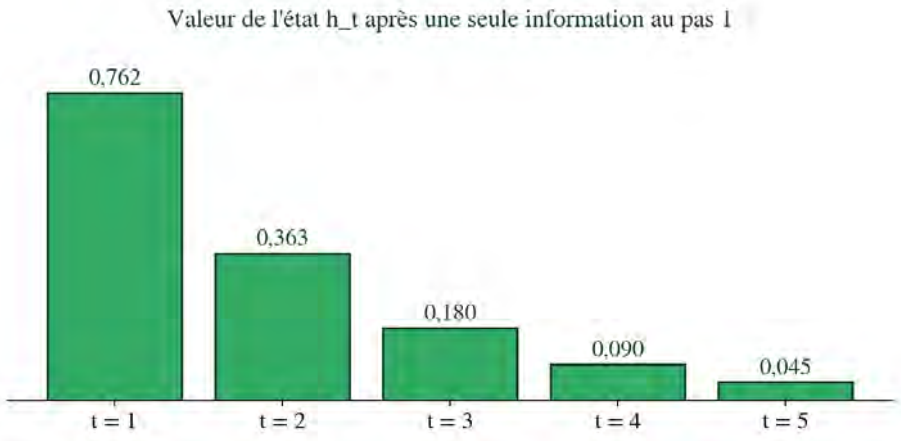

Figure 4 – L'information du premier pas, au fil des pas

L'information du premier pas est **divisée par environ 2 à chaque pas** : après 5 pas, il en reste 6 % (0,045 contre 0,762). Un RNN simple a donc une **mémoire courte** : il peine à relier deux éléments éloignés.

8.2 Gradients qui s'évanouissent ou explosent

La rétropropagation multiplie le gradient à chaque pas. Dans un modèle simplifié où l'on multiplie k fois par un même nombre W_hh :

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

- **Facteur inférieur à 1** : le gradient **s'évanouit**. Le début de la séquence n'influence plus l'apprentissage.
- **Facteur supérieur à 1** : le gradient **explose**. L'apprentissage devient instable.
- La dérivée de tanh est toujours inférieure ou égale à 1 : elle aggrave la diminution.

8.3 Une conséquence concrète

> « Les chats que mon voisin a adoptés la semaine dernière … » : pour écrire « **dorment** » et non « dort », le réseau doit se souvenir de « **chats** » (pluriel), 9 mots plus tôt. Avec un gradient presque nul sur 9 pas, il n'apprend pas cette règle.

8.4 Les remèdes

- **Contre l'explosion** : l'**écrêtage des gradients** (gradient clipping), qui plafonne leur norme.
- **Contre l'évanouissement** : des architectures à **portes**, le LSTM et le GRU (étapes 9 et 10).

Étape 9 – LSTM : une mémoire que l'on édite

9.1 Le problème à résoudre

Un RNN simple **réécrit entièrement** sa mémoire à chaque pas : h_t = tanh(W_xh x_t + W_hh h_(t−1) + b_h). L'ancien état traverse à chaque fois une multiplication par W_hh et un tanh. Après 10 ou 20 pas, l'information du début a été transformée et diluée 10 ou 20 fois (il en reste 6 % après 5 pas dans l'exemple de l'étape 8). À l'entraînement, le gradient suit le même chemin à l'envers et subit les mêmes multiplications : il s'évanouit.

9.2 L'idée : deux mémoires, dont un carnet que l'on édite

Au lieu de tout réécrire, le LSTM tient un **carnet de notes**, la **cellule c_t**, qu'il **modifie légèrement** à chaque pas : il **efface** un peu de l'ancien contenu et il **ajoute** un peu d'information nouvelle. Il garde aussi un état **h_t**, ce que le réseau laisse voir au reste du modèle.

Mémoire	Rôle	Analogie
c_t	mémoire interne, longue, protégée	le carnet de notes
h_t	ce que le réseau montre à ce pas (sortie et entrée du pas suivant)	ce que je dis à voix haute

Ce sont trois **portes** qui règlent ce qui entre et ce qui sort. Une porte est un nombre entre 0 et 1, calculé par une sigmoïde à partir de h_(t−1) et x_t : 0 bloque tout, 1 laisse tout passer. Le réseau apprend à ouvrir ou fermer chaque porte selon ce qu'il lit.

