

Mémoire de fin d'études
Pour l'obtention du diplôme d'Ingénieur d'État en Informatique
Option : Systèmes Informatiques

Création d'un corpus de l'aphasie de Broca et
développement d'un système Speech-to-speech de
réhabilitation de la parole

Réalisé par :
BELGOUMRI Mohammed
Djameleddine
im_belgoumri@esi.dz

Encadré par :
Pr. SMAILI Kamel
smaili@loria.fr
Dr. LANGLOIS David
david.langlois@loria.fr
Dr. ZAKARIA Chahnez
c_zakaria@esi.dz

Table des matières

Page de garde	i
Table des matières	i
Table des figures	ii
Sigles et abréviations	iii
1 Apprentissage séquence-à-séquence	1
1.1 Énoncé du problème	1
1.2 Perceptrons multicouches	2
1.2.1 Généralités	2
1.2.2 Application à la modélisation de séquence	3
1.2.3 Avantages et inconvénients	4
1.3 Architecture encodeur-décodeur	4
1.4 Réseaux de neurones récurrents	5
1.4.1 Réseaux de neurones récurrents simples	6
1.4.2 Portes, gated recurrent unit et long short-term memory	7
1.5 Réseau de neurones à convolutions	10
Bibliographie	12

Table des figures

1.1	Architecture sous-jacente d'un MLP de profondeur 4.	2
1.2	Architecture encodeur-décodeur	5
1.3	RNN v.s FFN	5
1.4	Dépliage temporel d'un RNN sur une entrée de longueur 4. . . .	6
1.5	Dépliage temporel d'un encodeur-décodeur récurrent.	7
1.6	Forme générale d'un RNN à portes.	8
1.7	Architecture interne d'un GRU (CHUNG et al., 2014, Fig. 1b) . . .	8
1.8	Architecture interne d'un LSTM (CHUNG et al., 2014, Fig. 1a) . . .	9

Sigles et abréviations

BPTT	rétro-propagation dans le temps
CNN	réseau de neurones à convolutions
FFN	réseau de neurones feed-forward
GRU	gated recurrent unit
LSTM	long short-term memory
ML	apprentissage automatique
MLP	perceptron multicouches
MT	traduction automatique
NLP	traitement automatique du langage
RNN	réseau de neurones récurrent
S2S	séquence-à-séquence

Chapitre 1

Apprentissage séquence-à-séquence

Les modèles séquence-à-séquence (S2S) sont une famille d’algorithmes d’apprentissage automatique (ML, de l’anglais : machine learning) dont l’entrée et la sortie sont des séquences (MARTINS, 2018). Plusieurs tâches d’ML, notamment en traitement automatique du langage (NLP, de l’anglais : natural language processing), peuvent être formulées comme tâches d’apprentissage S2S. Parmi ces tâches, nous citons : la création de chatbots, la réponse aux questions, la traduction automatique (MT) et la reconnaissance automatique de la parole (FATHI, 2021).

Dans ce chapitre, nous commençons par formuler le problème de modélisation de séquences. En suite, nous présentons les architectures neuronales les plus utilisées pour cette tâche. En fin, nous terminons avec une étude comparative de celles-ci.

1.1 Énoncé du problème

Formellement, le problème de modélisation S2S est celui de calculer une fonction partielle $f : X^* \rightarrow Y^*$, où :

- X est un ensemble dit d’entrées.
- Y est un ensemble dit de sorties.
- Pour un ensemble A , $A^* = \bigcup_{n \in \mathbb{N}} A^n$ est l’ensemble de suites de longueur finie d’éléments de A .

f prend donc un $x = (x_1, x_2, \dots, x_n) \in X^n$ et renvoie un $y = (y_1, y_2, \dots, y_m) \in Y^m$. Dans le cas général, $n \neq m$ et aucune hypothèse d’alignement n’est supposée. Il est souvent de prendre $X = \mathbb{R}^{d_e}$ et $Y = \mathbb{R}^{d_s}$ avec $d_e, d_s \in \mathbb{N}$. Dans ce cas, $x \in \mathbb{R}^{d_e \times n}$ et $y \in \mathbb{R}^{d_s \times m}$. Les indices peuvent représenter une succession temporelle ou un ordre plus abstrait comme celui des mots dans une phrase (MARTINS, 2018).

