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

Extraits de code

Sigles et abréviations

API	interface de programmation d'application
ASR	reconnaissance automatique de la parole
BART	bidirectional auto-regressive transformer
BERT	bidirectional encoder representations from transformers
BLEU	bilingual evaluation understudy
BPE	byte pair encoding
CNN	réseau de neurones à convolutions
DL	apprentissage profond
GPT	generative pre-trained transformer
ML	apprentissage automatique
MLM	modélisation masquée du langage
MLOps	Machine Learning Operations
MT	traduction automatique
NLP	traitement automatique du langage
NMT	traduction automatique neuronale
RNN	réseau de neurones récurrent
S2S	séquence-à-séquence

Chapitre 1

Conception

Dans les chapitres précédents, nous avons effectué une étude bibliographique sur l'aphasie de Broca et les méthodes de traitement automatique du langage (NLP, de l'anglais : natural language processing) qui peuvent être utilisées pour la traiter (particulièrement les modèles séquence-à-séquences (S2S)). Cela nous a permis de développer une idée claire d'un système S2S pour la réhabilitation de la parole aphasique.

Dans ce chapitre, nous allons présenter les détails de la conception de notre système. Nous commençons par décrire la démarche suivie pour le concevoir. Puis, nous présentons l'architecture générale du système, que nous détaillons par la suite.

1.1 Architecture générale de la solution

Pour résoudre le problème de la réhabilitation de la parole aphasique, nous proposons un système dont l'architecture générale est illustrée dans la Figure 1.1. Ce système est composé de deux parties principales : (a) le sous-système d'reconnaissance automatique de la parole (ASR, de l'anglais : automatic speech recognition) qui permet de transcrire la parole aphasique en texte (partie gauche) et (b) le sous-système de traduction automatique neuronale (NMT, de l'anglais : neural machine translation) qui permet de traduire le texte transcrit en parole saine (partie droite).

Pour la partie ASR, nous avons commencé par la collecte de données existantes d'AphasiaBank. Ces données n'étant pas suffisantes, nous avons collecté des données supplémentaires sur internet. Ces dernières ont été transcrites automatiquement à l'aide de Whisper. Les transcriptions automatiques ont été filtrées manuellement pour corriger les erreurs de transcription. Le résultat de cette étape est combiné avec les données d'AphasiaBank pour former un corpus de données de parole aphasique. Ce corpus peut être utilisé pour entraîner un modèle ASR.

Une fois les deux modèles entraînés, ils peuvent être enchainés pour former un système "speech-to-speech" (la phase d'exploitation dans la figure 1.1). Dans la suite de ce chapitre, nous reprenons dans le détail les différentes étapes de cette architecture, que nous avons abordées brièvement dans cette section.

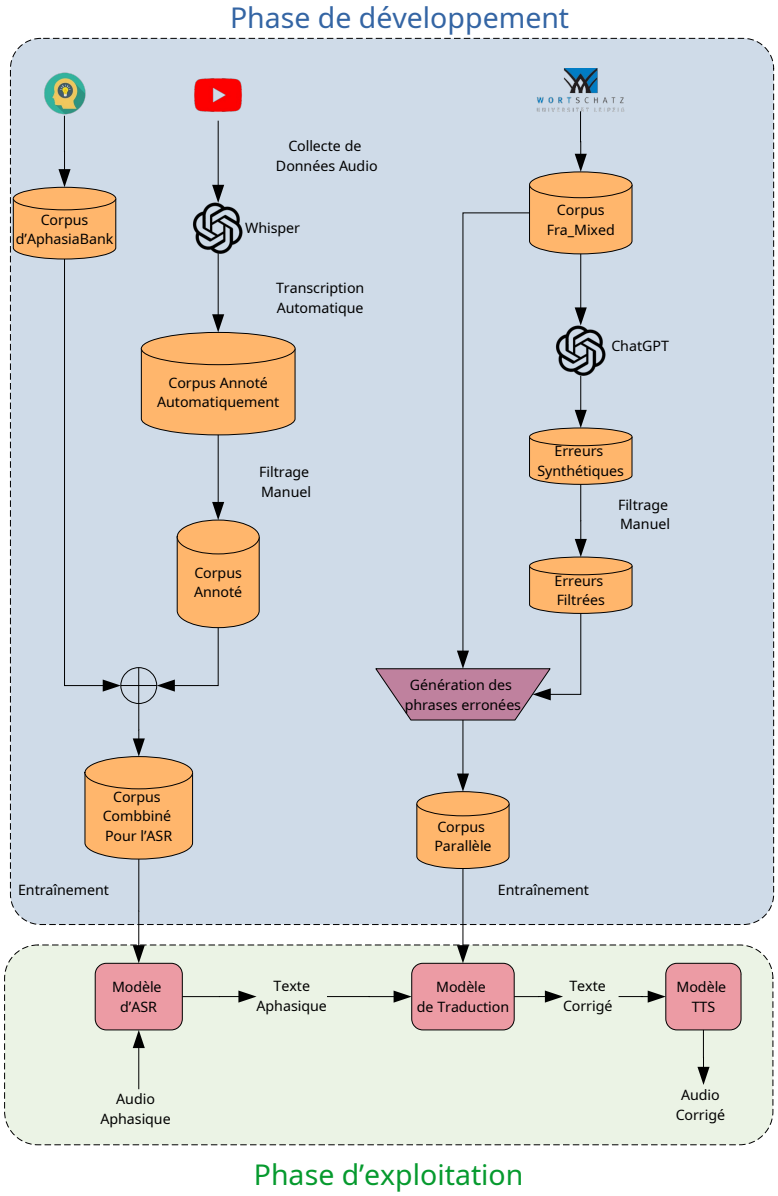


FIGURE 1.1 – Architecture générale de la solution.

1.2 Reconnaissance automatique de la parole

Le modèle d'ASR permet de transformer la parole aphasique en texte dans le but de l'utiliser comme entrée pour le modèle de traduction. Les étapes de sa construction sont présentées dans la Figure 1.2. Dans cette section, nous allons détailler ces étapes. Nous notons que la dernière condition de la Figure 1.1 n'a pas été satisfaite. En effet, nous n'avons pas pu entraîner un modèle ASR à cause du manque de données.

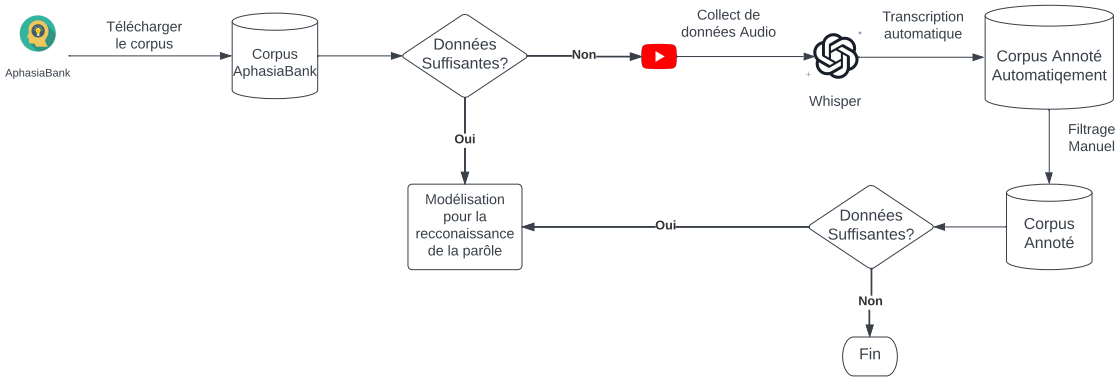


FIGURE 1.2 – Déroulement de la partie ASR.

1.2.1 Préparation des données

La première étape de tout projet de apprentissage automatique (ML, de l'anglais : machine learning) est la préparation des données. Dans ce cas, le jeu de données doit être constitué de couples de morceaux d'audio et de leurs transcriptions textuelles.

Corpus AphasiaBank

Notre choix s'est porté sur le corpus *AphasiaBank* (MACWHINNEY et al., 2011). Il fait partie du projet *TalkBank* (MACWHINNEY, 2007), une collection de bases de données créées pour l'étude du langage. AphasiaBank contient plusieurs vidéos d'entre-teints entre des chercheurs et des personnes souffrant de l'aphasie de Broca. Ses vidéos sont accompagnées par des transcriptions textuelles faites par des experts dans un format particulier. La qualité des transcriptions est donc excellente.

Cependant, le volume de données sur l'aphasie de Broca est très limité. En effet, un seul des 11 exemples disponibles en français est un exemple de l'aphasie de Broca. Il s'agit d'une vidéo de 12 min 03 s, qui contient 3000 mots. Il faut noter que la moitié de ces mots sont prononcés par le chercheur. La durée effective de la parole aphasique est donc de 6 min. Cela est très loin d'être suffisant pour entraîner un modèle profond. Pour ce but, il est nécessaire de collecter des données supplémentaires.

Collecte de données supplémentaires

Les données d'AphasiaBank étant insuffisantes, d'autres sources sont nécessaires. Plusieurs enregistrements de personnes souffrant de l'aphasie de Broca sont disponibles sur internet (YouTube, Vimeo, ...). La qualité de ces enregistrements est très variable et largement inférieure à celle d'AphasiaBank. Cependant, leur ajout au corpus est nécessaire pour augmenter sa taille.

Notre recherche nous a permis d'obtenir 22 enregistrements de personnes souffrant de l'aphasie de Broca d'une durée totale de 48 min 44 s. Des statistiques sur la répartition démographique de ces enregistrements sont présentées dans le tableau 1.1. On y observe

	Nombre	**Durée**
Hommes	7	13 min 45 s
Femmes	31	34 min 2 s
Groupe	2	9 min 57 s

TABLE 1.1 – Répartition des enregistrements collectés par genre.

que les enregistrements sont assez diverses en comparaisons à l'unique enregistrement trouvé sur AphasiaBank.

Transcription et filtrage des données

Les 22 enregistrements collectés sont transcrits à l'aide de Whisper (RADFORD et al., 2022). Cela permet d'avoir des transcriptions textuelles de qualité. Cependant, les transcriptions obtenues contiennent des erreurs (particulièrement pour les prononciations aphasiques).

L'étape suivante est de filtrer à la main les transcriptions obtenues. Cela permet de corriger les erreurs de transcription et de réintroduire les prononciations aphasiques éliminées par Whisper. La sortie de cette étape est un corpus (parole/écrit).

Les données d'AphasiaBank sont déjà transcrites, mais cela est fait dans un format particulier. Il est donc nécessaire de réécrire les transcriptions en français standard. L'exemple d'AphasiaBank et donc ajouté au corpus, ce qui fait un total de 1 h 0 min 47 s de parole aphasique. Cela est encore insuffisant pour entraîner un modèle profond. Cependant, il peut servir aux chercheurs qui souhaitent travailler sur l'aphasie de Broca. On rend donc disponible ce corpus.

1.2.2 Création et entraînement du modèle

Les données collectées n'étant pas suffisantes, il n'est pas possible d'entraîner un modèle de apprentissage profond (DL, de l'anglais : deep learning). Nous avons donc décidé de mettre en pause le développement de ce modèle jusqu'à ce que des données supplémentaires soient disponibles. Dans le but de faciliter l'accès à de telles données, nous avons mis notre corpus à la disposition des chercheurs. Pour le reste de ce projet, nous nous focalisons sur le modèle de traduction.

1.3 Traduction automatique neuronale

La deuxième partie de notre système (et celle qui réalise sa fonction principale) est le modèle de traduction. Ce modèle corrige la parole aphasique pour la rendre plus compréhensible. La procédure de sa création est décrite dans la Figure 1.3. Dans cette section, nous allons détailler les étapes de sa construction.

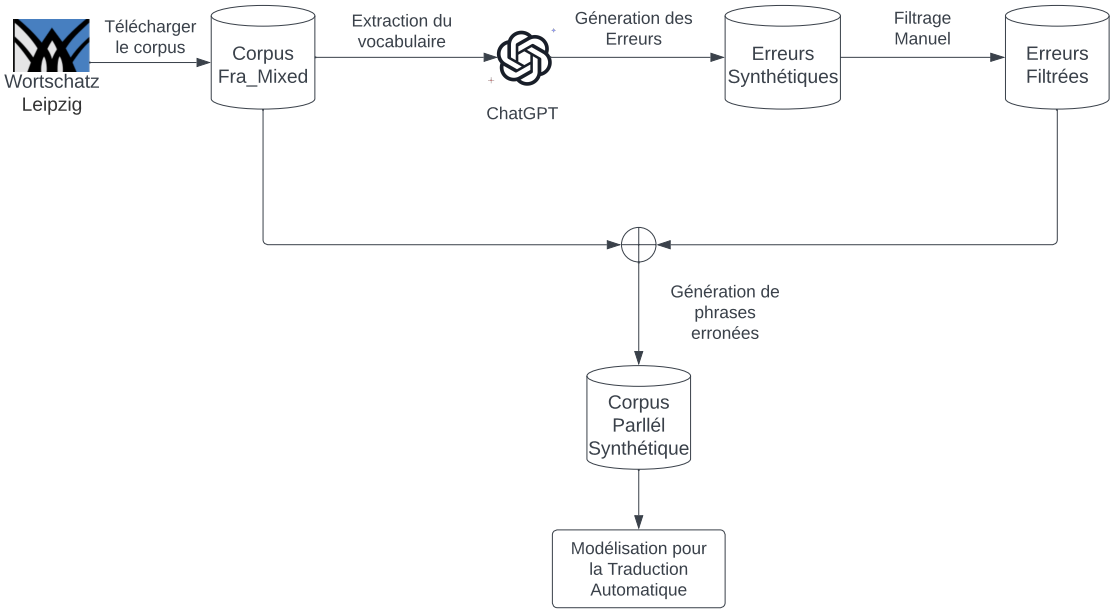


FIGURE 1.3 – Déroulement de la partie NMT.