9.3 Le schéma de la cellule

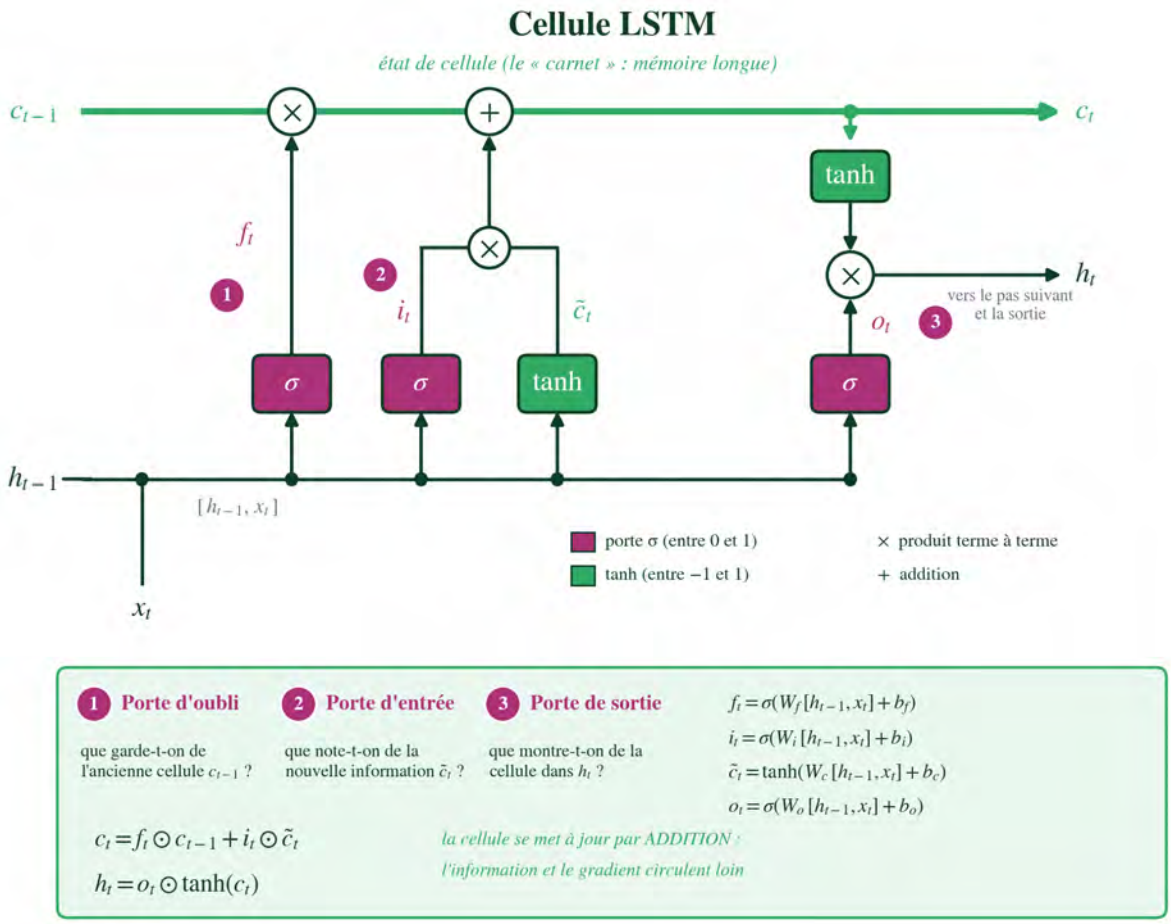

Figure 5 – Une cellule LSTM : ligne de cellule en haut, trois portes numérotées, bus [h_(t−1), x_t] en bas

Comment le lire : la ligne verte épaisse, en haut, est la cellule c. Elle traverse la cellule de gauche à droite en ne subissant que deux opérations : un **produit** (porte d'oubli) puis une **addition** (nouvelle information). Tout ce qui est en dessous sert à calculer les portes et le candidat, à partir du bus [h_(t−1), x_t].

