République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديـموقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي والبحث العلمي

المدرسة الوطنية للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
École nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

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

2022 / 2023

Table des matières

Table des figures

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	perceptorn 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 learining) 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 $f : X^* \to Y^*$, 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 $x = (x_1, x_2, \cdots, x_n) \in X^n$ et renvoie un $y = (y_1, y_2, \cdots, y_m) \in Y^m$. 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 Perceptorns 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 introduisant les perceptorns multicouches (MLP, de l'anglais : multi-layer perceptorn), 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).

Definition 1 (MLP, Mukherjee, 2021)**.**
Soient $k, w_0, w_1, \cdots, w_{k+1} \in \mathbb{N}$, un réseau de neurones feed-forward de profondeur $k+1$, à w_0 entrées et w_{k+1} sorties, est défini par une fonction :

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

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

À un tel réseau de neurones, on peut associer un graphe orienté acyclique qu'on appelle son "architecture sous-jacente" (Kearns & Vazirani, 1994). La Figure 1.1 illustre l'architecture d'un MLP de profondeur 4 avec $(w_0, w_1, w_2, w_3, w_4) = (4, 5, 7, 5, 4)$.

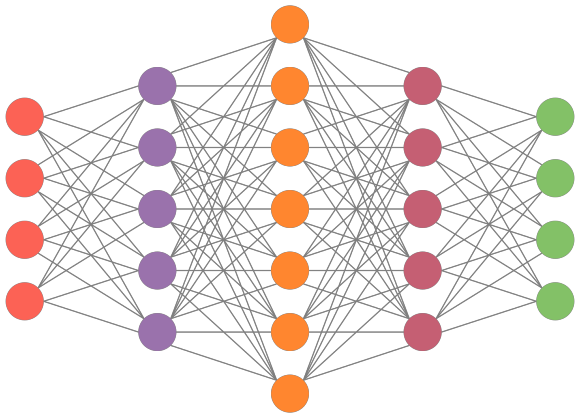


Figure 1.1 – Architecture sous-jacente d'un MLP de profondeur 4.

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 régler [1] 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*.

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

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

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

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

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 apprit 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 le Bloc de code 1.1).

```
def mlp_sequence(input: Sequence):
    """
    @param input: The sequence to be processed
    """
    output = []
    for element in input:
        e = embedding(element)
        output.append(e)
    return output
```

Bloc de code 1.1 – Passe d'un MLP

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 bloques 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 $x = (x_1, x_2, \cdots, x_n)$ et une sortie $y = (y_1, y_2, \cdots, y_m)$.

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

En plus de l'hypothèse implicite de l'existence d'une telle correspondance, cela suppose que les éléments d'une séquence sont complétement 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.

$$y_j = f(x) = f(x_1, x_2, \cdots, x_n) \qquad 1 \leq j \leq m \tag{1.6}$$

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 (Voire Figure 1.2). L'équation 1.7 le montre sur le même exemple.

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

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.

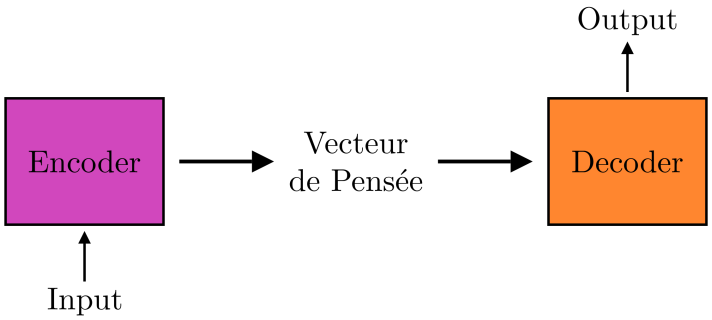
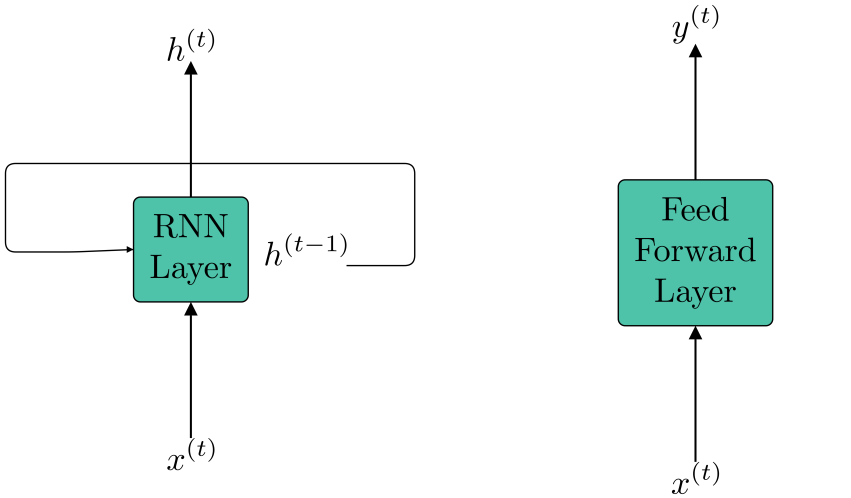