La majorité des outils mathématiques historiquement utilisées pour ce problème viennent de la théorie du traitement de signal numérique. Cependant, l’approche actuellement dominante et celle qui a fait preuve de plus de succès, est de les combiner avec les réseaux de neurones.

1.2 Perceptrons multicouches

Les réseaux de neurones profonds sont parmi les modèles les plus expressifs en ML. Leur succès pratique est incomparable aux modèles qui les ont précédés, que se soit en termes de qualité des résultats ou de variétés de domaines d’application (SEBASTIAN & MIRJALILI, 2017). De plus, grâce aux théorèmes dits d’approximation universelle, ce succès empirique est formellement assuré (CALIN, 2020).

1.2.1 Généralités

Dans cette section, nous introduisons les perceptrons multicouches (MLP, de l’anglais : multi-layer perceptron), l’architecture neuronale la plus simple et la plus utilisée. Il s’agit d’une simple composition de couches affines avec des activations non affines (voire Définition 1).

Définition 1 (MLP (MUKHERJEE, 2021)).

Soient $k, w_0, w_1, \dots, w_{k+1} \in \mathbb{N}$, un réseau de neurones feed-forward de profondeur $k + 1$, à w_0 entrées et w_{k+1} sorties, est défini par une fonction :

$$\begin{cases} \mathbb{R}^{w_0} \rightarrow \mathbb{R}^{w_{k+1}} \\ x \mapsto \varphi_{k+1} \circ A_{k+1} \circ \varphi_k \circ A_k \circ \dots \circ \varphi_1 \circ A_1(x) \end{cases} \quad (1.1)$$

Où les A_i sont des fonctions affines $\mathbb{R}^{w_{i-1}} \rightarrow \mathbb{R}^{w_i}$ et les φ_i sont des fonctions quelconques, typiquement non affines $\mathbb{R}^{w_i} \rightarrow \mathbb{R}^{w_i}$, dites *d’activations*. La fonction $\varphi_i \circ A_i$ est appelée la $i^{\text{ème}}$ couche du réseau.

Un tel réseau de neurones est souvent représenté par un graphe orienté acyclique $(k + 1)$ -partie appelé son “architecture sous-jacente” (KEARNS & VAZIRANI, 1994). La Figure 1.1 illustre l’architecture d’un MLP de profondeur 4 avec $(w_0, w_1, w_2, w_3, w_4) = (4, 5, 7, 5, 4)$.

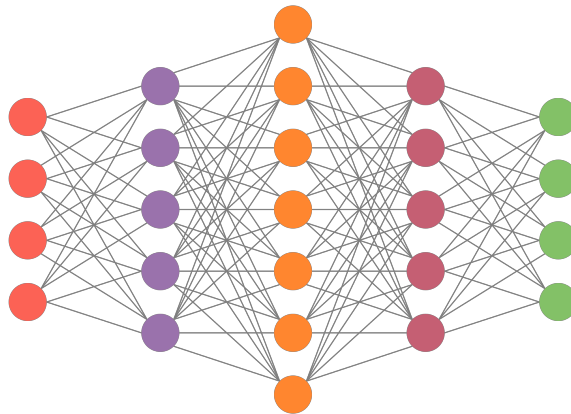


FIGURE 1.1 – Architecture sous-jacente d’un MLP de profondeur 4.

Deux MLP peuvent avoir la même architecture sous-jacente, en effet, cette dernière ne dépend que des dimensions de leurs couches respectives. De ce fait, une méthode de trouver pour une architecture et une fonction cible données le meilleur MLP est nécessaire.

Pour ce faire, nous exploitons le fait que les A_i soient des applications affines sur des espaces de dimensions finies. Nous pouvons donc les écrire comme combinaisons de produits matriciels et de translations. Le problème se réduit donc à régler¹ les paramètres des matrices en question. Cela nécessite une façon de quantifier la qualité d'approximation d'une fonction f par une autre \hat{f} . L'analyse fonctionnelle nous en donne plusieurs, les équations (1.2) et (1.3) sont deux exemples récurrents de fonctions dites *de perte*.

$$L_1(f, \hat{f}) = \|f - \hat{f}\|_1 = \int |f - \hat{f}| \quad (1.2)$$

$$L_2(f, \hat{f}) = \|f - \hat{f}\|_2^2 = \int |f - \hat{f}|^2 \quad (1.3)$$

Ayant fixé une fonction de perte L , l'entraînement revient à un problème d'optimisation comme représenté par l'équation 1.4.

$$f^* = \underset{\hat{f}}{\operatorname{argmin}} L(f, \hat{f}) \quad (1.4)$$

Dans le cas particulier où L est différentiable, l'algorithme du gradient peut être utilisé pour trouver un minimum local. Les gradients sont calculés en utilisant une méthode de dérivation automatique comme la rétro-propagation.