1.3.1 Création d'un corpus parallèle

L'entraînement d'un modèle de traduction nécessite la présence d'un *corpus parallèle*, c'et-à-dire un corpus contenant les mêmes phrases dans plusieurs langues (dans notre cas, Français et Français aphasique). Cependant, un tel corpus n'existe pas. Il est donc nécessaire de le créer. La création d'un tel corpus nécessite la collecte d'un corpus de parole aphasique de taille suffisante. Ce corpus doit être ensuite traiter par des experts pour corriger les erreurs dedans. Or, nous avons déjà établie qu'un corpus de parole aphasique de taille suffisante n'existe pas. Ce chemin donc est infranchissable dans ce moment.

L'alternative proposée dans (SMAÏLI et al., 2022) — et celle que nous prenons — est de créer un corpus synthétique. c.–à.–d., partir d'un corpus de parole normale et le modifier pour introduire des erreurs similaires à celles trouvées dans la parole aphasique.

Corpus de parole normale

Nous avons utilisé le corpus `fra_mixed100k` de *Leipzig Corpora Collection* (Goldhahn et al., 2012). Il s'agit d'un corpus de 100000 phrases Françaises collectées de plusieurs sites web. Ces phrases sont diverses dans leur contenu, longueur, vocabulaire et structure grammaticale.

Extraction du vocabulaire et sélection des mots à modifier

Les erreurs causées par l'aphasie de Broca ne sont pas déterministes. Un individu qui en souffre ne se trompe pas sur tous les mots, ni de la même manière sur le même mot. Cependant, les erreurs ne sont pas uniformément réparties sur le vocabulaire. Naturellement, un tel individu à tendance à se tromper plus sur les mots "*difficiles*". Il est aussi plus susceptible de se tromper sur les mots qu'il utilise le plus souvent. On peut donc s'attendre à un biais pour les mots fréquents et difficiles.

Pour simuler ce biais dans notre corpus, nous avons extrait le vocabulaire du corpus. Puis, nous avons sélectionné les mots "*difficiles*". Nous avons considéré un mot "*difficile*" s'il est long (plus de 2 syllabes). Nous avons considéré ses mots dans l'ordre de leur fréquence dans le corpus (les 1000 premiers).

Génération et filtrage des erreurs synthétiques

Les mots sélectionnés à l'étape précédente sont donnés à chatGPT qui est chargé de générer 10 erreurs pour chaque mot dans le style d'un individu souffrant de l'aphasie de Broca. Le résultat de cette opération est une liste de 10000 couples (mot, erreur).

Ces couples sont filtrés manuellement pour supprimer les erreurs trop similaires au mot original (par exemple celle qui en diffèrent uniquement par la suppression d'une lettre) et les erreurs dissimilaires à celle produite par un individu aphasique. Cela a donné une moyenne de 5 erreurs retenues par mot.

Génération des phrases erronées

Le corpus parallèle est créé à partir du corpus original C et de l'ensemble des erreurs filtrés L de la façon suivante (voir Figure 1.4) :

1. Sélectionner de C les phrases qui contiennent au moins un mot présent dans L.
2. Pour chaque phrase sélectionnée, générer des variantes erronées en remplaçant les mots présents dans L par les erreurs correspondantes (si plusieurs mots existent, prendre toutes les combinaisons possibles).
3. Insérer les phrases générées dans le corpus parallèle avec la phrase de départ comme traduction.

Pour illustrer le processus de modification des phrases, prenons l'exemple suivant de la phrase "Maintenant que j'ai suffisamment d'argent, je peux m'acheter cet appareil photo."

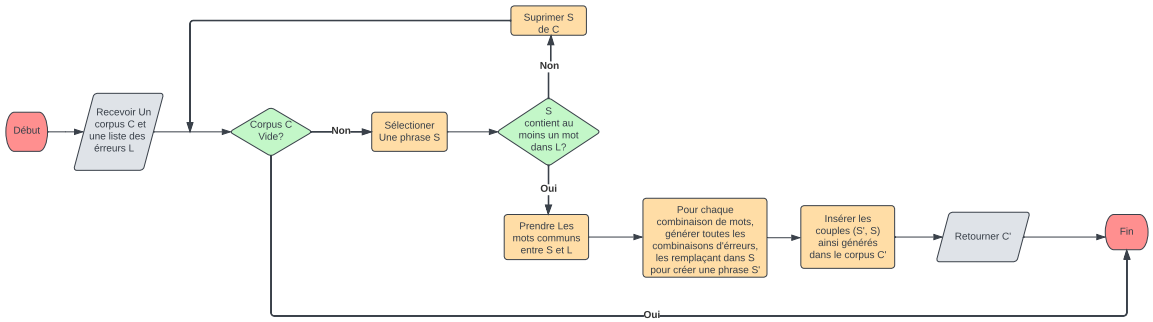


FIGURE 1.4 – Organigramme de la création du corpus parallèle.

avec

$$L = \begin{cases} \text{suffisamment} & \rightarrow & \text{fussamment} \\ \text{suffisamment} & \rightarrow & \text{suffimment} \\ \text{appareil} & \rightarrow & \text{apairel} \\ \text{appareil} & \rightarrow & \text{paparel} \\ \text{appareil} & \rightarrow & \text{pareil} \end{cases}$$

Pour cet exemple, nous obtenons 11 phrases modifiées : 2 où uniquement le mot “suffisamment” est modifié, 3 où uniquement le mot “appareil” est modifié et 6 où les deux mots sont modifiés.

Le corpus ainsi créé compte 282689 couples de phrases. Cela est bien suffisant pour entraîner un réseau de neurones. L’étape de modélisation peut donc être entamée.

1.3.2 Création du modèle

Après avoir construit le corpus, nous pouvons passer à la création du modèle de traduction. Dans cette section, nous allons détailler les étapes de sa construction. La procédure d’entraînement est discutée dans la section suivante.

Division du corpus

Le corpus est divisé en trois parties : entraînement, validation et test. La partie entraînement est utilisée pour optimiser les paramètres du modèle. Cela peut conduire au phénomène de sur-apprentissage (où le modèle est bien ajusté aux données d’entraînement, mais ne généralise pas bien). Pour détecter ce phénomène, nous utilisons la partie validation pour évaluer le modèle pendant l’entraînement. La partie test est réservée pour l’évaluation finale du modèle (après l’entraînement).

Tokenisation

La création du modèle implique plusieurs choix techniques. L'un des plus importants parmi ces choix est celui du tokeniseur. Dans Section **??**, nous avons introduit le tokeniseur byte pair encoding (BPE). Nous avons décidé d'utiliser ce tokeniseur basé sur BPE qui s'appelle " *WordPiece*". Il a été introduit par Google dans (Devlin et al., 2019).

Comme BPE, WordPiece part d'un vocabulaire réduit à l'alphabet du corpus puis, il l'élargit en fusionnant itérativement les tokens adjacents. Contrairement à BPE, la fusion n'est pas faite sur la base de la fréquence, mais sur celle de la fonction d'évaluation suivante :

$$\text{score}(x, y) = \frac{\mathbb{P}(x, y)}{\mathbb{P}(x)\,\mathbb{P}(y)} \tag{1.1}$$

où x et y sont deux tokens dans le vocabulaire à une itération donnée.

La division de la fréquence du couple par le produit de celles de ses composants a pour but de défavoriser la fusion des tokens fréquents (qui sont souvent des mots). Cela empêche la création de tokens plus longs qu'un mot et permet de conserver les tokens qui représentent des affixes communs (comme "im-" et "-able").

Nous avons opté pour une taille de vocabulaire de 5000 tokens pour la source et la cible. Des tokens spéciaux sont ajoutés au vocabulaire pour représenter la structure de la phrase. Ces tokens sont :

— `[BOS]` : début de la phrase ;
— `[EOS]` : fin de la phrase ;
— `[UNK]` : token inconnu (qui n'existe pas dans le vocabulaire) ;
— `[PAD]` : token de remplissage (utilisé pour ramener les phrases à la même longueur dans le cas de l'entraînement par lots).

À la sortie du tokeniseur, une opération de *numérisation* est effectuée. Il s'agit d'une application bijective entre les tokens et les entiers (que nous appelons "indices"). Cela permet de représenter les phrases par des suites d'entiers de longueur variable. Chose qui facilite leur représentation vectorielle. Le fait qu'elle soit bijective permet de reconstruire les phrases à partir de ces suites d'entiers produites par le modèle.

Représentation vectorielle des mots

Comme discuté dans la Section **??**, un plongement lexical est appliqué après la tokenisation. En effet, les indices retournés par le tokeniseur peuvent en principe être utilisés tels quels. Cela présente en outre deux problèmes majeurs. Le premier est que la plage des indices est aussi grande que la taille du vocabulaire. Cela garantit que les mots de grands indices écrasent les autres lors de l'entraînement. Le deuxième problème est que les indices ne sont pas structurés selon une quelconque relation sémantique. Bien que d'être le plus souvent basés sur la fréquence des tokens, les indices ne contiennent pas d'information sur leur distribution conjointe.

Un plongement lexical est une application $\Sigma \to \mathbb{R}^d$ où Σ est le vocabulaire (qui passe le plus souvent par les indices) et d est la dimension du plongement (que nous avons fixé

à 64) [1]. Cela permet de remédier aux problèmes cités ci-dessus. Le premier problème est résolu, car la norme des vecteurs de plongement peut être contrôlée. Le deuxième est réglé, car les plongements, étant des vecteurs, présentent une structure vectorielle. Il est possible de définir la similarité entre deux plongements en utilisant le produit scalaire. Cela permet de capturer les relations sémantiques entre les tokens en s'assurant que les tokens corrélés sont représentés par des vecteurs similaires (voir Figure 1.5).

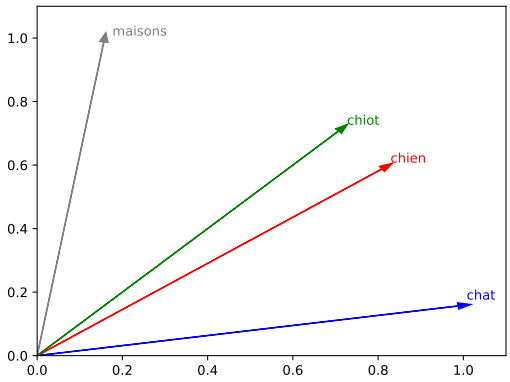


FIGURE 1.5 – Les plongements lexicaux de quelques mots avec $d = 2$.

Plusieurs algorithmes existent pour calculer les plongements lexicaux (voir Section **??**). Dans ce travail, nous utilisons une couche de plongement lexical simple. Il contient un tableau de $|\Sigma|$ vecteurs de dimension d. Elle est donc paramétrée par les composantes de ces vecteurs. Cela donne une matrice de paramètres W dont les lignes sont les vecteurs de plongement :

$$W = \overbrace{\left.\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1N} \\ w_{21} & w_{22} & \cdots & w_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \cdots & w_{dN} \end{bmatrix}\right\}}^{d=64} |\Sigma| = 5000 \tag{1.2}$$

qui a donc $|\Sigma| \times d = 5000 \times 64 = 320000$ paramètres. Pour calculer le plongement d'un token x d'indice i, il suffit de prendre la ligne $i + 1$ de W. La matrice W peut être optimisée comme n'importe quelle autre matrice de paramètres. Cela permet d'apprendre les plongements lexicaux à partir des données (PASZKE et al., 2019).

Architecture du modèle

Suite à l'étude bibliographique du Chapitre **??**, nous avons choisi d'utiliser un transformeur comme architecture de base pour notre modèle. Nous avons opté pour le transformeur de base décrit dans (VASWANI et al., 2017).

1. Il est commun dans la littérature de trouver des plongements qui sont des puissances de 2 (PASZKE et al., 2019 ; VASWANI et al., 2017). 64 nous semble être un bon compromis entre la dimension par défaut de 512 et la taille du vocabulaire.

Notre modèle est composé de 3 couches d'encodeurs et de 3 couches de décodeurs. Chaque couche a 4 têtes d'attention. La dimension de toutes les couches est de 64 et la fonction ReLU est utilisée comme fonction d'activation [2]. Pour l'encodage positionnel, nous avons choisi de l'apprendre plutôt que d'utiliser un encodage sinusoïdal comme celui décrit dans (Vaswani et al., 2017).

1.3.3 Entraînement

L'entraînement du modèle est un problème d'optimisation. Étant donné une fonction qui mesure la dissimilarité entre la sortie du modèle et la cible, le but est de trouver les valeurs des paramètres qui minimisent cette fonction.