FIGURE 1.2 – Architecture encodeur–décodeur

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 entres 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



(a) Couche récurrente (b) Couche feed-forward

FIGURE 1.3 – RNN v.s FFN

comportement, l'état d'un RNN est décalé d'une unité est réinjecté dans l'entrée (Voire Figure 1.3a). Cela est très différent des MLP qui n'ont pas de boucle de rétroaction, il s'agit de réseau de neurones feed-forwards (FFN, de l'anglais : feed-forward network longplural) (Voire 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

$$h^{(t)} = \varphi\left(Uh^{(t-1)} + Wx^{(t)} + b\right) \qquad 1 \leq t \leq n \tag{1.8}$$

où t est le temps [3], $x^{(t)}, h^{(t)}$ 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

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

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

Dépliement 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 *dépliement temporel* de $\mathcal{R}$ pour x. La Figure 1.4 montre le dépliement temporel de la Figure 1.3a pour une séquence de longueur 4 (LeCun et al., 2015).

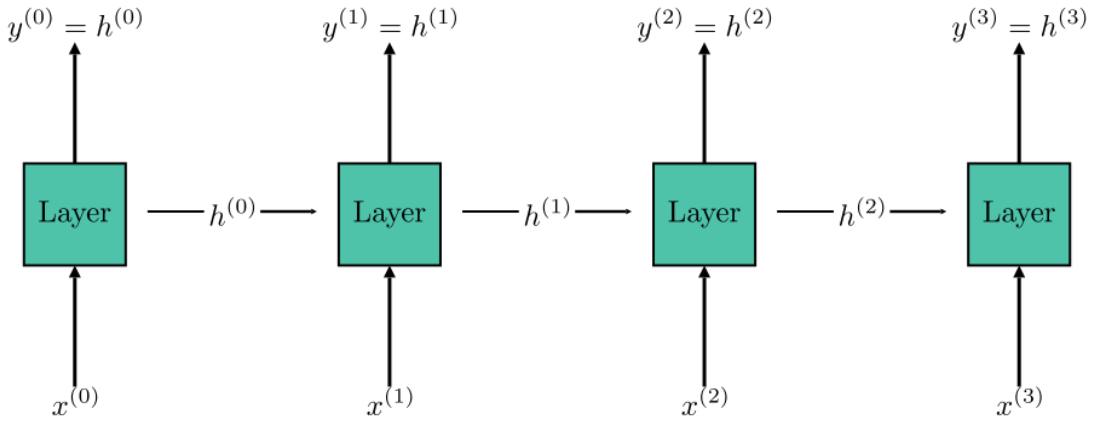


Figure 1.4 – Dépliement temporel d'un RNN sur une entrée de longueur 4.

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.

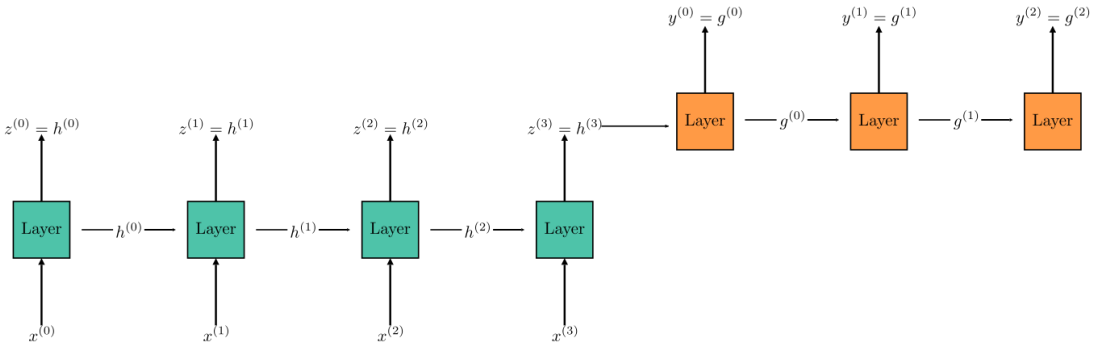


FIGURE 1.5 – Dépliement temporel d'un encodeur–décodeur récurrent sur une entrée de longueur 4 et une sortie de longueur 3.