9.4 Les quatre calculs d'un pas

- ① **Porte d'oubli f_t** : quelle fraction de l'ancien carnet garde-t-on ? On multiplie c_(t−1) par f_t.
- ② **Candidat c̃_t et porte d'entrée i_t** : le candidat (entre −1 et 1, par un tanh) est la nouvelle information proposée ; la porte d'entrée i_t décide **quelle fraction** on en écrit. On obtient i_t × c̃_t.
- **Mise à jour du carnet** : on additionne les deux. **c_t = f_t × c_(t−1) + i_t × c̃_t**
- ③ **Porte de sortie o_t** : que montre-t-on de ce carnet ? **h_t = o_t × tanh(c_t)**

> **Exemple chiffré** (une seule valeur, pour voir la mécanique) : $c_{t-1} = 2{,}0$; $f_t = 0{,}9$; $i_t = 0{,}3$; $\tilde{c}_t = 0{,}8$; $o_t = 0{,}7$.
>
> $c_t = 0{,}9 \times 2{,}0 + 0{,}3 \times 0{,}8 = 1{,}8 + 0{,}24 =$ **2,04** : on a gardé 90 % de l'ancien carnet et ajouté un peu de nouveau.
>
> $h_t = 0{,}7 \times \tanh(2{,}04) = 0{,}7 \times 0{,}967 \approx$ **0,677**.

9.5 Sur une phrase

Reprenons « Les chats que mon voisin a adoptés la semaine dernière **dorment** » : il faut garder l'information « sujet pluriel » pendant 9 mots. Les valeurs des portes ci-dessous sont **illustratives** ; dans un vrai modèle, elles sont apprises.

Mot lu	Ce que fait le réseau	Portes
« chats »	il écrit « sujet pluriel » dans le carnet	i_t élevée
« que mon voisin a adoptés la semaine dernière »	il ne touche pas à cette note et n'y ajoute rien	f_t proche de 1, i_t proche de 0
« dorment »	il montre la note pour choisir la bonne forme du verbe	o_t élevée

Un RNN simple n'a pas ce choix : il doit tout réécrire à chaque mot.

9.6 La différence sur la mémoire

On présente une information au pas 1, puis plus rien. Pour le LSTM, on suppose que le réseau a appris $f = 0{,}95$ et $i = 0$ après le pas 1 :

Pas	RNN simple (état h_t)	LSTM (cellule c_t, $f = 0{,}95$)
1	0,762	1,000
2	0,363	0,950
3	0,180	0,903
4	0,090	0,857
5	0,045	0,815
Conservé	6 %	81 %

Le LSTM n'a besoin que de mettre f proche de 1 pour conserver l'information. Le RNN simple n'a pas de réglage équivalent.

9.7 Pourquoi le gradient ne s'évanouit pas

RNN simple. Pour remonter d'un pas, le gradient est multiplié par la matrice $\mathrm{diag}(1 - h_t^2) \cdot W_{hh}$. La dérivée du tanh est inférieure ou égale à 1 et W_{hh} est souvent de petite norme : après k pas, le produit tend vers 0.

LSTM. Regardons l'équation de la cellule, $c_t = f_t \times c_{t-1} + i_t \times \tilde{c}_t$:

- sur le chemin de la cellule, la dérivée de c_t par rapport à c_{t-1} vaut **f_t**, un simple nombre entre 0 et 1, sans matrice ni dérivée de tanh ;

- le gradient qui remonte la cellule n'est donc multiplié que par les **portes d'oubli** f_t × f_(t−1) × … Ces portes sont **apprises** : le réseau peut les mettre proches de 1 là où il faut se souvenir ;
- la mise à jour est une **addition**, qui transmet le gradient sans l'écraser.

Cette « autoroute » de la cellule s'appelle aussi le « constant error carousel » (carrousel d'erreur constante). Voici le facteur appliqué au gradient sur le chemin de la cellule après k pas :

Pas remontés k	RNN (W = 0,5)	LSTM, f = 0,9	LSTM, f = 0,95	LSTM, f = 0,99
5	0,031	0,59	0,77	0,95
10	0,00098	0,35	0,60	0,90
20	0,00000095	0,12	0,36	0,82
50	≈ 10^{-15}	0,005	0,077	0,61

Avec f = 0,99, le gradient vaut encore 82 % après 20 pas, alors qu'il ne reste presque rien dans le RNN simple.