Algorithmes d'optimisation

Plusieurs algorithmes d'optimisation sont utilisés pour entraîner les modèles de DL. La majorité d'entre eux sont basés sur l'algorithme du gradient. C'est le cas de l'algorithme d'optimisation Adam (Kingma & Ba, 2017) que nous avons utilisé. Il s'agit d'un algorithme itératif qui met à jour les paramètres du modèle à chaque itération. Son comportement est contrôlé par plusieurs hyperparamètres, mais le seul que nous avons modifié est le taux d'apprentissage η que nous avons initialisé à 3×10^{-4}, une valeur récurrente dans la littérature (Islam et al., 2022 ; Lu et al., 2023).

Pour accélérer l'entraînement, il est fait d'une manière *stochastique*. Cela signifie que la fonction de perte est estimée sur un sous-ensemble des données d'entraînement qu'on appelle un *lot*. La taille du lot peut avoir un grand impact sur la performance du modèle. Dans notre cas, nous avons utilisé des lots de 256 paires de phrases [3].

Contre-mesures au sur-apprentissage

Le sur-apprentissage est un problème courant dans l'entraînement des modèles de DL. Pour mitiger ce risque, nous avons utilisé plusieurs techniques. L'une d'entre elles est le *dropout*. Il s'agit d'une technique de régularisation qui consiste à ignorer aléatoirement une fraction p_{drop} des valeurs d'une couche.

Une autre méthode de régularisation est la majoration de la norme des vecteurs de plongement. Cela permet de contraindre les paramètres des couches de plongement lexical et par conséquent, de réduire sa puissance de représentation.

2. $\text{ReLU} : \mathbb{R} \to \mathbb{R}, x \mapsto \max(0, x)$.
3. La plus grande taille de lot compatible avec la mémoire de la carte graphique utilisée.

Réglage des hyperparamètres

Les hyperparamètres sont les paramètres du modèle qui ne sont pas appris durant l'entraînement. Dans notre cas, ces paramètres incluent les dimensions de plongement, la taille du vocabulaire, le nombre de couches du transformeur, le nombre de têtes d'attention, le *dropout* et la taille des lots d'entraînement. Leurs valeurs sont souvent fixées par l'utilisateur. Cependant, elles peuvent avoir un impact significatif sur les performances du modèle. Elles sont donc souvent choisies en explorant systématiquement l'espace des possibilités. Cette exploration peut se faire d'une manière exhaustive ou aléatoire.

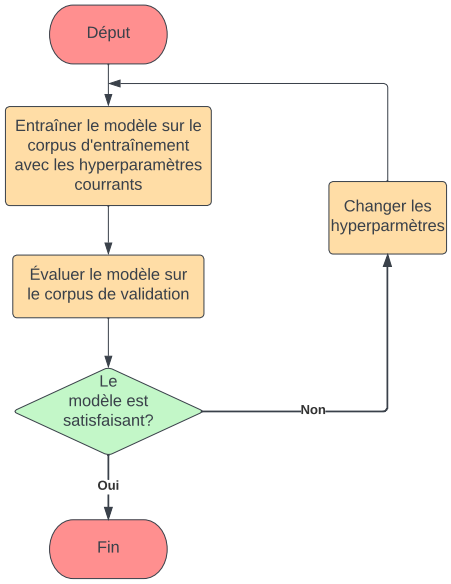


FIGURE 1.6 – Organigramme de la phase d'entraînement.

Le problème de réglage des hyperparamètres est aussi un problème d'optimisation. Or, il est souvent de nature discrète ou mixte, car les hyperparamètres ne sont pas nécessairement continus. Cela rend l'optimisation beaucoup plus difficile. La fonction objectif est généralement calculée à partir du jeu de données de validation.

Le processus d'entraînement dans sa totalité est illustré par la Figure 1.6. On note sur la figure que le réglage des hyperparamètres n'est effectué que si le modèle entraîné avec la combinaison initiale d'hyperparamètres n'est pas satisfaisant. Sinon la condition d'arrêt est atteinte et le modèle est utilisé pour la phase d'évaluation.

Il est important de noter que le corpus de test est particulièrement important dans le cas où le réglage des hyperparamètres est effectué. Dans ce cas, il est possible que le sur-apprentissage se produise simultanément sur le corpus d'entraînement et celui de validation. Le corpus de test est donc la seule manière d'obtenir une estimation non biaisée de la performance du modèle.

1.3.4 Évaluation et métriques

En ML, l'évaluation quantitative de la performance d'un modèle est une étape incontournable. Elle permet d'obtenir des mesures objectives de la qualité du modèle. Cela est d'autant plus important que le modèle est destiné à être déployé dans un système critique. Pour ce faire, il faut définir des métriques qui peuvent, à partir des prédictions du modèle et des valeurs réelles, fournir un nombre (où une liste de nombres) qui représente la qualité du modèle. La traduction automatique (MT, de l'anglais : machine translation) étant un exemple de problème de classification en grande dimension [4] (YANG et al., 2020), les métriques utilisées sont celles de la classification.

Entropie croisée

L'entropie croisée est une mesure de la dissimilarité de deux lois de probabilité (VASILEV et al., 2019). Pour deux lois de probabilité p et q sur le même espace probabilisable $(\Omega, \mathcal{A})$, leur entropie croisée $H(p, q)$ est définie par :

$$H(p, q) = H(p) + D_{KL}(p \parallel q) \tag{1.3}$$

où $H(p)$ est l'entropie de p et $D_{KL}(p \parallel q)$ est la divergence de Kullback-Leibler de p par rapport à q (BISHOP, 2006). Dans le cas de lois de probabilité discrètes, l'équation 1.3 devient :

$$H(p, q) = \mathop{\mathbb{E}}_{X \sim p}[-\log q(x)] = -\sum_{x \in \Omega} p(x) \log q(x) \tag{1.4}$$

où nous avons, par abus de notation, noté $p(x)$ et $q(x)$ les probabilités des événements $\{x\}$ par les lois p et q respectivement.

Quand l'entropie croisée est utilisée dans le cadre d'un problème de classification, on prend souvent comme p la loi de probabilité *cible*, c'est-à-dire celle des valeurs réelles. Dans notre cas, il s'agit de la loi du prochain token sachant la phrase source et les tokens précédents. On prend généralement une *loi de Dirac* [5], on parle dans ce cas d'encodage *one-hot*. Pour la loi q, la sortie du modèle (typiquement avec une fonction softmax) est prise. L'entropie croisée devient donc une mesure de la déviation de la sortie du modèle de la vérité terrain. Sa minimisation est donc un objectif naturel pour l'entraînement du modèle.

L'entropie croisée présente un avantage important par rapport aux autres métriques de classification : elle est *dérivable* par rapport aux paramètres du modèle. La Figure 1.7 l'illustre sur le cas d'un modèle de régression logistique en deux dimensions. On y voit que la surface qui représente l'entropie croisée est lisse. Elle est également convexe dans ce cas, mais cela n'est pas vrai pour un réseau de neurones.

Le fait d'être dérivable (contrairement à la majorité des alternatives) rend possible sa minimisation par un algorithme de gradient comme Adam. C'est pour cette raison que

4. Où les classes sont les tokens du vocabulaire de la langue cible.

5. Pour un espace probabilisable $(\Omega, \mathcal{A})$ et $a \in \Omega$, la loi de Dirac $\delta_a : \mathcal{A} \to \{0, 1\}$ est définie par : $\delta_a(X) = \begin{cases} 1 & \text{si } a \in X \\ 0 & \text{sinon} \end{cases}$

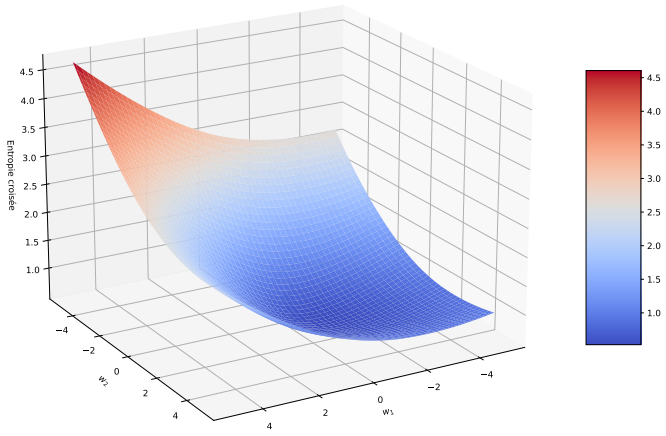


FIGURE 1.7 – Entropie croisée d'un classifieur linéaire binaire.

l'entropie croisée est la fonction de perte la plus utilisée en classification. Pour cette même raison, elle est utilisée dans notre système pour l'entraînement du modèle.

Cela étant dit, l'entropie croisée a le défaut de ne pas être interprétable. Comme elle prend ses valeurs dans l'intervalle $[0, +\infty]$, elle ne donne aucune information sur la différence relative entre deux modèles. Elle permet de déterminer si un modèle $\mathcal{M}$ est plus performant qu'un autre modèle $\mathcal{M}'$, mais elle ne nous dit rien sur le degré auquel $\mathcal{M}$ dépasse $\mathcal{M}'$. Pour cette raison, des métriques normalisées sont nécessaires pour comparer des modèles.

Exactitude, précision, rappel et mesure F_β

Une façon universelle d'évaluer la performance d'un modèle de classification est de regarder sa *matrice de confusion*. Il s'agit d'une matrice carrée M dont les lignes sont indexées par les classes réelles et les colonnes par les classes prédites. L'élément à la position (i, j) de la matrice est le nombre d'exemples de la classe i attribués par le modèle à la classe j. Un modèle parfait à donc une matrice diagonale pour matrice de confusion (SEBASTIAN & MIRJALILI, 2017).

Il est moins facile d'interpréter la matrice de confusion d'un modèle imparfait. Pour cette raison, il est utile d'en extraire des nombres directement interprétables. Il est possible de construire ces nombres à partir de la matrice de confusion en posant 3 questions assez naturelles :

(1) Quelle est la proportion d'exemples correctement classés ? C'est-à-dire, quelle est la proportion d'exemples pour lesquels la classe prédite est la même que la classe réelle ?

(2) Quelle proportion des exemples de la classe i est attribuée à la classe i ?

(3) Quelle proportion des exemples attribués à la classe j est réellement de la classe j ?

La réponse à la première question est donnée par *l'exactitude* du modèle. Il s'agit d'un nombre entre 0 et 1 qui mesure la similarité entre la matrice de confusion du modèle et la matrice diagonale du modèle parfait. L'exactitude est définie par :

$$\text{exactitude} = \frac{\operatorname{tr} M}{\operatorname{sum}(M)} = \frac{\sum_{i=1}^{n} M_{ii}}{\sum_{i=1}^{n}\sum_{j=1}^{n} M_{ij}} \tag{1.5}$$

elle donne une mesure de la performance globale du modèle, indépendamment des classes. Cela n'est pas toujours souhaitable, un exemple d'un cas problématique est celui d'un modèle qui prédit toujours la classe majoritaire. L'exactitude de ce modèle est minorée par la fréquence de cette classe. Dans le cas où les classes sont très déséquilibrées, un modèle sans aucune capacité à décerner les classes minoritaires peut avoir une exactitude élevée en attribuant simplement toutes les classes à l'une des classes majoritaires. Cela suggère un besoin de mesures spécifiques aux classes, ce qui est ce que nous obtenons en abordant les questions (2) et (3) (Sebastian & Mirjalili, 2017).

Les réponses des deux questions restantes sont parfaitement symétriques (comme le sont les questions elles-mêmes). Elles sont données respectivement par le *rappel* et la *précision* du modèle. Le rappel de la classe i est défini par :

$$\text{rappel}_i = \frac{M_{ii}}{\sum_{j=1}^{n} M_{ij}} \tag{1.6}$$

il nous permet de savoir, sachant la classe d'un exemple, la probabilité que le modèle le classe correctement. Une valeur élevée de rappel suggère que le modèle a tendance à bien identifier les éléments de la classe i (on dit qu'il a peu de *faux négatifs* pour cette classe). La précision de la classe j est définie par :

$$\text{précision}_j = \frac{M_{jj}}{\sum_{i=1}^{n} M_{ij}} \tag{1.7}$$

elle nous permet de savoir, sachant la prédiction du modèle, la probabilité qu'elle soit correcte. Une grande précision suggère que le modèle a tendance à ne pas attribuer la classe j à tort (on dit qu'il a peu de *faux positifs* pour cette classe). Il est possible d'obtenir à partir de la précision et du rappel des mesures globales comme l'exactitude. Il suffit pour cela de prendre la moyenne des précisions ou des rappels sur toutes les classes. Cela présente l'avantage de traiter équitablement toutes les classes (Sebastian & Mirjalili, 2017).

La mesure F_β permet d'agréger le rappel et la précision en une seule mesure qui contient l'information sur les faux positifs et les faux négatifs. Elle est paramétrée par un réel positif β qui peut être interprété comme le degré d'importance des faux positifs comparé aux faux négatifs. Un $\beta = 2$ signifie que les faux positifs sont pénalisés 2 fois plus que les faux négatifs (van Rijsbergen, 2002). Le score F_β est défini comme la *moyenne harmonique* de la précision et le rappel pondérée par $\alpha = \frac{1}{1+\beta^2}$:

$$F_\beta = \frac{1}{\frac{\alpha}{P} + \frac{1-\alpha}{R}} \tag{1.8}$$

où P et R sont respectivement la précision et le rappel du modèle. La valeur la plus courante de β est 1, on parle alors simplement de F_1-mesure.