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'appel la rétro-propagation dans le temps (BPTT, de l'anglais : back-propagation throught time). Cela est problématique, car la taille de l'entrée n'est théoriquement pas bornées. 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 de le contourner qui a eu un énorme succès expérimental, est l'introduction d'un mécanisme de contrôle sur la boucle de rétroaction (Voire Figure 1.6). Ce mécanisme est généralement implémenté avec des *portes*, des unités entraînables 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).

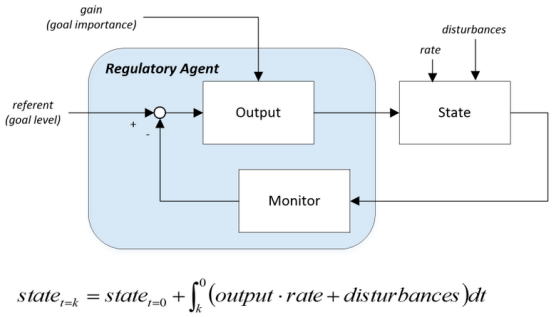


FIGURE 1.6 – Forme générale d'un RNN à portes.

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).

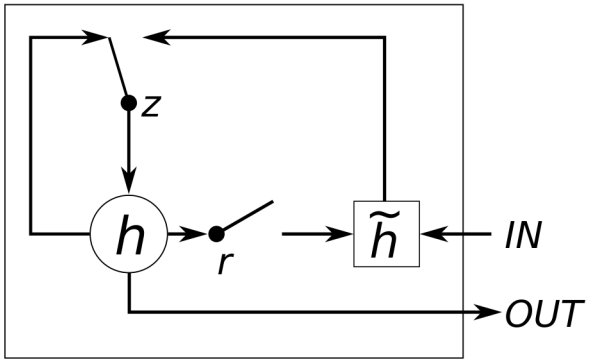


FIGURE 1.7 – Architecture interne d'un GRU (CHUNG et al., 2014, Fig. 1b)

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'Hadamard[5].

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

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

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

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

4. $\sigma : \mathbb{R} \to \mathbb{R}, x \mapsto \frac{1}{1+e^{-x}}$
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 protes (Chung et al., 2014). Elle a été introduite par (Hochreiter & Schmidhuber, 1997). Un LSTM implémente trois portes : une porte d'entrés (i), une porte d'oublie (f) et une porte de sortie (o)(Voir Figure 1.8).

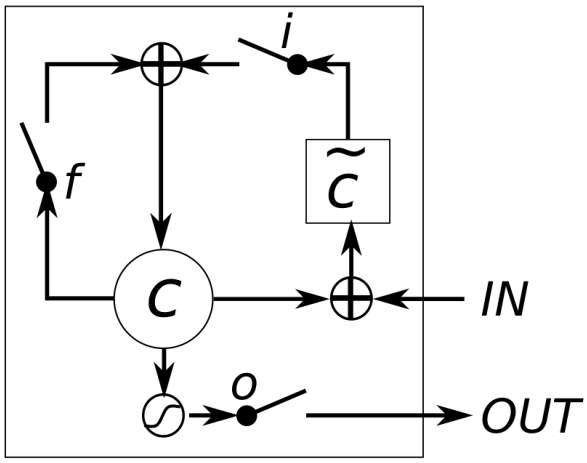


Figure 1.8 – Architecture interne d'un LSTM (Chung et al., 2014, Fig. 1a)

Le fonctionnement des portes est similaire à celui du GRU. Les équations 1.14–1.19 le montrent en détails [6] (Hochreiter & Schmidhuber, 1997).

$$f^{(t)} = \sigma\left(W_f x^{(t)} + U_f h^{(t-1)} + b_f\right) \tag{1.14}$$
$$i^{(t)} = \sigma\left(W_i x^{(t)} + U_i h^{(t-1)} + b_i\right) \tag{1.15}$$
$$o^{(t)} = \sigma\left(W_o x^{(t)} + U_o h^{(t-1)} + b_o\right) \tag{1.16}$$
$$\tilde{c}^{(t)} = \tanh\left(W_c x^{(t)} + U_c h^{(t-1)} + b_c\right) \tag{1.17}$$
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)} \tag{1.18}$$
$$h^{(t)} = o^{(t)} \odot \varphi\left(c^{(t)}\right) \tag{1.19}$$

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

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

En dépit d'être une architecture naturelle pour le traitement des séquences, les RNN sont trop lents pour la majorité des cas d'utilisation pratiques. Cela est dû à leur nature séquentielle qui à son tour, est due à leur utilisation de boucles de rétroaction (Voire Section 1.4). Une architecture sans telles boucles (i.e une architecture d'FFN) est donc préférable.