9.8 Les limites

- **Le LSTM ne supprime pas totalement le problème** : si le réseau apprend des portes d'oubli faibles, le gradient diminue quand même. Il existe aussi d'autres chemins, par h_t, où il peut s'évanouir.
- **L'explosion reste possible** : l'écrêtage des gradients reste utile.
- **Astuce courante** : initialiser le biais de la porte d'oubli à une valeur positive (par exemple 1). Le réseau démarre alors avec f proche de 0,73 : plutôt « garder » qu'« oublier ».

Étape 10 – GRU : la version simplifiée

10.1 L'idée : fusionner les deux mémoires

Le GRU (Gated Recurrent Unit) reprend l'idée du LSTM avec moins de pièces. Il supprime la cellule séparée : **un seul état h_t** sert à la fois de mémoire et de sortie. Il n'a que **deux portes** au lieu de trois.

10.2 Le schéma de la cellule

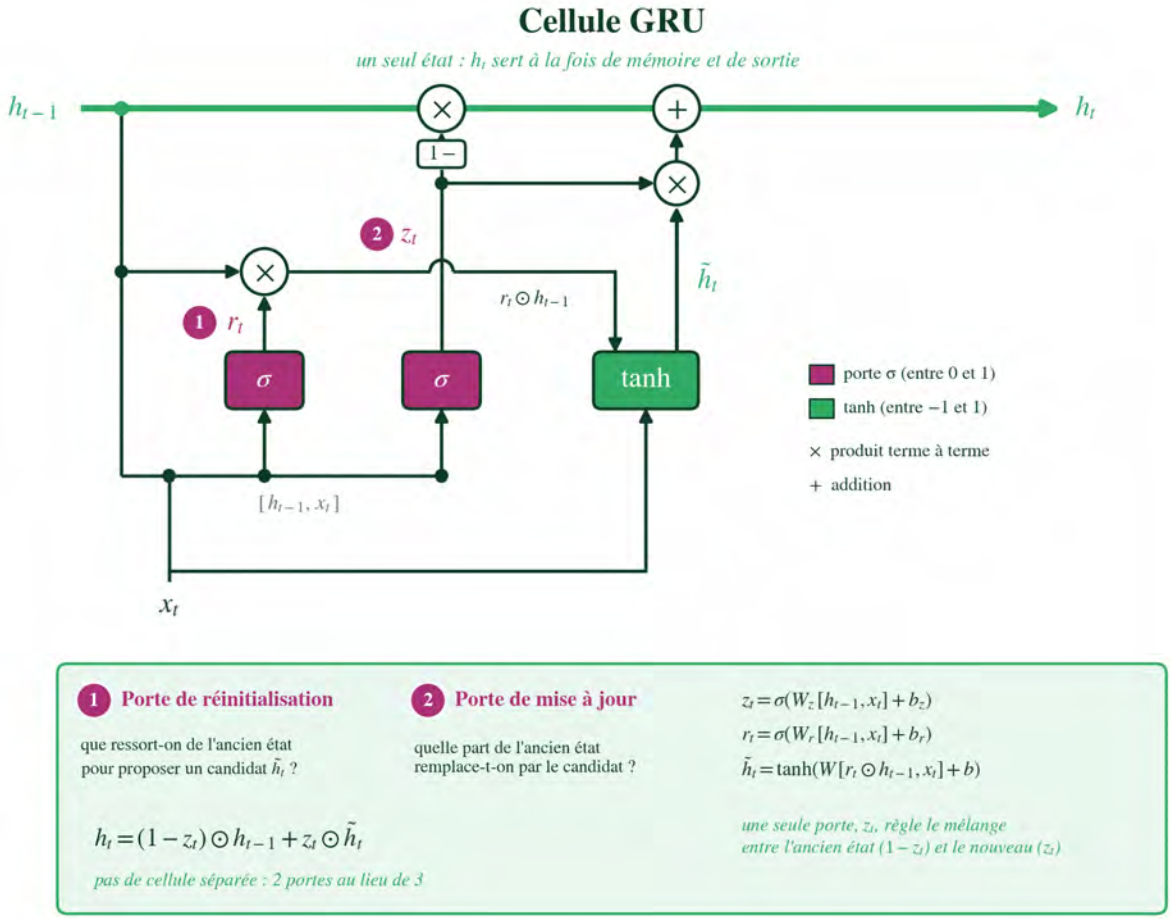

Figure 6 – Une cellule GRU : un seul état, deux portes (réinitialisation et mise à jour)

Comment le lire : la ligne verte du haut est l'état h. Elle subit un produit par (1 − z_t) puis une addition avec le candidat pondéré par z_t. La porte de réinitialisation r_t agit en amont : elle règle ce que l'on reprend de l'ancien état pour **calculer le candidat**.

10.3 Les calculs d'un pas

- ① **Porte de réinitialisation r_t** : dans quelle mesure l'ancien état entre-t-il dans le calcul du candidat ? **h̃_t = tanh(W [r_t ⊙ h_(t−1) , x_t] + b)**. Si r_t est proche de 0, le candidat ignore le passé (utile au début d'une nouvelle phrase).
- ② **Porte de mise à jour z_t** : quelle part de l'ancien état remplace-t-on par le candidat ?
- **Mélange** : **h_t = (1 − z_t) ⊙ h_(t−1) + z_t ⊙ h̃_t**. Une seule porte règle le dosage entre l'ancien et le nouveau.

> **Exemple chiffré** (une valeur) : h_(t−1) = 0,8 ; z_t = 0,2 ; h̃_t = 0,6.
>
> h_t = (1 − 0,2) × 0,8 + 0,2 × 0,6 = 0,64 + 0,12 = **0,76** : on garde 80 % de l'ancien état et on en remplace 20 % par le candidat.