1.2.2 Application à la modélisation de séquence

Les réseaux de neurones opèrent sur des vecteurs. À fin de les utiliser dans le contexte de la modélisation S2S, il faut donc utiliser une représentation vectorielle des entrées. Une telle représentation s'appelle un *plongement* (embedding en anglais). Le plongement peut-être appris ou prédéfini.

Dans le cas des MLP, la séquence d'entrée est d'abord décomposée en sous-séquences. En suite, les plongements de ces sous-séquences sont traités un par un par le réseau de neurones, ce qui produit une séquence de vecteurs en sortie. (Voire l'Algorithme 1.1).

ALGORITHME 1.1 : Passe d'un MLP

```

Input : input_sequence // Séquence d'entrée
1 begin
2    $y \leftarrow \emptyset$ 
3   foreach  $x \in \text{sub\_sequences}(\text{input\_sequence})$  do
4      $\tilde{x} \leftarrow \text{embedding}(x)$ 
5      $y \leftarrow y \cup \text{FFN}(\tilde{x})$ 
6   end
7 end
Output :  $y$  // Séquence de sortie

```

1. En ML, le terme "entraîner" est plutôt utilisé.

1.2.3 Avantages et inconvénients

Les MLP présentent deux avantages par rapport aux architectures discutées dans le reste de ce chapitre. Le premier est leur simplicité. Elle les rend plus simples à comprendre et à implémenter. Le deuxième est le fait qu'ils traitent indépendamment les sous-séquences. Cela rend très facile la tâche de les paralléliser, et par conséquent, il accélère considérablement leur entraînement.

Cependant, ce dernier point pose un grand problème. Comme ils traitent indépendamment les blocs de la séquence, les MLP ne peuvent pas modéliser les dépendances inter-bloc. Par conséquent, leur performance sur les séquences composées de plusieurs blocs est très médiocre. La solution de ce problème est d'augmenter la taille du bloc (est donc aussi la dimension d'entrée). Or, cela suppose un alignement par blocs entre les deux séquences. Une hypothèse invalide selon la Section 1.1.

1.3 Architecture encodeur-décodeur

Les lacunes des MLP sont en large partie due au traitement séparé des parties des séquences. Dans la production de l'élément (ou bloc) courant de la sortie, un MLP se base uniquement sur l'élément (ou bloc) correspondant de l'entrée. L'équation 1.5 l'illustre pour une entrée $x = (x_1, x_2, \dots, x_n)$ et une sortie $y = (y_1, y_2, \dots, y_m)$.

$$y_j = f(x_i, x_{i+1}, \dots, x_{i+\ell}) \quad 1 \leq j \leq m \quad (1.5)$$

En plus de l'hypothèse implicite de l'existence d'une telle correspondance, cela suppose que les éléments d'une séquence sont complètement indépendants l'un de l'autre. Cette dernière hypothèse n'est presque jamais vérifiée.

Une façon naturelle de combler ces lacunes est d'abandonner le traitement par bloc de l'entrée. Tout élément de la séquence de sortie est considéré comme fonction de la séquence d'entrée toute entière. L'équation 1.6 montre cette approche sur l'exemple précédent.

$$y_j = f(x) = f(x_1, x_2, \dots, x_n) \quad 1 \leq j \leq m \quad (1.6)$$

L'architecture encodeur-décodeur y parvient en combinant deux modules : un encodeur et un décodeur. L'encodeur consomme la suite d'entrée et produit un vecteur de dimension fixe qui la représente.² Le décodeur consomme ce vecteur et produit la sortie (Voi- re Figure 1.2). L'équation 1.7 le montre sur le même exemple.

$$\begin{aligned} c &= \text{encoder}(x) \\ y &= \text{decoder}(c) \end{aligned} \quad (1.7)$$

L'équation 1.7 ne dépend pas du fonctionnement interne de l'encodeur et du décodeur. Les deux modules peuvent avoir deux architectures quelconques, qui peuvent ou non être les mêmes (YANG et al., 2020). Dans le reste de ce chapitre, nous examinons les architectures communes en apprentissage S2S.