Les réseaus 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). Les CNN atteignent des performances comparables à celles des RNN sans mémoire explicite. Ils y parviennent en exploitant un outil mathématique appelé produit de *convolution* (Calin, 2020).

Principes mathématiques de fonctionnement

Le produit de convolution de deux signaux f et g est le signal $f * g$ donné par

$$u \mapsto \int_{\mathbb{R}} f(u-t)g(t)\,\mathrm{d}\,t \tag{1.20}$$

il s'agit d'une opération commune en probabilité, analyse fonctionnelle, traitement de signaux et traitement d'images (Barbe & Ledoux, 2012 ; Oppenheim & Schafer, 2013). À partir d'elle, une autre opération appelée *l'inter-corrélation* est définie. L'inter-corrélation de f et g est notée $f \star g$. Elle est définie par

$$u \mapsto \int_{\mathbb{R}} f(t)g(t-u)\,\mathrm{d}\,t = (f * g^{-})(u) \tag{1.21}$$

ou $g^{-} : t \mapsto g(-t)$. Intuitivement, elle mesure la similarité entre les deux signaux en question. Les CNN utilisent l'inter-corrélation à la place des applications linéaires quelconques

d'un MLP. Une couche de convolution unidimensionnelle calcul donc la fonction suivante

$$x \mapsto \varphi\left(b + \sum_{i=1}^{n} x \star w_i\right) \tag{1.22}$$

ou x est l'entrée et les vecteurs w_i sont les paramètres entraînables de la couche, aussi appelés ses *noyaux* ou *masques* (Voire Figure 1.9). La sortie de cette couche est une combinaison de tous les éléments de la séquence d'entrée. En composant suffisamment de telles couches, un CNN apprend une représentation de l'entrée beaucoup plus structurée qu'un MLP. On parle par fois de représentation *hiérarchique*.

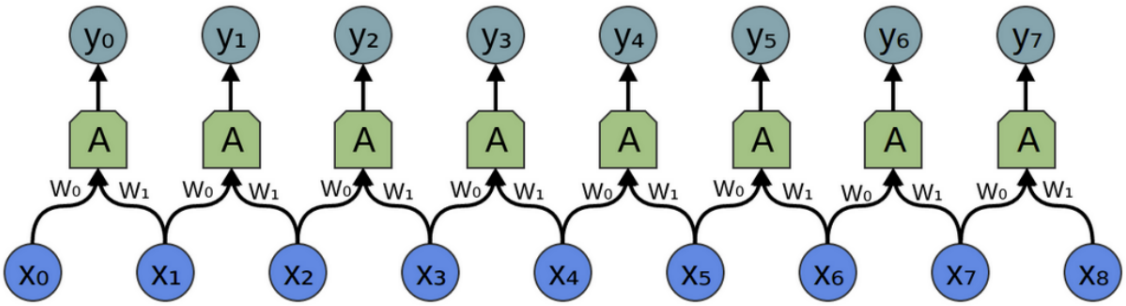


FIGURE 1.9 – Couche convolutive unidimensionnelle.

Application à l'apprentissage S2S

Leur tendance naturelle à produire des représentations synthétiques des entrées, fait des CNN des encodeurs très puissants pour une grande variété de tâches, y compris des tâches de transduction de séquences. Plusieurs travaux ont combiné un encodeur convolutif avec un décodeur récurrent(Voire Figure 1.10, KALCHBRENNER et BLUNSOM, 2013, Fig. 3) (YANG et al., 2020).

Cependant, des architectures totalement convolutives pour l'apprentissage S2S existent également. ByteNet (Voire Figure 1.11 KALCHBRENNER et al., 2017) et ConvS2S (Voire Figure 1.12 KAMEOKA et al., 2020) sont deux exemples de telles architectures.

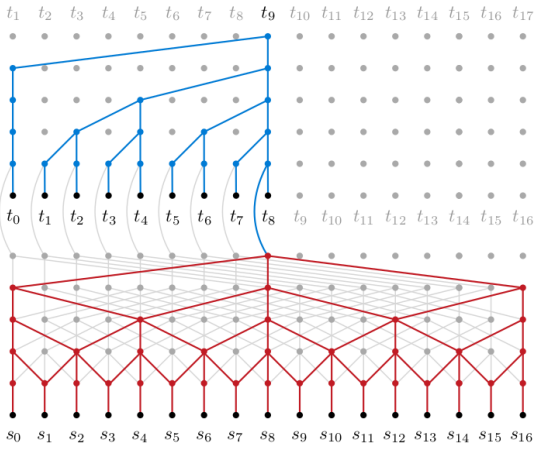


FIGURE 1.11 – Architecture de ByteNet.