10.4 Le lien avec le LSTM

Idée	LSTM	GRU
Garder de l'ancienne mémoire	porte d'oubli f_t	1 − z_t
Écrire de la nouvelle information	porte d'entrée i_t	z_t
Nombre de portes	3 (oubli, entrée, sortie)	2 (réinitialisation, mise à jour)
Mémoires	cellule c_t et état h_t	un seul état h_t
Sortie contrôlée par une porte	oui (o_t)	non : h_t est exposé tel quel

Le GRU **couple** l'oubli et l'écriture : ce que l'on garde de l'ancien (1 − z_t) et ce que l'on écrit (z_t) sont liés, au lieu d'être réglés séparément.

10.5 Pourquoi le gradient est préservé aussi

Comme pour le LSTM, la mise à jour est une **addition** : $h_t = (1 - z_t) \odot h_{(t-1)} + z_t \odot \tilde{h}_t$. Sur ce chemin, la dérivée de h_t par rapport à h_(t−1) contient le terme **(1 − z_t)**, un simple nombre entre 0 et 1, sans matrice ni dérivée de tanh. Si le réseau apprend z_t proche de 0 là où il faut se souvenir, l'état est quasiment recopié d'un pas à l'autre.

Avec z = 0,05, on a (1 − z) = 0,95 : exactement le cas f = 0,95 du LSTM. L'information du pas 1 est conservée à 81 % après 5 pas, et le gradient vaut 0,36 après 20 pas (contre 10^{-6} pour le RNN simple du tableau de l'étape 9).

10.6 Attention aux conventions

- **Le rôle de z est inversé selon les sources.** Le schéma et les équations de ce document suivent l'article d'origine : z_t pondère le **candidat**. **PyTorch et Keras** écrivent $h_t = z_t \odot h_{(t-1)} + (1 - z_t) \odot \tilde{h}_t$: z_t pondère alors l'**ancien état**. Les deux écritures sont équivalentes en échangeant z_t et 1 − z_t. Je l'ai vérifié en fixant les paramètres d'une couche : avec h_(t−1) = 0,8, une porte de 0,2 et un candidat de 0,6, PyTorch et Keras renvoient 0,64, et non 0,76.
- **La porte de réinitialisation** est appliquée à des endroits légèrement différents selon les bibliothèques (avant ou après la transformation linéaire de h_(t−1)).