Score BLEU

Toutes les métriques discutées jusqu'à présent sont des métriques de classification. Elles ne sont pas spécifiques à la tâche de traduction. En effet, elles mesurent la performance du modèle sur le niveau des tokens individuels. Elles ne nous apprennent rien sur sa performance au niveau des phrases entières.

Pour combler cette lacune, des métriques comme bilingual evaluation understudy (BLEU) (voir Section **??**) ont été conçues. Dans notre travail, nous avons choisi de l'utiliser comme objectif pour le réglage des hyperparamètres pendant la validation. Nous l'avons calculé sur les sorties du modèle avec *forçage de l'enseignement*. Cela signifie que la sortie est produite à partir de l'entrée et d'une version décalée d'elle-même avec un masque causal (voir Section **??**).

1.4 Conclusion

Dans ce chapitre, nous avons présenté les détails de la conception de la solution que nous avons proposé. Il s'agit d'un système à deux composantes principales : un modèle d'ASR et un modèle de traduction. Pour chacune de ces composantes, nous avons expliqué la procédure suivie pour l'acquisition et la préparation des données.

Dans le cas du modèle d'ASR, nous avons trouvé que les données ne sont pas suffisantes pour entraîner un modèle de qualité. Nous nous sommes donc contentés de la création d'un corpus de données qui peut être exploité par des futurs travaux.

Pour le cas de la traduction, nous avons créé un corpus synthétique de taille suffisante pour entraîner un modèle. Nous avons présenté les différentes décisions que nous avons pris pour la création et l'entraînement de ce modèle. Ces décisions incluent le choix de l'architecture et de l'algorithme d'entraînement, les valeurs des hyperparamètres, les métriques à mesurer et la possibilité d'optimisation des hyperparamètres.

Dans le chapitre suivant, nous présentons notre réalisation de la solution conformément à la conception proposée.

Chapitre 2

Réalisation

Dans le chapitre précédent, nous avons présenté les détails fonctionnels de la conception de notre solution. Le présent chapitre est consacré à la réalisation de cette dernière. Nous commençons par présenter les outils et technologies utilisés pour sa mise en place. Ensuite, nous présentons dans le détail les points importants de l'implémentation des différentes étapes de cette solution.

2.1 Outils et technologies

La création d'un système aussi sophistiqué que le nôtre est une tâche assez complexe. Elle comporte plusieurs étapes dont certaines nécessitent d'entraîner des réseaux de neurones ou de faire appel à des réseaux de neurones pré-entraînés par le biais d'interfaces de programmation d'application (API, de l'anglais : application programming interface). Pour faciliter certaines de ces étapes, nous avons exploité plusieurs outils et technologies open-source. Cette section est consacrée à la présentation de ces derniers.

2.1.1 Python

Python est un langage de programmation open-source interprété, orienté objet et multi paradigme. Introduit en 1991 par Guido van Rossum (« About Python », 2022), Python est aujourd'hui l'un des langages de programmation les plus populaires au monde. Le (« Stack Overflow Developer Survey », 2022) l'a classé comme la 4^{e} technologie la plus populaire au monde (48.07%) et la 6^{e} technologie la plus aimée (67.34%). La documentation officielle de Python le décrit comme suit :

> "Python est un langage de programmation puissant et facile à apprendre. Il dispose de structures de données de haut niveau et permet une approche simple, mais efficace de la programmation orientée objet. Parce que sa syntaxe est

> élégante, que son typage est dynamique et qu'il est interprété, Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur la plupart des plateformes."
>
> — « Le tutoriel Python », 2023

La popularité de Python fait qu'il y a une grande communauté de développeurs qui contribuent à son écosystème. En effet, le répertoire officiel de Python contient plus de 450 000 packages. Tous les programmes écrits dans le cadre de ce projet sont écrits en Python.

2.1.2 Jupyter

Jupyter est un environnement d'exécution interactif pour Python (et plusieurs autres langages). Il permet de créer des documents appelés "notebooks" qui combinent le code exécutable avec une documentation textuelle. Cela est réalisé en divisant le document en unités appelées des "cellules". Une cellule est associée à un langage qui peut-être écrit dedans. Si ce langage est un langage de programmation (Python, R ou Julia), la cellule est exécutable. Si ce langage est Markdown, elle ne l'est pas.

2.1.3 Google Colaboratory, Kaggle et Paperspace Gradient

Colaboratory est un service fourni par Google. Il donne à ses utilisateurs l'accès à un environnement d'exécution Jupyter hébergé sur le cloud. Il est possible d'y créer et d'y exécuter des notebooks sans installation ni configuration. Il offre également (et gratuitement) le choix entre les exécuter sur un CPU ou un GPU. La machine virtuelle qui héberge cet environnement est réinitialisée après 12 h. Cette limite peut être étendue en utilisant un compte payant.

Kaggle est une plateforme en ligne qui permet d'organiser des compétitions en data science. Dans ce cadre, elle donne accès à un environnement similaire à celui trouvé sur Google Colaboratory. Les GPU qu'elle offre ont 16 Go de mémoire. Le temps de calcul sur Kaggle est limité à 30 heures par semaine.

Paperspace est une plateforme du calcul cloud dédiée au ML et à l'intelligence artificielle. Elle permet de créer des machines virtuelles avec des configurations personnalisées. Paperspace possède une grande variété d'architectures matérielles. Des CPU et des GPU y sont disponibles, mais aussi des TPU et des IPU. Il est possible de personnaliser le nombre de CPU, GPU, TPU et IPU à inclure dans une machine, ainsi que le type et les caractéristiques des CPU et GPU.

Gradient est un service offert par Paperspace. Similairement à Google Colaboratory et Kaggle, il permet d'accéder à un environnement Jupyter hébergé sur le cloud. Cependant, il offre plus de flexibilité que ces derniers. Il permet de personnaliser d'une manière similaire aux machines virtuelles de Paperspace.

2.1.4 Anaconda

Anaconda est une distribution logicielle open-source qui regroupe Python, R et une multitude de leurs packages consacrés au calcul scientifique et aux data science. Anaconda permet de créer des environnements virtuels pour Python pour isoler les dépendances entre les projets. Chaque environnement possède sa propre installation de Python et de ses packages. Un gestionnaire de packages appelé "`conda`" est fourni comme alternative à `pip`.

2.1.5 CUDA

CUDA est une technologie de calcul parallèle développée par NVIDIA. Elle permet d'utiliser les GPU de NVIDIA pour accélérer les calculs parallélisables. CUDA est programmable en C++. Cependant, il existe des bibliothèques qui permettent de l'utiliser avec d'autres langages (y compris Python).

2.1.6 PyTorch

PyTorch est une bibliothèque open-source de DL pour Python. Elle permet de créer et d'entraîner rapidement et facilement des réseaux de neurones. Elle est développée par Facebook AI, mais elle rejoint la fondation Linux en 2022.

PyTorch possède plusieurs modules. Les plus importants sont :

— `torch.Tensor` : ce module fournit une implémentation du tenseur, un tableau multidimensionnel qui peut être manipulé mathématiquement d'une façon qui généralise les scalaires, les vecteurs et les matrices.

— `torch.nn` : ce module contient une implémentation des réseaux de neurones. La classe la plus importante dans ce module est `torch.nn.Module`. Il s'agit d'une classe abstraite qui doit être héritée pour créer un réseau de neurones. Des implémentations pour les architectures les plus courantes sont fournies dans ce module (réseau de neurones à convolutions (CNN, convolutio-

nal neural network), réseau de neurones récurrent (RNN, de l'anglais : recurrent neural network), transformeur, etc.).

— `torch.optim` : ce module offre un ensemble d'algorithmes d'optimisation.

— `torch.autograd` : ce module fournit une implémentation de la dérivation automatique (backward). Il permet de calculer les gradients des paramètres en construisant un graphe de calcul puis en le traversant pour évaluer les dérivées.

Toutes les opérations de PyTorch peuvent être exécutées sur un GPU en utilisant CUDA. Cela rend très rapide l'entraînement des réseaux de neurones.

L'écosystème de PyTorch est très riche. Plusieurs packages sont construits là-dessus par la communauté ainsi que par l'équipe de PyTorch. Parmi les packages `torchtext` et `torchdata` nous sont les plus utiles. Le premier nous offre des objets utiles pour le traitement de texte en langage naturel (tokeniseurs, jeux de données populaires, modèles de langage, etc.). Le second implémente les fonctionnalités de manipulation de données. Les deux classes les plus importantes dans ce module sont `torchdata.Dataset` et `torchdata.DataLoader`.

2.1.7 Lightning

Lightning.AI est une plateforme pour créer, entraîner et déployer des modèles de DL. La bibliothèque `lightning` distribuée contient 3 packages (voir Figure 2.1) :

— `lightning.pytorch`

— `lightning.fabric`

— `lightning.applications`

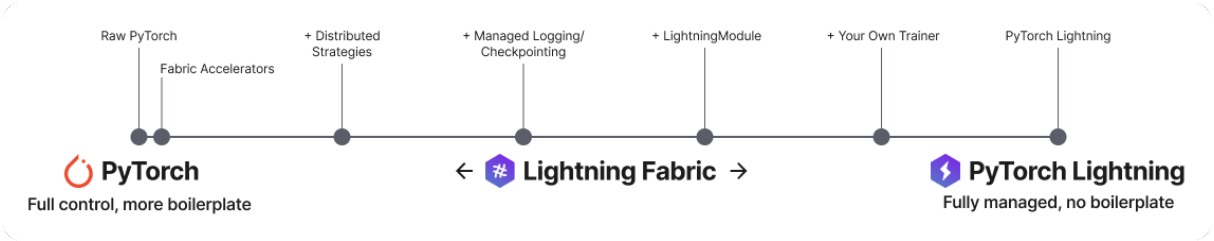


FIGURE 2.1 – Composantes de la bibliothèque `lightning` (FALCON & THE PYTORCH LIGHTNING TEAM, 2019).

`lightning.pytorch` (aussi appelé `pytorch-lightning`) est le seul que nous avons utilisé. Il permet d'étendre PyTorch pour faciliter la création et l'entraînement des modèles. Les fonctionnalités qu'il offre incluent les rappels de fonctions et la journalisation.

Lightning.AI distribue un quatrième package en dehors de la bibliothèque `lightning` : `torchmetrics`. Ce dernier implémente plus de 90 métriques d'évaluation, dont l'exactitude et le score BLEU. Nous avons fait usage de ce package pour calculer ces métriques.

2.1.8 Weights & Biases

Weights & Biases est une plateforme de Machine Learning Operations (MLOps). Son but est de faciliter le processus de développement de journalisation des expériences, de gestion des versions de jeux de données et de collaboration entre les membres d'une équipe.

l'API de Weights & Biases est accessible via un module Python `wandb` ou via une interface en ligne de commande du même nom. Elle offre la possibilité de journaliser les métriques d'évaluation, les hyperparamètres, les graphes de calcul, etc. Elle est intégrée avec `lightning` à travers la classe `lightning.pytorch.loggers.WandbLogger`.

2.1.9 Hugging Face

Hugging Face est une entreprise qui se spécialise en intelligence artificielle. Elle distribue gratuitement plusieurs bibliothèques open-source et des modèles pré-entraînés.

Parmi ces bibliothèques, nous avons utilisé `tokenizers` qui permet de créer des tokeniseurs et `evaluate` qui implémente plusieurs métriques d'évaluation (dont la perplexité). Nous l'avons utilisé plutôt que `torchmetrics`, car elle permet d'utiliser les modèles pré-entraînés dans Hugging Face Hub.

2.1.10 Open AI

Open AI est une entreprise de recherche en intelligence artificielle. C'est cette entreprise qui a créé les modèles generative pre-trained transformer (GPT) et Whisper discutés dans le chapitre **??**.

Nous avons fait appel à l'API d'Open AI pour utiliser chatGPT, une version de GPT-3 qui est entraînée sur des conversations. Nous l'avons utilisé pour générer les erreurs synthétiques. Nous avons également utilisé Whisper pour la transcription des vidéos collectées, mais nous l'avons utilisé localement, car il est open-source.

2.1.11 Pynecone

Pynecone est une bibliothèque qui permet de développer des sites web entièrement en Python (sans besoin d'écrire du code HTML, CSS ou JavaScript). Nous l'avons utilisé pour créer une interface web pour le modèle de correction d'erreurs.

2.1.12 Autres bibliothèques

La Table 2.1 donne une liste des autres bibliothèques dont nous avons fait un usage secondaire. Une liste complète des bibliothèques Python installées dans notre environnement est donnée par l'Annexe A.

Bibliothèque	**Description et utilisation**
`pytorch_memlab`	Diagnostique de l'utilisation du mémoire GPU par CUDA
`PyHyphen`	Division des mots en liste de syllabes
`torchview`	Dessin des graphes de calcul des modules de PyTorch
`PyYAML`	Chargement et écriture des fichiers `yaml` en Python
`beautifulsoup4`	Analyse syntaxique des pages HTML
`Requests`	Utilisation du protocole HTTP
`scikit_learn`	Division du jeu de données
`pandas`	Division du jeu de données
`python-dotenv`	Chargement des variables d'environnement
`tqdm`	Visualisation de l'état d'avancement d'une boucle

TABLE 2.1 – Bibliothèques auxiliaires.