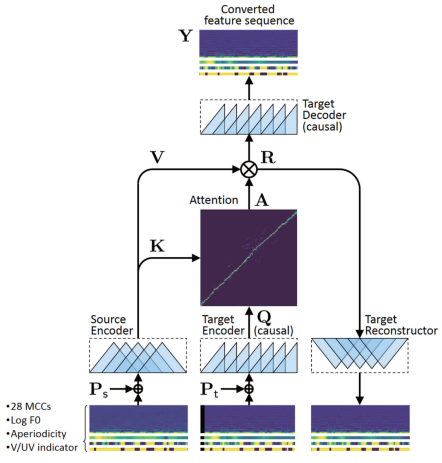


FIGURE 1.12 – Architecture de ConvS2S.

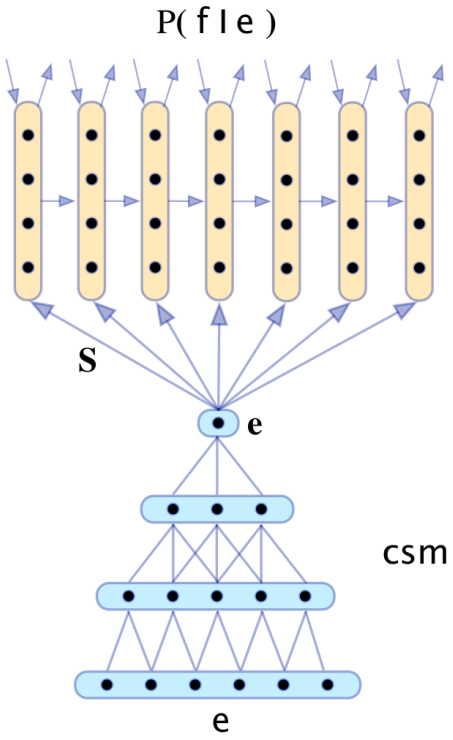


FIGURE 1.10 – Recurrent Continuous Translation Model

Avantages et inconvénients

Le produit de convolution (et par conséquent, inter-corrélation) est une opération très facile à paralléliser. Il est donc possible d'exploiter des architectures matérielles parallèles (GPU) pour accélérer l'entraînement des CNN. À cet effet, les CNN sont beaucoup plus efficaces que les RNN pour les entrées longues. Ils peuvent être employés dans des cas pratiques de tailles raisonnables (LI et al., 2016).

Or, la nature locale du produit de convolution rend difficile aux CNN d'apprendre les corrélations globales[7]. En effet, pour représenter toutes les relations dans une séquence de longueur n, un CNN dont les masques sont de taille $k < n$ doit avoir $\log_k n$ couche (VASWANI et al., 2017). Le traitement de séquences très longues nécessite alors des CNN trop profonds, cequi donne lieu au problèmes de dispartion et d'exploision des gradients.

1.6 Transformeurs

Les architectures discutées dans les sections 1.2 à 1.5 (notamment les encodeurs-décodeurs à base de CNN et RNN) forment la colonne vertébrale des modèles classiques d'apprentissage S2S (YANG et al., 2020). Cependant, leur mise en place est inhibée par des problèmes de performance (Voire sections 1.4 et 1.5).

7. Dépendances entre des couples d'éléments éloignés dans la séquence

Le transformeur (aussi appelé réseau de neurones auto-attentif) est une architecture performante pour le traitement de séquences (Shim & Sung, 2022). Introduite par (Vaswani et al., 2017), elle est basée sur le mécanisme d'attention (Larochelle & Hinton, 2010). Une opération mathématique parallélisable à l'instar du produit de convolution, mais dont la complexité ne dépend pas de la distance dans les séquences.

1.6.1 Architecture générale

Le transformeur a une architecture encodeur–décodeur (Voire Figure 1.13). L'encodeur et le décodeur ont des architectures très similaires, comprenant chacun une couche d'encodage positionnel, une couche d'auto-attention et un MLP. Le décodeur se distingue de l'encodeur en ce qu'il a une couche d'attention croisée entre l'auto-attention et l'MLP. Chacune des couches susmentionnées est suivie d'une couche de normalisation (Ba et al., 2016) qui reçoit également une connexion directe à l'entrée de la couche [8]. L'encodeur et le décodeur peuvent être empilés pour former un transformeur profond.