10.7 LSTM ou GRU ?

Critère	LSTM	GRU
Portes	3	2
Paramètres (entrée de 3 valeurs, état de 4, PyTorch)	144	108 (RNN simple : 36)
Mémoire	carnet séparé, plus de contrôle	plus simple, un seul état
Vitesse d'entraînement	un peu plus lent	un peu plus rapide
Choix en pratique	souvent essayé en premier sur les longues séquences	bon choix quand les données sont limitées

Aucun des deux n'est meilleur en général : sur la plupart des tâches, leurs performances sont proches, et on compare les deux sur le jeu de validation. Paramètres comptés avec PyTorch (deux biais par porte).

Étape 11 – Pour le voir par soi-même

Ce code (PyTorch) reproduit exactement l'exemple de l'étape 5, avec les mêmes poids :

```
import torch

mots = ["le", "chat", "dort", "mange"]
W_xh = torch.tensor([[0.5, -0.3, 0.2, 0.1],
                     [0.1,  0.8, -0.4, 0.6]])        # 2 x 4
W_hh = torch.tensor([[0.5, 0.1],
                     [-0.2, 0.4]])                   # 2 x 2
W_hy = torch.tensor([[0.2, -0.1],
                     [0.4,  0.3],
                     [-0.3, 0.5],
                     [0.1,  0.2]])                   # 4 x 2

h = torch.zeros(2)                                   # h_0
for mot in ["le", "chat", "dort"]:
    x = torch.eye(4)[mots.index(mot)]                # vecteur one-hot
    h = torch.tanh(W_xh @ x + W_hh @ h)              # nouvel état caché
    y = torch.softmax(W_hy @ h, dim=0)               # probabilités du mot suivant
    print(f"{mot:6s} h = {h.numpy().round(3)}  y = {y.numpy().round(3)}")
```

Il affiche :

```
le     h = [0.462 0.1  ]  y = [0.252 0.288 0.212 0.248]
chat   h = [-0.059  0.634]  y = [0.2   0.255 0.301 0.243]
dort   h = [ 0.23  -0.134]  y = [0.266 0.264 0.219 0.25 ]
```

Avec les couches toutes faites, la même idée s'écrit en quelques lignes : `nn.RNN(3, 4, batch_first=True)` en PyTorch, `keras.layers.SimpleRNN(4, return_sequences=True)` en Keras. Les données ont alors la forme **(lot, longueur de séquence, caractéristiques par pas)**, et l'état caché a ici 4 valeurs. `nn.LSTM` et `nn.GRU` s'utilisent de la même façon.

Pour compter les paramètres du RNN simple, du GRU et du LSTM (entrée de 3 valeurs, état de 4) :

```
import torch.nn as nn
for nom, couche in [("RNN", nn.RNN(3, 4)), ("GRU", nn.GRU(3, 4)), ("LSTM", nn.LSTM(3,
4))]:
    print(nom, sum(p.numel() for p in couche.parameters()))
# RNN 36, GRU 108, LSTM 144
```

À retenir

1. Les séquences posent trois difficultés : **longueur variable, ordre, dépendances à distance**.

2. Un RNN lit la séquence élément par élément et met à jour un **état caché** h_t : sa mémoire.

3. h_t = tanh(W_xh x_t + W_hh h_(t−1) + b_h) : le nouvel état combine le mot courant et le passé.

4. Les **mêmes poids** sont réutilisés à chaque pas : le nombre de paramètres ne dépend pas de la longueur.

5. On entraîne par **rétropropagation dans le temps** : le réseau est déplié, les gradients s'additionnent.

6. Les gradients **s'évanouissent ou explosent** sur les longues séquences : le **LSTM** et le **GRU** ajoutent des portes et une mise à jour **additive**, qui laissent circuler l'information et le gradient.

7. Le LSTM tient un **carnet** c_t qu'il édite (oubli, entrée, sortie) ; le GRU fusionne les deux mémoires avec **deux portes**.