2.2 Création des corpus

Dans cette section, nous donnons les détails de l'implémentation de la première étape de notre solution, à savoir la création des corpus. Nous commençons par le corpus de la partie ASR. Ensuite nous passons à celui de la partie MT.

2.2.1 Reconnaissance automatique de la parole

Pour la création du corpus de la partie ASR, nous avons commencé par repérer les vidéos pertinentes sur YouTube et Vimeo. Le résultat de cette opération est une liste de 3 vidéos qui n'ont pas déjà été transcrites.

Après cela, nous avons divisé chaque vidéo en plusieurs segments en fonction de la personne qui parle. Nous avons organisé ces segments en un fichier `url.yaml` dont un extrait est donné ci-dessous.

```
allo-docteurs:
    url: https://www.youtube.com/watch?v=rqoSKafN3aw
    is_broca: yes
    segments:
        jean-dominique:
            - [2:00, 5:40]
            - [8:20, 10:02]
            - [16:36, 18:00]
```

```
            - [20:36, 22:48]
            - [23:17, 23:39]
      raquel:
            - [10:20, 11:20]
            - [11:37, 11:52]
            - [12:24, 13:50]
hug:
    url: https://www.youtube.com/watch?v=d4Cybwx3sHk
    is_broca: yes
    segments:
        marie:
            - [34, 1:07]
            - [6:42, 6:47]
        jean-pierre:
            - [7:57, 8:57]
```

Une fois préparé, ce fichier est passé à un script Python qui se charge de télécharger les vidéos (en utilisant `youtube-dl`) et de les découper en segments. Ces segments sont ensuite transcrits en utilisant l'interface ligne de commande de Whisper.

2.2.2 Traduction automatique

Pour la création du corpus de la partie MT, nous avons interrogé l'API d'Open AI, plus précisément, le modèle `gpt-3.5-turbo` qui est celui derrière chatGPT (voir Extrait de code 2.1). Nous avons passé 3 paramètres à chatGPT :

— `messages` : une liste d'interactions décrivant l'histoire de la conversation. Chaque interaction est un dictionnaire contenant les clés `role` et `content`. La clé `role` représente l'identité de l'interlocuteur. Elle admet 3 valeurs : `user`, `bot` et `system`. Le message `system` est utilisé pour configurer le comportement de chatGPT pour le reste de la conversation.

— `timeout` : le temps maximal que chatGPT peut prendre pour générer une réponse (en secondes).

— `n` : le nombre de réponses à générer.

Cette opération peut échouer à cause des limitations de l'API d'Open AI. Si le nombre de requêtes dépasse le quota autorisé, l'API renvoie une erreur. Dans ce cas, nous attrapons l'erreur et nous réessayons après une seconde. Dans le cas contraire, nous passons au mot suivant. À la fin de la boucle, les erreurs sont enregistrées dans un fichier `errors.yaml`.

```python
import os
import time

import openai
from yaml import Dumper, dump

openai.api_key = os.getenv("openai")

system_prompt = (
    "You are a linguisitics researcher. "
```

```
    + "you are trying to understand the speaking patterns "
    + "of people with aphasia. "
    + "You will be given a french word. "
    + "You must try to modify it in the same way "
    + "a french speaker with aphasia would."
    + "Your response should be a single word, "
    + "with no punctuation, nor line breaks."
)

num_errors_per_word = 10
errors = {}
for word in words:
    prompt = (
        f'Deforme le mot "{word}" a la facon dont le ferait '
        + "un aphasique de Broca. Repond avec un seul mot."
    )

    success = False
    ntries = 0
    while not success:
        try:
            responses = openai.ChatCompletion.create(
                model="gpt-3.5-turbo",
                messages=[
                    {"role": "system", "content": system_prompt},
                    {"role": "user", "content": prompt},
                ],
                timeout=100,
                n=num_errors_per_word,
            )
        except openai.error.RateLimitError:
            ntries += 1
            print(f"Retry {ntries}")
            time.sleep(1)
            continue
        success = True
    errors[word] = list(
        response_object["message"]["content"]
        for response_object in responses["choices"]
    )

with open("errors.yaml", "x", encoding="utf-8") as f:
    dump(errors, f, Dumper, allow_unicode=True, sort_keys=False)
```

Extrait de code 2.1 – Génération des erreurs avec chatGPT.

Le fichier `errors.yaml` est combiné avec le corpus pour produire un corpus parallèle synthétique. Cette procédure est illustrée par l'extrait de code 2.2. Pour chaque phrase, les mots modifiables sont mis dans une liste `corruptable_words`. À partir de cette liste, une deuxième liste `corruptions` est générée qui contient toutes les combinaisons de modifications possibles. Ces modifications sont donc appliquées à la phrase d'origine pour produire des phrases corrompues. Les phrases corrompues (différentes de la phrase d'origine) sont ajoutées au corpus parallèle qui est retourné.

```
from itertools import product


```

```python
def get_corrupted_sentences(corpus, errors):
    result = {}
    for sentence in corpus:
        variants = []
        corruptable_words = [
            word
            for word in errors  # The keys of the errors dictionary
            if word in sentence.split(" ")  # Only those in the sentence
        ]
        corruptions = list(
            product(  # Cartesian product of all possible corruptions
                *[
                    [(word, word)]  # Identity corruption
                    + [
                        (word, error) for error in errors[word]
                    ]  # Modification corruptions
                    for word in corruptable_words
                ]
            )
        )
        for corruption in corruptions:
            splt = sentence.split(" ")
            for word, error in corruption:
                splt[splt.index(word)] = error
            if splt != sentence.split():
                variants.append(" ".join(splt))
        result[sentence] = set(variants)
    return result
```

Extrait de code 2.2 – Création du corpus parallèle synthétique.

Il est important de souligner que les erreurs sont divisées en 3 parties (entraînement, validation et test) avant d'être combinées avec le corpus. Cela a pour but d'éviter le sur-apprentissage à cause de fuites de données, c'est-à-dire que les règles de modification soient les mêmes dans les 3 parties même si les phrases sont différentes.

2.3 Création du modèle de traduction automatique

La section précédente fournit une vue de la procédure de création d'un corpus parallèle pour la MT. Dans cette section, nous exploitons ce corpus pour créer et entraîner un modèle de NMT.

2.3.1 Création du modèle

Comme discuté dans Section 2.1, nous avons choisi d'utiliser PyTorch et PyTorch Lightning pour la partie DL. La classe `Transformer` qui représente notre modèle hérite donc de la classe `lightning.pytorch.LightningModule` qui elle-même hérite de la classe `torch.nn.Module`. L'Extrait de code 2.3 montre la méthode d'initialisation de cette classe.

```python
def __init__(
    self,
```

```
    d_model: int,
    nhead: int,
    source_vocab_size: int,
    target_vocab_size: int,
    source_pad_idx: int,
    num_encoder_layers: int,
    num_decoder_layers: int,
    max_len: int,
    dim_feedforward: int,
    dropout: float,
    lr: float,
):
    super().__init__()

    self.input_embedding = nn.Embedding(source_vocab_size, d_model)
    self.input_position_embedding = nn.Embedding(max_len, d_model)

    self.output_embedding = nn.Embedding(target_vocab_size, d_model)
    self.output_position_embedding = nn.Embedding(max_len, d_model)

    self.transformer = nn.Transformer(
        d_model,
        nhead,
        num_encoder_layers,
        num_decoder_layers,
        dim_feedforward,
        dropout,
    )

    self.linear = nn.Linear(d_model, target_vocab_size)
    self.dropout = nn.Dropout(dropout)
    self.source_pad_idx = source_pad_idx

    self.lr = lr
```

Extrait de code 2.3 – Méthode d'initialisation d'un transformeur.

La méthode `__init__` prend en argument les hyperparamètres du modèle (`d_model`, `nhead`, `num_encoder_layers`, `num_decoder_layers`, `dim_feedforward` et `dropout`) ainsi que des paramètres de configuration de l'entraînement et de l'inférence (`source_vocab_size`, `target_vocab_size`, `source_pad_idx`, `max_len` et `lr`).

L'appelle à la méthode `super().__init__()` a l'effet de construire par défaut un `LightningModule`. Les lignes qui suivent cette instruction permettent d'initialiser ce module en définissant les couches qui le composent. Parmi ces couches, la plus importante et `self.transformer`, un objet de la classe `nn.Transformer`. Tous les autres modules sont des couches de prétraitement (encodage positionnel) ou de post-traitement (projection linéaire). La Figure 2.2 montre le graphe de calcul du modèle résultant.

En utilisant les valeurs proposées dans le Chapitre 1 pour les hyperparamètres :

$$
\begin{aligned}
d_{\text{model}} &= d_{\text{FFN}} = 64 \\
N_{\text{head}} &= 4 \\
N_{\text{encoder}} &= N_{\text{decoder}} = 3
\end{aligned}
$$

nous obtenons un modèle dont la Table 2.2 résume les paramètres par couche.

	Name	Type	Params
0	input_embedding	Embedding	320 K
1	input_position_embedding	Embedding	6.4 K
2	output_embedding	Embedding	320 K
3	output_position_embedding	Embedding	6.4 K
4	transformer	Transformer	201 K
5	transformer.encoder	TransformerEncoder	75.8 K
6	transformer.decoder	TransformerDecoder	126 K
7	linear	Linear	325 K
8	dropout	Dropout	0

1.2 M	Paramètres entraînables
0	Paramètres non entraînables
1.2 M	Paramètres totaux
4.719	Taille totale estimée (en Mo)

TABLE 2.2 – Résumé du modèle.

2.3.2 Entraînement

Pour entraîner le modèle, un objet *entraîneur* de la classe `lightning.pytorch.Trainer` est utilisé. Pour configurer sans comportement, plusieurs paramètres sont utilisés (voir Extrait de code 2.4

— `max_epochs` : le nombre maximal des passes sur le corpus d'entraînement.

— `deterministic` : un indicateur booléen, s'il est mis à `vrai`, des algorithmes déterministes sont utilisés pour la reproductibilité.

— `logger` : le journaliseur à utiliser pour garder trace des métriques. `WandbLogger` utilise *Weights & Biases* pour la journalisation.

— `callbacks` : une liste de rappels de fonctions à effectuer à la fin de chaque époque. `early_stopping` implémente l'arrêt précoce et `checkpoint` permet de sauvegarder le meilleur modèle.

— `gradient_clip_val` : un majorant sur la norme du vecteur gradient. Son but est d'éviter l'explosion des gradients.

Pour lancer l'entraînement, il suffit d'appeler la méthode `fit` de l'entraîneur comme suit : `trainer.fit(model, datamodule=dm)`.

```
trainer = Trainer(
    max_epochs=epochs,
    deterministic=True,
    logger=WandbLogger(project="project-name")
    callbacks=[early_stopping, checkpoint],
    gradient_clip_val=1.0,
)
```

Extrait de code 2.4 – Creation de l'entraîneur.

Le paramètre `datamodule` passé à la méthode `Trainer.fit` est un objet de classe `lightning.pytorch.LightningDataModule`. Il s'agit d'une classe d'utilité qui implé-

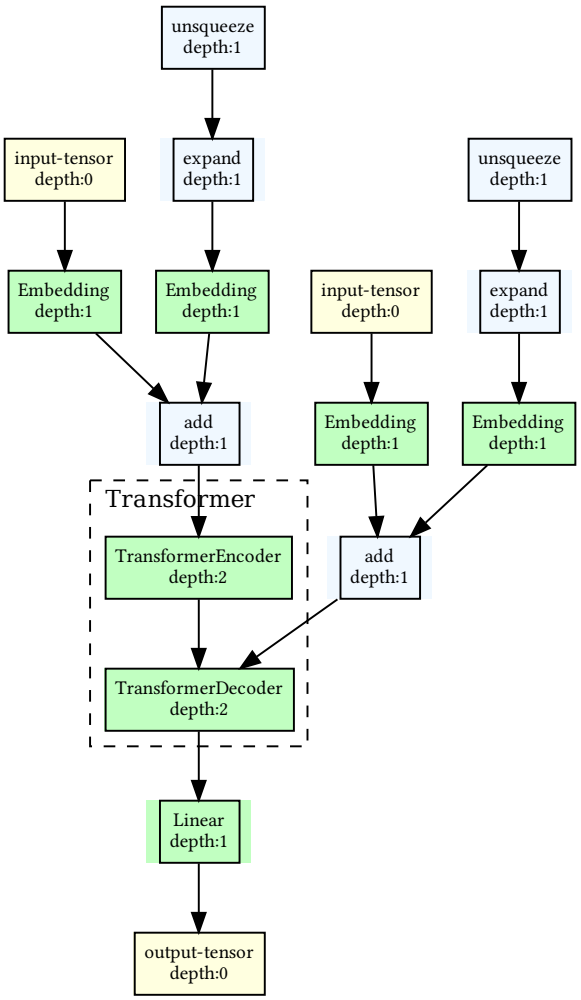


FIGURE 2.2 – Graphe de calcul du modèle définit dans la classe `Transformer` (profondeur = 2).

mente des fonctions de chargement de données. Ses 3 méthodes `train_dataloader`, `val_dataloader` et `test_dataloader` retournent respectivement les chargeurs d'entraînement, de validation et de test. L'entraîneur se charge de les appeler au moment opportun.