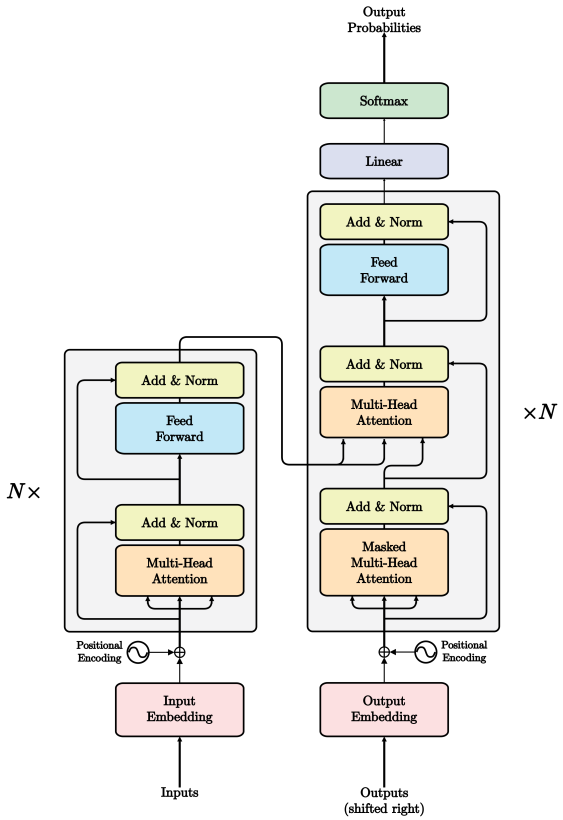


Figure 1.13 – Architecture de transformeur (Vaswani et al., 2017, Fig 1).

Étant donné une séquence d'entrée $x = (x_1, \ldots, x_n)$, l'encodeur calcule un vecteur de pensée z qu'il passe au décodeur. Le décodeur prédit le prochain élément y_i de la séquence de sortie à partir de z et les éléments précédents $(y_1, \ldots, y_{i-1})$. Pour produire y_0, le décodeur commence avec une séquence de sortie vide.

8. On parle de connexion résiduelle ou de connexion de saut (skip connection) (He et al., 2016)

1.6.2 Encodage positionnel

La première couche de l'encodeur aussi bien que du décodeur est une couche d'encodage positionnel. Elle permet de calculer une représentation vectorielle de la séquence en question. Elle est composée d'une couche de plongement et d'une couche d'encodage de la position.

La couche de plongement mappe chaque élément de la séquence à un vecteur de dimension fixe d, ainsi transformant une séquence de longueur n en une matrice de $\mathbb{R}^{n\times d}$. Elle peut avoir une architecture quelconque en fonction de la nature des données. Il peut s'agir d'une couche de plongement lexical dans le cas d'un texte, d'une couche de convolution dans le cas de l'audio ou d'une couche récurrente dans le cas d'une série chronologique.

Le transformeur est un invariant aux permutations [9]. La sortie de la couche de plongement est donc insuffisante pour représenter la séquence. Il lui faut ajouter une information sur l'ordre des éléments. C'est ce que fait la couche d'encodage de la position. Les auteurs (Vaswani et al., 2017) proposent l'encodage suivant pour la position i :

$$\mathrm{PE}_k(i) = \begin{cases} \sin\left(\frac{i}{r^{2k/d}}\right) & \text{si } k \text{ est pair} \\ \cos\left(\frac{i}{r^{2k/d}}\right) & \text{si } k \text{ est impair} \end{cases} \qquad 1 \leq k \leq d \tag{1.23}$$

où d est la dimension de plongement et r est un hyperparamètre fixé à 10^4 par les auteurs. La figure 1.14 montre la matrice d'encodage des 256 premières positions avec $r = 10^4$ et $d = 512$.

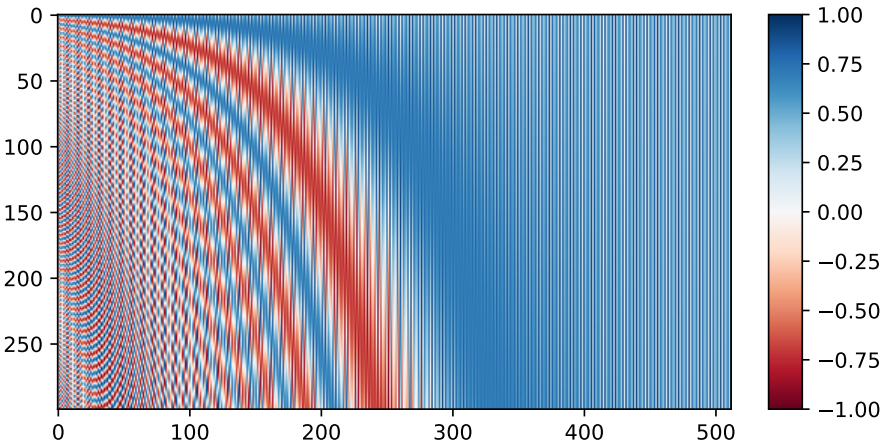


Figure 1.14 – Matrice d'encodage de la position pour $r = 10^4$ et $d = 512$.