2. Ce vecteur est appelé un vecteur de contexte, vecteur de pensée ou encore un encodage.

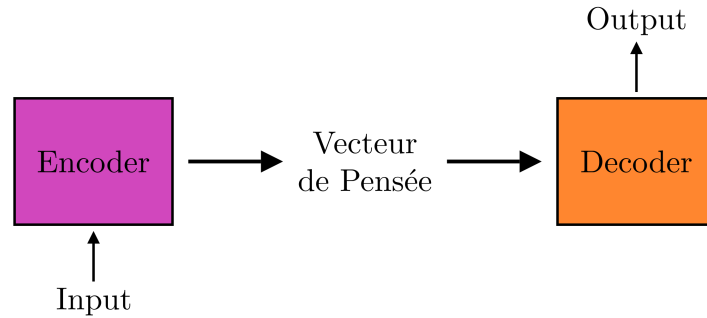


FIGURE 1.2 – Architecture encodeur-décodeur

1.4 Réseaux de neurones récurrents

L'un des principaux défauts que nous avons observés avec les MLP et qui nous ont poussés à introduire l'architecture encodeur-décodeur, est leur incapacité de représenter la dépendance entre les éléments d'une séquence. Une incapacité qui résulte de leur traitement indépendant des éléments.

Les réseaux de neurones récurrents (RNN, de l'anglais : recurrent neural network) tentent à résoudre ce problème en utilisant un état interne persistant. Chaque élément de la séquence modifie cet état lors de son traitement. Cela permet aux premiers éléments d'affecter le traitement des éléments qui les suivent, ainsi permettant à l'information de se propager vers le futur, ce qui donne lieu à une *mémoire*. Pour implémenter ce type de

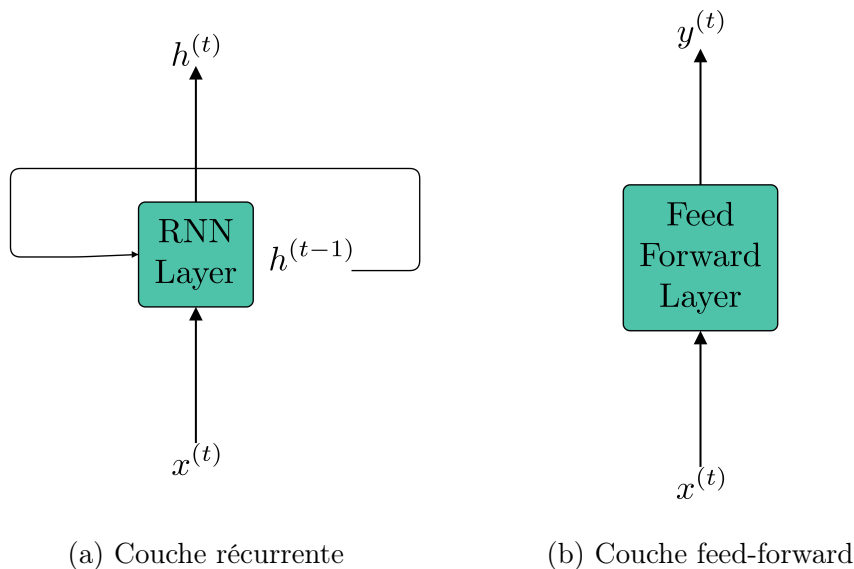


FIGURE 1.3 – RNN v.s FFN

comportement, l'état d'un RNN est décalé d'une unité et réinjecté dans l'entrée (Voir Figure 1.3a). Cela est très différent des MLP qui n'ont pas de boucle de rétroaction, il s'agit de réseaux de neurones feed-forwards (FFN, de l'anglais : feed-forward network longplural) (Voir Figure 1.3b). Par conséquent, ils n'ont pas d'état ni de mémoire (FATHI, 2021).