2.3.3 Réglage des hyperparamètres

Pour cette tâche, nous avons utilisé *Sweeps*, un outil fourni par Weights & Biases. Pour l'utiliser, il faut créer une configuration qui définit l'espace et la stratégie de recherche pour les hyperparamètres. Trois stratégies de recherche sont offertes :

— Grid : une recherche exhaustive sur l'espace des possibilités. Cette méthode garantit l'optimalité du résultat, mais elle est trop coûteuse et ne marche qu'avec un espace

de recherche fini.

— Random : une recherche aléatoire sur l'espace de possibilité. Cette méthode ne donne pas de garanties sur la qualité des hyperparamètres qu'elle produit. Cependant, elle est beaucoup plus rapide que la recherche exhaustive (car il est possible de définir le nombre des combinaisons à explorer) et elle peut être utilisée sur un espace infini.

— Bayesian : une méthode d'amélioration itérative. Sa première itération est identique à la recherche aléatoire, mais après elle utilise l'information sur le dernier essai pour informer les choix des paramètres du suivant. Elle converge généralement vers la solution optimale.

Pour définir l'espace de recherche, un dictionnaire ou un fichier `yaml` est utilisé. L'objet de configuration possède 3 clés obligatoires :

— `methode` : la stratégie de recherche.

— `metric` : la métrique à optimiser. Il faut fournir le nom de la métrique, le mode d'optimisation (`maximize` ou `minimize`) et éventuellement une cible à atteindre.

— `parameters` : un dictionnaire qui contient la définition de l'espace des paramètres. Chaque paramètre est défini par un couple clé-valeur. La clé est le nom du paramètre et la valeur est un dictionnaire qui contient les informations suivantes :

  — Si la valeur du paramètre est fixée, il suffit de fournir sa valeur pour la clé `value`.

  — Si le paramètre a un nombre fini de valeurs possibles, il faut fournir la liste de ces valeurs pour la clé `values` et éventuellement sa loi de probabilité pour la clé `probabilities`. Si cette dernière n'est pas fournie, la loi uniforme est utilisée.

  — Si le paramètre est continu, il faut fournir une loi de probabilité pour la clé `distribution` en donnant le nom de la loi et ses paramètres.

Ci-dessous un exemple de configuration pour une recherche bayésienne. La métrique à maximiser est `val/bleu_score` et sa valeur cible est de 99. Les paramètres `lr` et `dropout` sont continus et suivent une loi log-uniforme sur les intervalles $[10^{-5}, 10^{-1}]$ et $[0.1, 0.5]$ respectivement. Le paramètre `batch_size` est fixé à 256.

```
method: bayes
metric:
  name: val/bleu_score
  goal: maximize
  target: 99
parameters:
  lr:
    distribution: log_uniform_values
    min: 1.e-5
    max: 1.e-1
  dropout:
    distribution: log_uniform_values
```

```
    min: 0.1
    max: 0.5
  batch_size:
    value: 256
```

Une fois la configuration définie, il suffit de créer une Sweep en appelant la méthode `wandb.sweep`, puis de lancer la recherche en appelant la méthode `wandb.agent`. Cette méthode prend en paramètre l'identifiant de la Sweep, une fonction qui définit l'entraînement et le nombre d'essais à effectuer (voir Extrait de code 2.5).

```python
with open("config/sweep.yaml") as cfg:
    sweep_cfg = load(cfg, Loader)

sweep_id = wandb.sweep(sweep_cfg, project="project-name")
wandb.agent(sweep_id, train, count=10)
```

Extrait de code 2.5 – Création et lancement d'une Sweep.

2.4 Interface utilisateur

Dans l'absence d'un système d'ASR, il est impossible de créer un système text-to-speech complet. Une autre interface est donc nécessaire pour utiliser le modèle. Pour cette fin, nous avons développé une interface web de traduction.

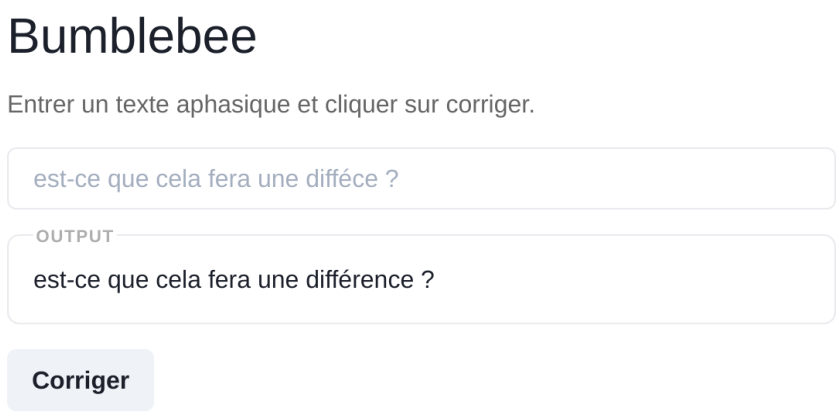


FIGURE 2.3 – Interface web de traduction.

La Figure 2.3 montre l'interface que nous avons développée. Nous l'avons nommée *Bumblebee*[1]. L'utilisateur peut saisir un texte aphasique dans la zone de texte en haut et cliquer sur le bouton *Corriger*. La correction s'affiche dans la zone de texte en bas. La figure montre un exemple de correction pour la phrase : "*est-ce que cela fera une différence ?*" où le mot "*différence*" est remplacé par "*difféce*".

1. Une référence à un personnage de la série *Transformers* qui souffre de mutisme, mais qui reprend la parole.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les détails de la réalisation de notre solution. Nous y avons listé les outils, technologies et bibliothèques logicielles utilisés pour sa mise en place. Ensuite, nous nous sommes étalés sur les étapes menées pour accomplir cette tâche. Dans le chapitre suivant, nous présentons les résultats obtenus et nous les discutons.

Chapitre 3

Tests et résultats

Dans le chapitre précédent, nous avons présenté les détails de la réalisation de notre solution. Le présent chapitre est consacré à la présentation des résultats obtenus. Dans ce but, nous présentons les différents tests que nous avons effectués, les résultats obtenus et la signification de ces derniers.

3.1 Erreurs générées

Parmi les erreurs générées par chatGPT, celles qui ressemblent le plus à des erreurs humaines ont été manuellement sélectionnées. Le résultat de cette sélection est une liste de 217 mots avec une moyenne de 5 erreurs retenues par mot (1104 erreurs en termes absolus). Ces erreurs ont été analysées et classées en 4 catégories :

— des suppressions : de lettres ou de syllabes,

— des ajouts : de lettres ou de syllabes,

— des substitutions : de lettres ou de syllabes,

— des transpositions : de lettres ou de syllabes.

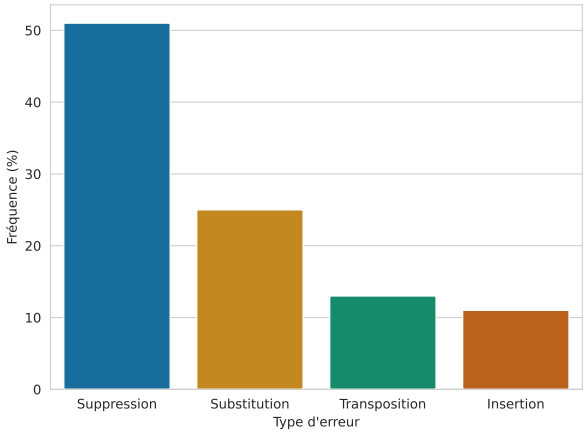


FIGURE 3.1 – Fréquences des catégories d'erreurs

Les fréquences de ces erreurs (pour les 32 premiers mots qui ont 327 modifications) sont présentées dans la Figure 3.1. Sur la base de ces fréquences, nous avons créé une fonction qui génère pour un mot donné, des erreurs qui suivent les mêmes fréquences (voir le code 3.1).

```python
def corrupt_word(
    word,
    p_remove=0.51,
    p_substitute=0.25,
    p_transpose=0.13,
    p_insert=0.11,
    p_skip=0.5,
    all_syllables=None,
):
    from random import seed, randint, random
    from hyphen import Hyphenator

    hyphenator = Hyphenator("fr_FR")
    syls = hyphenator.syllables(word)

    # skip words that are too short
    if len(syls) < 3:
        return word

    # skip all words with probability p_skip
    if random() < p_skip:
        return word

    # remove a syllable with probability p_remove
    if random() < p_remove:
        idx = randint(0, len(syls) - 1)
        del syls[idx]

    # substitute a syllable with probability p_substitute
    if random() < p_substitute:
        idx1 = randint(0, len(syls) - 1)
        syls[idx] = choice(all_syllables)

    # transpose two syllables with probability p_transpose
    if random() < p_transpose:
        idx1 = randint(0, len(syls) - 1)
        idx2 = randint(0, len(syls) - 1)
        syls[idx1], syls[idx2] = syls[idx2], syls[idx1]

    # insert a syllable with probability p_insert
    if random() < p_insert:
        idx = randint(0, len(syls) - 1)
        syls.insert(idx, choice(all_syllables))
    return "".join(syls)
```

Extrait de code 3.1 – Génération des erreurs pour un mot

Les erreurs générées par cette fonction sont similaires aux erreurs générées par chatGPT (par exemple, maintenant → temain | tenant, entendu → enten | tendu | tenendu.). Cependant, certaines parmi elles ne sont pas prononçables (par exemple, maintenant → nantmain, simplement → mentple). Pour cette raison, nous avons décidé de ne pas les utiliser dans le corpus. Cela étant dit, il nous paraît intéressant d'explorer des méthodes de

filtrage de ces erreurs. Si réussies, elles permettent de générer des erreurs plus rapidement et plus facilement que par chatGPT.

3.2 Corpus créé

Après avoir construit le corpus parallèle en suivant la démarche décrite dans Sections 1.1 et 2.2.2, il est nécessaire d'évaluer sa qualité. Pour cela, nous avons choisi deux métriques : le score BLEU (voir Section **??**) et la perplexité.

3.2.1 Perplexité

La perplexité est une mesure de la qualité d'un modèle de langue par rapport à un corpus. Si la qualité du modèle est connue (et bonne), la perplexité est une mesure de la qualité du corpus. La perplexité du corpus $\mathcal{C}$ par rapport au modèle de langue $\mathcal{M}$ est définie comme la moyenne géométrique des probabilités des phrases du corpus (voir l'équation 3.1).

$$\text{perplexité}(\mathcal{C}, \mathcal{M}) = \prod_{s \in \mathcal{C}} \mathcal{M}(s)^{-\frac{1}{|\mathcal{C}|}} \tag{3.1}$$

En prenant le logarithme de la perplexité, on retrouve l'entropie croisée de la loi de probabilité donnée par $\mathcal{M}$ et celle induite par $\mathcal{C}$ [1]. Elle mesure ainsi la dissimilarité entre ces deux lois (voir Section **??**). La perplexité des phrases générées (pour simuler l'aphasie de Broca) est une mesure de la dissimilarité entre ses phrases et le français courant.

	french	aphasia
`gpt-2`	392.07	566.07
`gpt-2-large`	182.20	417.30
`gpt-fr-cased-small`	147.84	1357.31
`gpt-fr-cased-base`	123.93	1023.12

TABLE 3.1 – Perplexité des phrases du corpus par rapport à différents modèles de langue.

Pour calculer la perplexité, nous avons utilisé quatre modèles de langage basés sur GPT-2 et hébergés sur Hugging Face Hub :

- `gpt2` : ce modèle compte 124 M de paramètres et a été entraîné sur 40 Go de texte en plusieurs langues.
- `gpt2-large` : ce modèle compte 774 M de paramètres et a été entraîné sur le même corpus que `gpt2`.
- `gpt-fr-cased-small` : ce modèle compte 124 M de paramètres et a été entraîné sur un corpus de textes en français d'une taille non précisée.
- `gpt-fr-cased-base` : ce modèle compte 1.017 G de paramètres et a été entraîné sur le même corpus que `gpt-fr-cased-small`.

1. À une constante multiplicative dépendante de la base du logarithme près

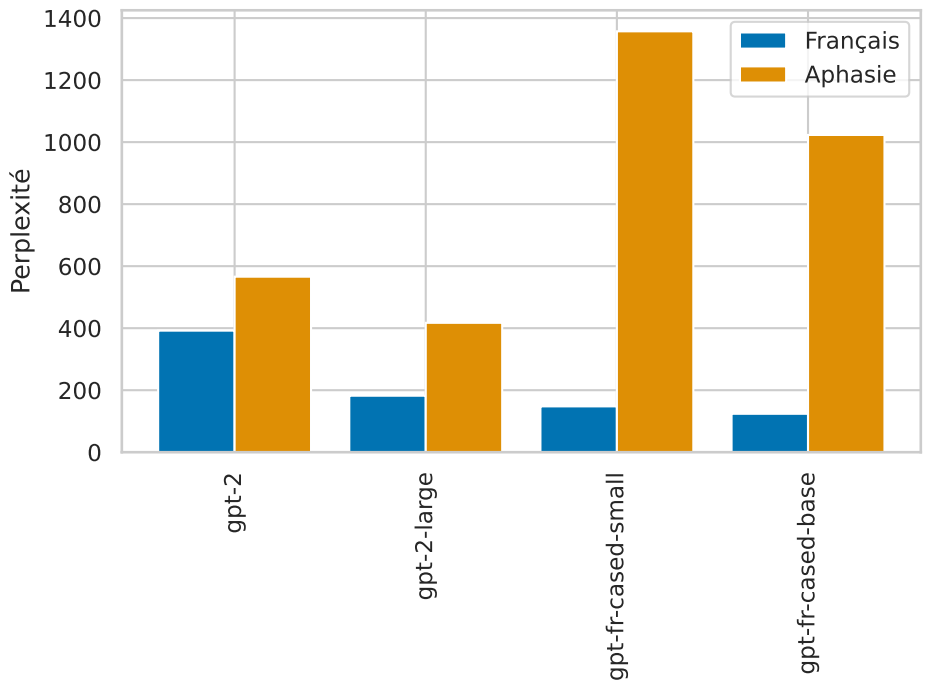


FIGURE 3.2 – Perplexité des phrases du corpus par rapport à différents modèles de langue.