La représentation finale de la séquence est la somme de la sortie de la couche de plongement et celle de la couche d'encodage de la position. Une couche de plongement lexical peut être substituée à l'encodage donné par l'équation (1.23) avec des résultats similaires (Vaswani et al., 2017).

9. Pour une séquence $x = (x_1, \ldots, x_n)$ et une permutation $\pi \in S_n$, x et $\left(x_{\pi(i)}\right)_{1\leq i\leq n}$ ont deux encodages identiques.

1.6.3 Couches attention

Le mécanisme d'attention est la partie centrale du transformeur qui réalise sa fonctionnalité. Tous les autres modules sont une forme de prétraitement de son entrée ou de post-traitement de sa sortie. Chaque couche de l'encodeur implémente un module d'attention, tandis que chaque couche du décodeur en implémente deux.

Plusieurs types de mécanismes d'attention ont été proposés dans la littérature, notamment l'attention additive (Bahdanau et al., 2016) et l'attention multiplicative (Luong et al., 2015). Le transformeur utilise une variante de l'attention multiplicative appelée *"scaled dot-product attention"* (Vaswani et al., 2017). Elle opère sur trois matrices Q, K, V [10] de dimensions respectives $d_Q \times l, d_K \times m, d_V \times n$, qui représentent chacune l'encodage d'une séquence [11]. Le résultat est donné par l'équation (1.24).

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V \tag{1.24}$$

Intuitivement, la scaled dot-product attention peut être vue comme une forme de recherche floue dans une base de données. Le terme QK^T est un produit scalaire entre les lignes de la requête Q et la clé K. Il représente la similarité entre les deux. La fonction softmax le normalise pour obtenir des poids positifs de somme 1. Enfin, le résultat et une somme des valeurs V pondérées par ces poids. C'est dans ce sens-là qu'il s'agit d'une recherche floue : les valeurs correspondantes aux clés qui répondent le mieux à la requête sont sélectionnées, mais au lieu de renvoyer discrètement la meilleure valeur, toutes les valeurs contribuent dans la mesure de leur pertinence (« CS480/680 Lecture 19 : Attention and Transformer Networks », 2019). La division par $\sqrt{d_K}$ est une forme de régularisation. Elle permet d'éviter que le module de l'entrée du softmax devienne trop grand, ce qui rend son gradient trop petit ainsi ralentissant l'apprentissage [12] (Vaswani et al., 2017).

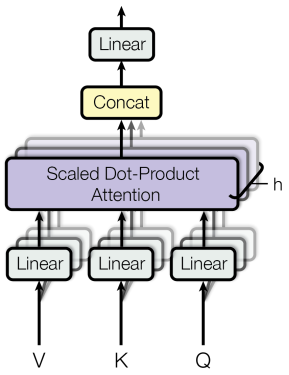


Figure 1.15 – Schéma d'une couche attention multi-tête (Vaswani et al., 2017, Fig 2).

10. De la terminologie anglophone : *"Query"*, *"Key"*, *"Value"*, empruntée aux systèmes de recherche de l'information.

11. d_Q, d_K, d_V sont les dimensions de plongement et l, m, n sont les longueurs des séquences. Notez que l'équation (1.24) impose les contraintes $d_Q = d_K$ et $m = n$, ce qui est vérifié pour les trois modules d'attention du transformeur.

12. Le choix de la valeur de $\sqrt{d_K}$ n'est pas arbitraire. Sous l'hypothèse que les valeurs de Q et K sont centrées et réduites, les valeurs de QK^T sont centrées de variance d_K. La division par $\sqrt{d_K}$ les ramène à une variance de 1 (Vaswani et al., 2017).

Les auteurs de (VASWANI et al., 2017) ont observé une amélioration de la performance en utilisant plusieurs modules (ou têtes) de scaled dot-product attention, projetant les entrées différemment avant de les passer à chaque tête. Les sorties des têtes sont concaténées et passées à une couche linéaire qui produit la sortie finale. Les projections sont réalisées par des couches linéaires entraînables et le nombre h de têtes est un hyperparamètre fixé à 8 par les auteurs(Voire la Figure 1.15).