1.4.1 Réseaux de neurones récurrents simples

L’RNN le plus simple à imaginer se réduit à une combinaison affine de l’état passé et de l’entrée. Il permet de l’information sur le passé de se propager sans contrôle particulier. Mathématiquement, une couche d’un tel RNN prend la forme suivante

$$h^{(t)} = \varphi \left(U h^{(t-1)} + W x^{(t)} + b \right) \quad 1 \leq t \leq n \quad (1.8)$$

où t est le temps³, $x^{(t)}, h^{(t)}$ sont respectivement l’entrée et la sortie à l’instant t , n est la longueur de la séquence et φ est la fonction d’activation (FATHI, 2021). Dans le cas où φ est l’identité, la transformée en z de l’équation 1.8 est donnée par

$$H(z) = z(zI - U)^{-1}(WX(Z) + b) \quad (1.9)$$

il s’agit donc d’un système à réponse impulsionnelle infinie (FATHI, 2021).

Dépliage temporel et encodeur–décodeur récurrent

Le traitement d’une séquence x par un RNN (\mathcal{R}), est équivalent à son traitement par un FFN (\mathcal{F}) dont la profondeur est égale à la longueur de x . \mathcal{F} est donc appelé *dépliage temporel* de \mathcal{R} pour x . La Figure 1.4 montre le dépliage temporel de la Figure 1.3a pour une séquence de longueur 4 (LECUN et al., 2015).

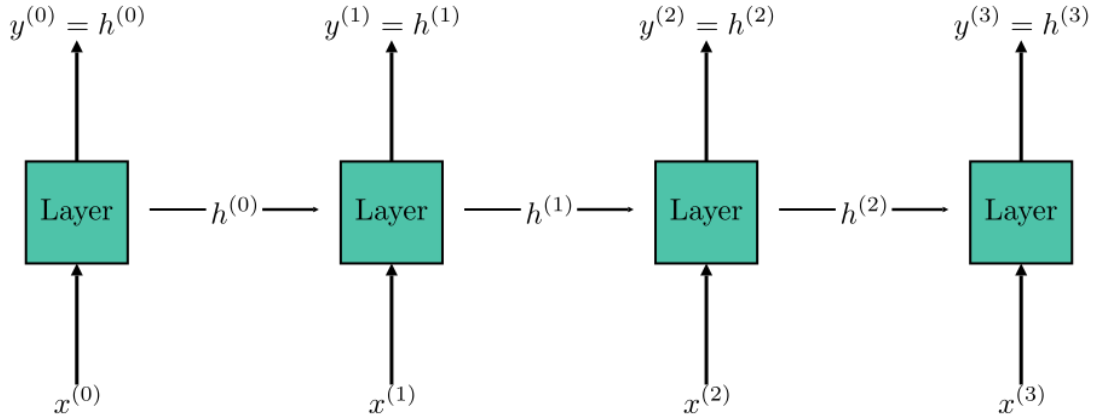


FIGURE 1.4 – Dépliage temporel d’un RNN sur une entrée de longueur 4.

Chaque état caché $h^{(i)}$ contient de l’information sur tous les $x^{(j)}$, $j \leq i$. En particulier, $h^{(n)}$ contient de l’information sur toute la séquence x . Un encodeur récurrent peut donc retourner son dernier état caché comme vecteur d’encodage (YANG et al., 2020).

De sa part, un décodeur récurrent peut conditionner sur l’encodage et passer ses états cachés à une couche supplémentaire qui les interprète comme plongements des éléments de la sortie y (FATHI, 2021). Un tel décodeur n’a pas besoin d’entrée séquentielle (Voire Figure 1.5)