Le code utilisé pour calculer la perplexité est donné par l'Extrait de code 3.2. Ce code a été exécuté sur Google Colaboratory avec une instance GPU munie de 16 Go de mémoire vive.

```python
import pandas as pd
import evaluate

# Load perplexity metric
perplexity = evaluate.load("perplexity", module_type="measurement")

# Load data
df = pd.read_csv("val.csv")
fr = df.french.tolist()
aph = df.aphasia.tolist()

# Compute perplexities
fr = perplexity.compute(
  model_id='asi/gpt-fr-cased-base',
  add_start_token=False,
  data=fr
)
aph = perplexity.compute(
  model_id='asi/gpt-fr-cased-base',
  add_start_token=False,
  data=aph
)

print(f'{fr["mean_perplexity"]=:.2f}, {aph["mean_perplexity"]=:.2f}')
```

Extrait de code 3.2 – Calcul de la perplexité avec le modèle `gpt-fr-cased-base`.

Les résultats sont donnés par Figure 3.2 et Table 3.1. On y observe que tous les modèles attribuent une perplexité plus faible aux phrases en français qu'à celles qui simulent l'aphasie de Broca. On note également que cette différence est plus prononcée pour les modèles entraînés sur des textes en français.

Les différences données par ces derniers sont compatibles avec les résultats obtenus par (Ghumman, 2021) avec des transcriptions de patients aphasiques et un groupe de contrôle. Il est raisonnable d'en conclure que les phrases générées sont similaires à celles produites dans le cas d'une aphasie de Broca.

3.2.2 Score BLEU

Le problème avec l'utilisation de la perplexité pour mesurer la différence entre les phrases générées et le langage ordinaire, est qu'elle ne prend pas en compte la relation entre une phrase aphasique et la phrase correcte qui lui correspond. Elle traite les deux corpus comme étant indépendants. BLEU est une métrique qui prend en compte cette relation (voire Section **??**). En prenant la moyenne des scores BLEU de chaque couple de phrases dans le corpus parallèle, nous obtenons un seul nombre qui mesure la similarité entre les deux parties du corpus.

La fonction `bleu_score` de `torchmetrics` prend en entrée une liste de phrases qui représentent les traductions candidates (dans notre cas, les phrases aphasiques) et une liste de listes de phrases qui représentent les traductions de référence (dans notre cas, les phrases en français). Un score BLEU de 62.72% a été obtenu sur le corpus de validation.

3.3 Entraînement du modèle

Tous les tests présentés dans le reste de ce chapitre ont été effectués sur un ordinateur portable équipé d'un processeur `Intel Core i9-8950HK` et d'une carte graphique `NVIDIA GeForce GTX 1050 Ti`. Cette machine dispose de 32 Go de mémoire vive et de 4 Go de mémoire vidéo. Le système d'exploitation est `Ubuntu 20.04 LTS`, la version de Python utilisée est `3.10.6` et celle de CUDA est `11.6.124`.

L'entraînement du modèle a été effectué en suivant la procédure illustrée par la figure 3.5. Le modèle est d'abord entraîné avec un choix d'hyperparamètres plus ou moins arbitraire pour vérifier que l'entraînement se déroule correctement. Ensuite, une recherche d'hyperparamètres est effectuée pour trouver les meilleurs hyperparamètres. Cette recherche est effectuée uniquement si l'entraînement initial n'a pas donné des résultats satisfaisants.

3.3.1 Choix des hyperparamètres

Pour notre premier test, nous avons entraîné le modèle pour 8 époques (cela a pris 58 min 25 s). Les hyperparamètres ont été choisis comme suit :

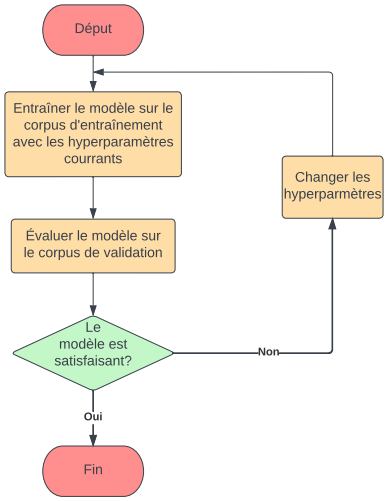


FIGURE 3.3 – Organigramme de la phase d'entraînement

— Dimension de plongement : 64
— Dimension de la couche cachée : 64
— Taux d'apprentissage : 3×10^{-4}
— Dropout : 0.1
— Nombre de couches de l'encodeur/décodeur : 3
— Nombre de têtes d'attention : 4
— Norme maximale des plongements : 1
— Taille de lot : 256
— Norme maximale du gradient : 1
— Coefficient de régularisation L_2 : 10^{-4}
— Nombre de processus pour le chargement des données : 4

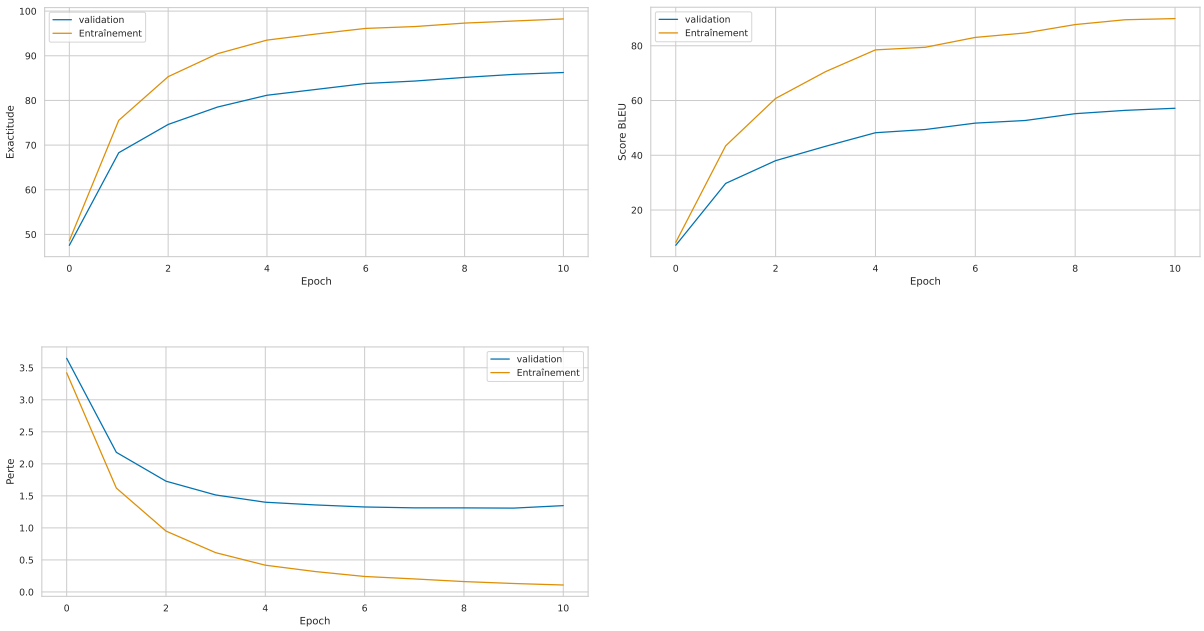
3.3.2 Entraînement initial

L'objectif de ce test est de vérifier que le modèle fonctionne correctement. Cela inclut la vérification du bon déroulement du chargeur de données, des passes forward et backward et l'absence de sur-apprentissage. La probabilité de ce dernier point n'est pas négligeable dans notre cas, car le modèle est grand en comparaison avec la complexité de la procédure de génération des données.

Pour chaque époque, nous avons calculé la moyenne de la fonction de perte, de l'exactitude et du score BLEU sur le corpus de validation et sur le corpus d'entraînement. Les résultats de ce test sont présentés dans la Figure 3.4. Ces résultats montrent que l'entraînement ne présente pas d'anomalies. Les 11 premières époques se déroulent sans erreurs et les métriques évoluent comme prévu. Cependant, la Figure 3.4c montre des signes clairs

(a) Exactitude

(b) BLEU



(c) Perte

FIGURE 3.4 – Évolution des métriques au cours de l'entraînement.

de sur-apprentissage. L'écart entre les courbes d'entraînement et de validation est très important.

Comme évoqué précédemment, ce résultat n'est pas surprenant. En effet, le modèle peut apprendre les règles de substitution utilisées pour générer les erreurs plutôt que la correspondance entre la phrase d'origine et la phrase erronée. Il est donc nécessaire de trouver une façon de forcer le modèle à apprendre la correspondance entre les phrases.

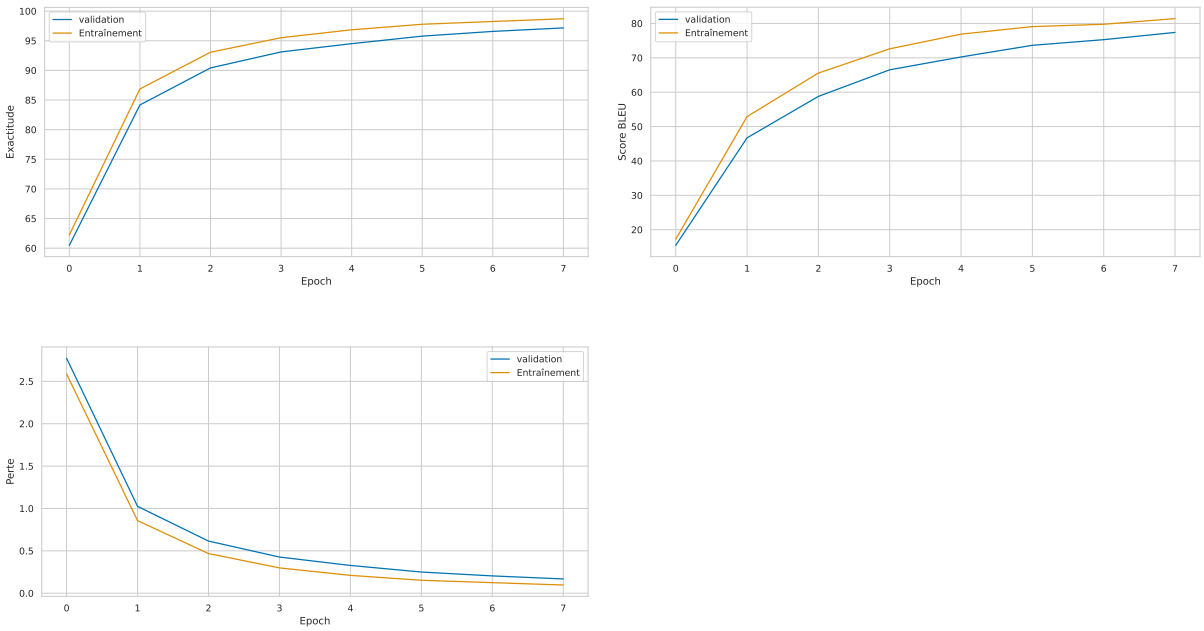
3.3.3 Entrainement avec les erreurs masquées

Pour remédier au problème de sur-apprentissage rencontré lors de l'entraînement initial, nous avons bruité les données d'entraînement. Pour empêcher le modèle d'apprendre les règles de substitution, nous avons masqué les erreurs en les remplaçant par le token `[UNK]`. Cette technique est inspirée du modélisation masquée du langage (MLM, de l'anglais : masked language modeling) de bidirectional encoder representations from transformers (BERT) (DEVLIN et al., 2019) et de l'auto-encodage de bidirectional auto-regressive transformer (BART) (LEWIS et al., 2019).

Les résultats sont présentés dans la Figure 3.5. Au bout de 8 époques, la perte sur le corpus d'entraînement est de 0.09. L'exactitude est de 98.71% et le score BLEU est de 81.41%. Les résultats sur le corpus de validation sont comparables. Les mêmes métriques valent respectivement 0.16, 97.16% et 77.38%.

(a) Exactitude

(b) BLEU



(c) Perte

FIGURE 3.5 – Évolution des métriques au cours de l'entraînement.

Les courbes de la Figure 3.5 montrent une forte correspondance entre les courbes d'entraînement et de validation. Cela suggère que le modèle ne sur-apprenne plus les erreurs. On note également que les pentes des trois courbes ne sont pas négligeables. Il est donc probable que le modèle puisse encore s'améliorer en augmentant le nombre d'époques.