La couche attention de l'encodeur et la première couche attention du décodeur sont des couches d'*auto-attention* i.e leurs requêtes, clés et valeurs sont les mêmes. L'auto-attention du décodeur se distingue de celle de l'encodeur par la présence du *masque*. En produisant un élément de la séquence de sortie, le décodeur ne doit faire attention qu'aux éléments qui le précèdent. Toutes les autres valeurs doivent avoir un poids nul. Pour ce faire, la matrice M

$$M_{ij} = \begin{cases} 0 & \text{si } j > i \\ -\infty & \text{sinon} \end{cases} = \begin{bmatrix} -\infty & -\infty & \cdots & -\infty \\ 0 & -\infty & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\infty \end{bmatrix} \tag{1.25}$$

est ajoutée à l'entrée de la fonction softmax.

La deuxième couche attention du décodeur est une couche d'*attention croisée*. Les requêtes viennent de la sortie et les clés et valeurs viennent de l'entrée. Cette couche calcul les dépendances entre l'entrée et la sortie, tandis que les deux autres couches calculent les dépendances à l'intérieur de l'entrée et de la sortie. C'est cette couche qui permet au décodeur de conditionner sa sortie sur l'entrée.

Bibliographie

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization [arXiv :1607.06450 [cs, stat]], (arXiv :1607.06450). http://arxiv.org/abs/1607.06450

Bahdanau, D., Cho, K., & Bengio, Y. (2016). Neural Machine Translation by Jointly Learning to Align and Translate [arXiv :1409.0473 [cs, stat]], (arXiv :1409.0473). https://doi.org/10.48550/arXiv.1409.0473

Barbe, P., & Ledoux, M. (2012). *Probabilité (L3M1)*. EDP Sciences.

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

CS480/680 Lecture 19 : Attention and Transformer Networks. (2019). https://www.youtube.com/watch?v=OyFJWRnt_AY

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

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition, 770-778. https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *9*(8), 1735-1780.

Kalchbrenner, N., & Blunsom, P. (2013). Recurrent Continuous Translation Models. *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 1700-1709. https://aclanthology.org/D13-1176

Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A. v. d., Graves, A., & Kavukcuoglu, K. (2017). Neural Machine Translation in Linear Time [arXiv :1610.10099 [cs]], (arXiv :1610.10099). http://arxiv.org/abs/1610.10099

Kameoka, H., Tanaka, K., Kwaśny, D., Kaneko, T., & Nobukatsu, H. (2020). ConvS2S-VC : Fully Convolutional Sequence-to-Sequence Voice Conversion. *IEEE/ACM*

Transactions on Audio, Speech, and Language Processing, *28*, 1849-1863. https://doi.org/10.1109/TASLP.2020.3001456

Kearns, M. J., & Vazirani, U. (1994). *An Introduction to Computational Learning Theory* [Google-Books-ID : vCA01wY6iywC]. MIT Press.

Larochelle, H., & Hinton, G. E. (2010). Learning to combine foveal glimpses with a third-order Boltzmann machine. *Advances in Neural Information Processing Systems*, *23*. https://proceedings.neurips.cc/paper/2010/hash/677e09724f0e2df9b6c000b75b5da10d-Abstract.html

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

Li, X., Zhang, G., Huang, H. H., Wang, Z., & Zheng, W. (2016). Performance Analysis of GPU-Based Convolutional Neural Networks. *2016 45th International Conference on Parallel Processing (ICPP)*, 67-76. https://doi.org/10.1109/ICPP.2016.15

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective Approaches to Attention-based Neural Machine Translation [arXiv :1508.04025 [cs]], (arXiv :1508.04025). http://arxiv.org/abs/1508.04025

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

Oppenheim, A. V., & Schafer, R. W. (2013). *Discrete-time Signal Processing*. Pearson.

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* (2e éd., T. 1). Packt Publishing.

Shim, K., & Sung, W. (2022). A Comparison of Transformer, Convolutional, and Recurrent Neural Networks on Phoneme Recognition [arXiv :2210.00367 [cs, eess]], (arXiv :2210.00367). http://arxiv.org/abs/2210.00367

Stahlberg, F. (2020). Neural Machine Translation : A Review. *Journal of Artificial Intelligence Research*, *69*, 343-418. https://doi.org/10.1613/jair.1.12007

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, & Polosukhin, I. (2017). Attention is All you Need. *Advances in Neural Information Processing Systems*, *30*. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

Yang, S., Wang, Y., & Chu, X. (2020). A survey of deep learning techniques for neural machine translation. *arXiv preprint arXiv :2002.07526*.