3. Il peut être continu ou discret, réel ou abstrait.

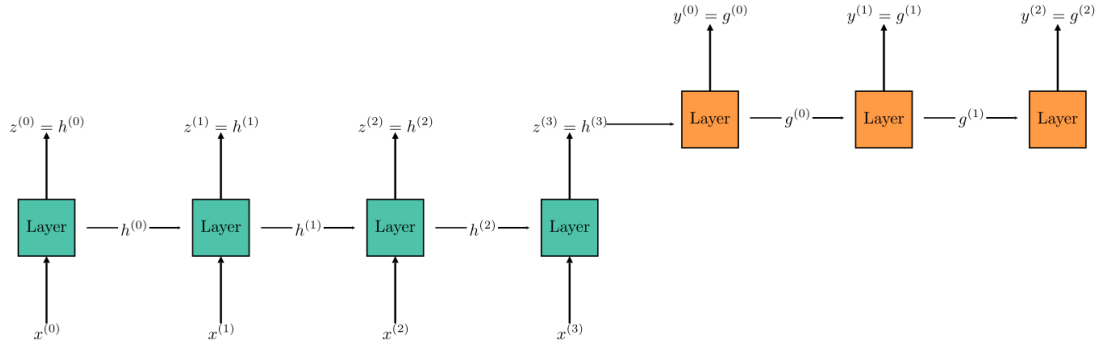


FIGURE 1.5 – Dépliement temporel d’un encodeur-décodeur récurrent sur une entrée de longueur 4 et une sortie de longueur 3.

Rétro-propagation dans le temps et mémoire à court terme

L’entraînement d’un RNN sur un exemple se fait en le dépliant, puis en entraînant le FFN qui résulte par rétro-propagation. L’algorithme résultant s’appelle la rétro-propagation dans le temps (BPTT, de l’anglais : back-propagation through time). Cela est problématique, car la taille de l’entrée n’est théoriquement pas bornée. Par conséquent, la profondeur effective d’un RNN ne l’est pas non plus (FATHI, 2021).

Or, dans l’entraînement d’un réseau de neurones trop profond, les modules des gradients peuvent atteindre des valeurs trop grandes ou trop petites. Il s’agit respectivement des problèmes de “l’explosion du gradient” et de “la disparition du gradient” (BASODI et al., 2020). Une conséquence de ce phénomène est que les RNN simples ont du mal avec les entrées pour lesquelles le dépliement temporel est profond, (i.e les longues entrées). Cette incapacité à modéliser les corrélations à long terme est appelée “mémoire à court terme” (BENGIO et al., 1994).

1.4.2 Portes, gated recurrent unit et long short-term memory

Pour une grande partie des problèmes d’apprentissage S2S, les séquences peuvent être très longues. Le mémoire à court terme constitue donc un véritable obstacle pour l’utilisation des RNN simples en pratique. Une approche pour le contourner qui a eu un énorme succès expérimental, est l’introduction d’un mécanisme de contrôle sur la boucle de rétroaction (Voir Figure 1.6). Ce mécanisme est généralement implémenté avec des *portes*, des unités entraînaables qui peuvent réguler le flux d’information dans la couche récurrente. On parle alors d’*RNN à portes*. Dans cette section, nous explorons les deux variants les plus utilisés d’RNN à portes : le gated recurrent unit (GRU) et le long short-term memory (LSTM).

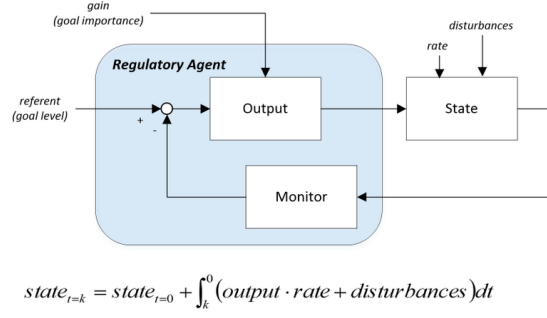


FIGURE 1.6 – Forme générale d'un RNN à portes.

Gated recurrent unit

Le GRU, introduit par (CHO et al., 2014), est une architecture récurrente à portes très simple (Voir Figure 1.7). Elle utilise deux portes. La première est la porte de réinitialisation (r dans la figure 1.7). Elle détermine le point auquel l'information sur le passé peut se propager (quand $r = 0$, pas de propagation et quand $r = 1$, propagation totale). Sa sortie s'appelle *l'état candidat* (\tilde{h} dans la figure). La deuxième est la porte de mise à jour (z dans la figure). Elle détermine les contributions respectives de l'état candidat et l'état passé (si $z = 0$, seul l'état candidat contribue et si $z = 1$, seule l'état courant contribue). Sa sortie est la sortie globale du GRU (CHO et al., 2014).

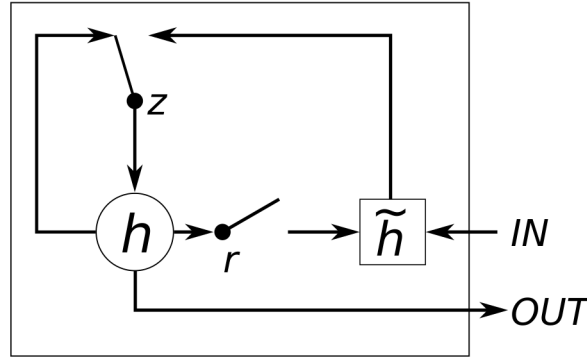


FIGURE 1.7 – Architecture interne d'un GRU (CHUNG et al., 2014, Fig. 1b)

Le fonctionnement des portes d'un GRU est simple. Leurs valeurs sont calculées à partir de l'entrée et de l'état courants par les équations 1.10 et 1.11, où σ est la fonction *sigmoïde*⁴ et \odot et le produit d'Hadamard⁵.

$$z^{(t)} = \sigma(W_z x^{(t)} + U_z h^{(t-1)} + b_z) \quad (1.10)$$

$$r^{(t)} = \sigma(W_r x^{(t)} + U_r h^{(t-1)} + b_r) \quad (1.11)$$

$$\tilde{h}^{(t)} = \varphi(W_h x^{(t)} + U_h (r^{(t)} \odot h^{(t-1)}) + b_h) \quad (1.12)$$

$$h^{(t)} = z^{(t)} \odot h^{(t-1)} + (1 - z^{(t)}) \odot \tilde{h}^{(t)} \quad (1.13)$$

4. $\sigma : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto \frac{1}{1+e^{-x}}$

5. Pour $u, v \in \mathbb{R}^n$, $u \odot v \in \mathbb{R}^n$ et $(u \odot v)_i = u_i v_i$

L'état candidat est calculé à partir de l'entrée et l'état pondéré par la porte de réinitialisation par l'équation 1.12, où φ est la fonction d'activation. Finalement, l'état futur (la sortie) est la moyenne pondérée par z de l'état courant et l'état candidat (CHO et al., 2014). Notons que le GRU devient un RNN simple si les portes de réinitialisation et de mise à jour sont respectivement fixés à 1 et 0 (FATHI, 2021).

Long short-term memory

Il s'agit de l'une des premières architectures récurrentes à portes (CHUNG et al., 2014). Elle a été introduite par (HOCHREITER & SCHMIDHUBER, 1997). Un LSTM implémente trois portes : une porte d'entrées (i), une porte d'oublie (f) et une porte de sortie (o) (Voir Figure 1.8).

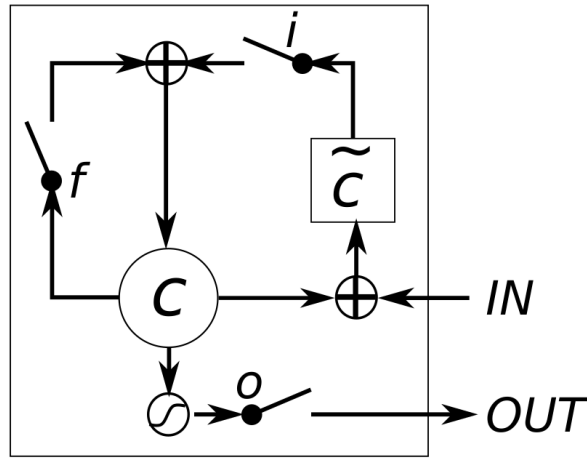


FIGURE 1.8 – Architecture interne d'un LSTM (CHUNG et al., 2014, Fig. 1a)

Le fonctionnement des portes est similaire à celui du GRU. Les équations 1.14–1.19 le montrent en détails⁶ (HOCHREITER & SCHMIDHUBER, 1997).

$$f^{(t)} = \sigma(W_f x^{(t)} + U_f h^{(t-1)} + b_f) \quad (1.14)$$

$$i^{(t)} = \sigma(W_i x^{(t)} + U_i h^{(t-1)} + b_i) \quad (1.15)$$

$$o^{(t)} = \sigma(W_o x^{(t)} + U_o h^{(t-1)} + b_o) \quad (1.16)$$

$$\tilde{c}^{(t)} = \tanh(W_c x^{(t)} + U_c h^{(t-1)} + b_c) \quad (1.17)$$

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)} \quad (1.18)$$

$$h^{(t)} = o^{(t)} \odot \varphi(c^{(t)}) \quad (1.19)$$

6. $\tanh : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Évaluation des réseaux de neurones récurrents

Nous avons établi que les RNN simples souffrent du mémoire à court terme (BENGIO et al., 1994 ; PASCANU et al., s. d.). En utilisant les portes pour contrôler le flux d'information, les GRU et LSTM résolvent le problème au prix d'architectures plus complexes et par conséquent plus lourdes à entraîner. Ils ont des performances similaires est largement meilleures que celle des RNN (CHUNG et al., 2014).

Cependant, toutes les architectures récurrentes présentent un problème fondamental : elles fonctionnent séquentiellement. Bien que cela les rend naturellement mieux adaptées à la modélisation de séquences, il les rend aussi quasi impossibles à paralléliser pour exploiter des architectures parallèles (GPU). Par conséquent, l'entraînement des RNN est extrêmement lent (STAHLBERG, 2020).

1.5 Réseau de neurones à convolutions

Les réseaux de neurones à convolutions (CNN, convolutional neural network) sont une famille d'FFN typiquement utilisés en traitement d'images. Ils ont été introduits par (FUKUSHIMA, 1980) et popularisés par (LECUN et al., 1989 ; LECUN et al., 1998).

Bibliographie

- BASODI, S., JI, C., ZHANG, H., & PAN, Y. (2020). Gradient amplification : An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3(3), 196-207. <https://doi.org/10.26599/BDMA.2020.9020004>
- BENGIO, Y., SIMARD, P., & FRASCONI, P. (1994). Learning long-term dependencies with gradient descent is difficult. 5(2), 157-166. <https://doi.org/10.1109/72.279181>
- CALIN, O. (2020). *Deep Learning Architectures*. Springer. <https://link.springer.com/book/10.1007/978-3-030-36721-3>
- CHO, K., van MERRIENBOER, B., BAHDANAU, D., & BENGIO, Y. (2014). On the Properties of Neural Machine Translation : Encoder-Decoder Approaches [arXiv :1409.1259 [cs, stat]], (arXiv :1409.1259). <http://arxiv.org/abs/1409.1259>
- CHUNG, J., GULCEHRE, C., CHO, K., & BENGIO, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling [arXiv :1412.3555 [cs]], (arXiv :1412.3555). <http://arxiv.org/abs/1412.3555>
- FATHI, S. (2021). *Recurrent Neural Networks*. <https://link.springer.com/book/10.1007/978-3-030-89929-5>
- FUKUSHIMA, K. (1980). Neocognitron : A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4), 193-202. <https://doi.org/10.1007/BF00344251>
- HOCHREITER, S., & SCHMIDHUBER, J. (1997). Long short-term memory. 9(8), 1735-1780.
- KEARNS, M. J., & VAZIRANI, U. (1994). *An Introduction to Computational Learning Theory* [Google-Books-ID : vCA01wY6iywC]. MIT Press.
- LECUN, Y., BOSER, B., DENKER, J. S., HENDERSON, D., HOWARD, R. E., HUBBARD, W., & JACKEL, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4), 541-551. <https://doi.org/10.1162/neco.1989.1.4.541>
- LECUN, Y., BOTTOU, L., BENGIO, Y., & HAFFNER, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324. <https://doi.org/10.1109/5.726791>
- LECUN, Y., BENGIO, Y., & HINTON, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. <https://doi.org/10.1038/nature14539>
- MARTINS, A. (2018). Lecture 9 : Machine Translation and Sequence-to-Sequence Models.
- MUKHERJEE, A. (2021). A Study of the Mathematics of Deep Learning [arXiv :2104.14033 [cs, math, stat]], (arXiv :2104.14033). <http://arxiv.org/abs/2104.14033>
- PASCANU, R., MIKOLOV, T., & BENGIO, Y. (s. d.). On the difficulty of training recurrent neural networks.
- SEBASTIAN, R., & MIRJALILI, V. (2017). *Python Machine Learning : Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow* (2^e éd., T. 1). Packt Publishing.

- STAHLBERG, F. (2020). Neural Machine Translation : A Review. *Journal of Artificial Intelligence Research*, 69, 343-418. <https://doi.org/10.1613/jair.1.12007>
- YANG, S., WANG, Y., & CHU, X. (2020). A survey of deep learning techniques for neural machine translation. *arXiv preprint arXiv :2002.07526*.