3.4 Réglage des hyper-paramètres

Les résultats présentés dans Section 3.3.3 sont satisfaisants. Il n'est donc pas strictement nécessaire d'effectuer un réglage des hyperparamètres. Cependant, il est possible en le faisant d'obtenir des résultats comparables en moins d'époques ou avec un modèle plus petit. Pour cette raison, nous avons décidé de l'entamer.

3.4.1 Configuration

Nous avons fait le choix de restreindre la portée de ce réglage à deux hyperparamètres : le taux d'apprentissage (η ou `lr`) et le (`dropout`). Nous avons opté pour une recherche bayésienne avec des lois log–uniformes sur les intervalles $[10^{-5}, 10^{-1}]$ et $[0.1, 0.5]$ respectivement. Les autres hyperparamètres ont été fixés à leurs valeurs données dans Section 3.3.1.

Pour chaque combinaison de `lr` et `dropout`, nous avons entraîné le modèle pendant 2 époques. L'objectif de la recherche est de maximiser le score BLEU sur le corpus de validation. Nous avons limité le nombre d'essais à 20.

3.4.2 Résultats

Les 20 essais ont été effectués en 2 h 17 min 13 s 11 ms. Les résultats sont présentés dans Figure 3.6. Il s'agit d'une visualisation en coordonnées parallèles des résultats. L'axe

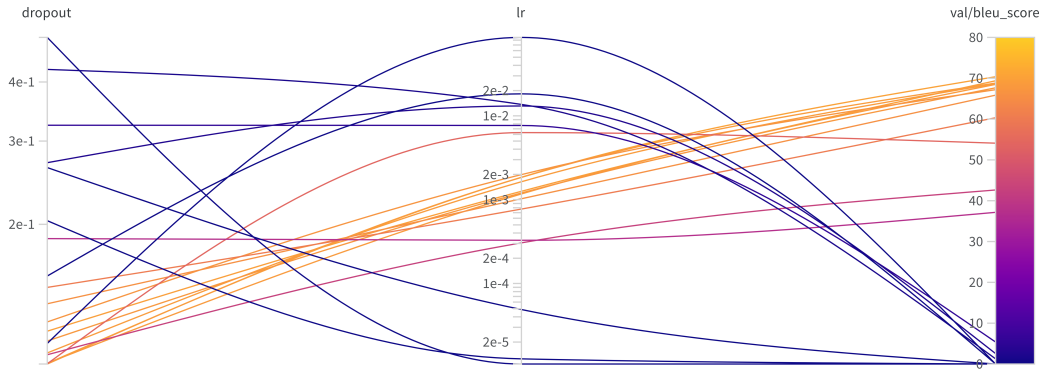


FIGURE 3.6 – Résultats de la recherche bayésienne des hyperparamètres.

de gauche représente la valeur de `dropout` sur une échelle linéaire, celui du milieu la valeur de `lr` sur une échelle logarithmique et celui de droite le score BLEU sur une échelle de 0 à 100. Un essai est représenté par une courbe qui relie les 3 points correspondants sur les 3 axes. La couleur de la courbe indique le score BLEU associé. Elle est interpolée entre le violet (0%) et l'orange (100%). Les mêmes informations sont présentes dans Table 3.2 sous forme numérique. La Figure 3.7 représente l'évolution temporelle du score BLEU de validation.

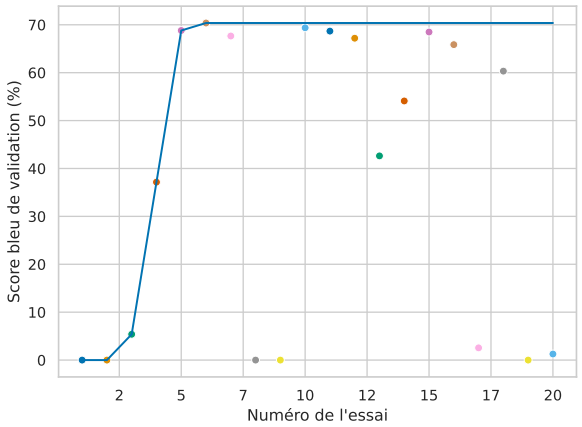


FIGURE 3.7 – Évolution du score BLEU au cours du réglage des hyperparamètres.

dropout	lr	bleu_score(%)
0.263684	0.000049	0.000000
0.155497	0.018280	0.000000
0.324059	0.007670	5.386821
0.186584	0.000328	37.148937
0.101305	0.001844	68.802765
0.101600	0.001849	70.371147
0.113328	0.001201	67.655579
0.111991	0.086200	0.000000
0.496975	0.000011	0.000000
0.118870	0.001259	69.372551
0.106921	0.001636	68.679092
0.124348	0.001989	67.192963
0.105984	0.000305	42.627316
0.101419	0.006308	54.094482
0.101742	0.001170	68.488480
0.135847	0.001025	65.854347
0.270016	0.013106	2.553007
0.147102	0.000772	60.324165
0.203584	0.000013	0.000000
0.425286	0.013681	1.264920

TABLE 3.2 – Résultats de la recherche bayésienne des hyperparamètres.

On observe sur la Figure 3.6 un regroupement des lignes oranges (les meilleurs essais) dans la région qui correspond à `dropout` $\in [0.1, 0.15]$ et $\eta \approx 10^{-3}$. Cela est aussi apparent sur le tableau. Une analyse statistique sur les essais effectués nous permet d'estimer

	Importance	Corrélation
`dropout`	63.9%	−0.899
`lr`	36.1%	−0.647

TABLE 3.3 – Importance et corrélation des hyperparamètres.

l'importance des hyperparamètres [2] ainsi que leurs corrélations avec le score BLEU. Ils sont tous deux négativement corrélés avec ce dernier (BLEU augmente quand l'un diminue). Le `dropout` est le plus important entre les deux (voir Table 3.3 et Figure 3.8). La meilleure valeur du score BLEU est obtenue avec `dropout` ≈ 0.101599 et $\eta \approx 0.0018488801$. Pour cette valeur, le score BLEU sur le corpus de validation est de 70.37% après 2 époques.

2. Un foret aléatoire est entraîné pour prédire le score BLEU en fonction des hyperparamètres. L'importance d'un hyperparamètre est mesurée par rapport à ce foret (« Parameter Importance | Weights & Biases Documentation », s. d.).

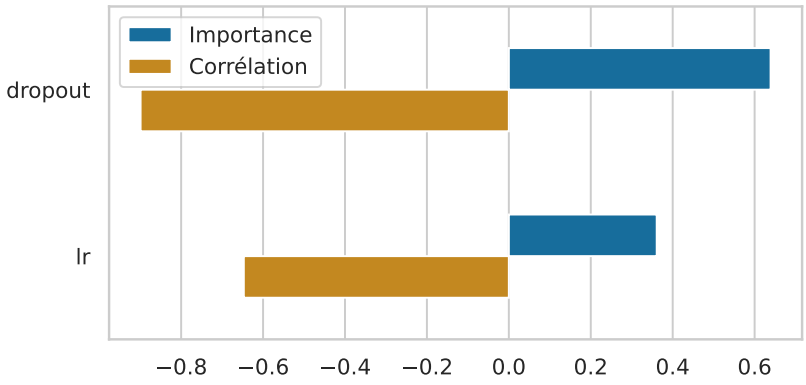


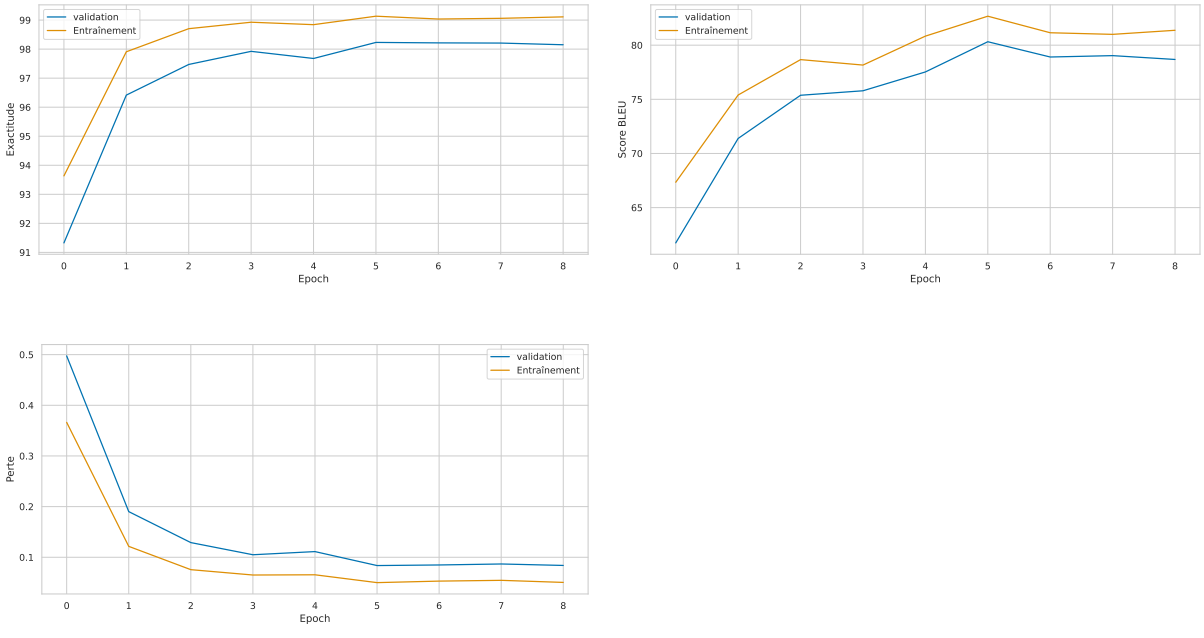
FIGURE 3.8 – Importance des hyperparamètres et corrélation avec le score BLEU.

3.5 Entraînement avec les hyperparamètres optimaux

Après le réglage des hyperparamètres, nous avons entraîné le modèle en spécifiant un nombre maximal d'époques de 20. L'entraînement s'est arrêté après 9 époques à cause du rappel de fonction `EarlyStopping`, car le score BLEU sur le corpus de validation n'a pas augmenté pendant 3 époques consécutives. L'exécution a duré 1 h 26 min 27 s.

(a) Exactitude

(b) BLEU



(c) Perte

FIGURE 3.9 – Évolution des métriques avec les hyperparamètres optimaux.

Les courbes d'apprentissage sont présentées sur la Figure 3.9. On remarque que les métriques d'entraînement et de validation sont très fortement corrélées. Il est donc im-

probable que le modèle sur-apprenne sur le corpus d'entraînement. La meilleure valeur du score BLEU a été obtenue après 6 époques et elle vaut 80.32%.

Sur le corpus de test, le score BLEU est de 79.61%, l'exactitude est de 98.15% et la perte de 8.39×10^{-2}. Cela indique que le modèle généralise bien et qu'il n'a pas surappris sur les corpus d'entraînement et de validation. Comparé à la valeur de base présentée dans la Section 3.2, le score BLEU a augmenté de 16.89%. Une amélioration de 2.22% a été obtenue par rapport au modèle entraîné avec les hyperparamètres par défaut.

3.6 Conclusion

Dans ce chapitre, nous avons présenté les résultats que nous avons obtenus à l'issue de notre travail. Nous avons obtenu une liste de 1104 erreurs pour 217 mots. Des statistiques sur la nature des erreurs et les mots concernés ont été présentées.

Ensuite, nous avons décrit le corpus que nous avons construit à partir de ces erreurs. Il compte 282 k couple de phrases avec une grande différence de perplexité entre les phrases correctes et les phrases erronées. Or, le score BLEU initial est plus élevé que celui atteint par la majorité des modèles de traduction.

L'entraînement d'un modèle de traduction sur ce corpus a montré des signes de sur-apprentissage Un deuxième tour d'entraînement masqué a donné des résultats satisfaisants (un score BLEU de 77.38%). Cependant, nous avons fait une recherche d'hyperparamètres qui a réussi à améliorer encore les résultats. Une deuxième phase d'entraînement a été effectuée avec les meilleurs hyperparamètres qui a donné des résultats encore meilleurs (un score BLEU de 79.61%).

Bibliographie

About Python. (2022). https://pythoninstitute.org/about-python

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning* [Google-Books-ID : qWPwnQEACAAJ]. Springer.

Devlin, J., Chang, M.-W., Lee, K., & Touta11a, K. (2019). BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding [arXiv :1810.04805 [cs]], (arXiv :1810.04805). http://arxiv.org/abs/1810.04805

Falcon, W., & The PyTorch Lightning team. (2019). *PyTorch Lightning* (Version 1.4). https://doi.org/10.5281/zenodo.3828935

Ghumman, N. S. (2021). *Training and Probing Language Models for Discerning between Speech of People with Aphasia and Healthy Controls* (thèse de doct.). University of Georgia.

Goldhahn, D., Eckart, T., & Quasthoff, U. (2012). Building Large Monolingual Dictionaries at the Leipzig Corpora Collection : From 100 to 200 Languages.

Islam, K., Zaheer, M. Z., & Mahmood, A. (2022). Face Pyramid Vision Transformer. *arXiv preprint arXiv :2210.11974*.

Kingma, D. P., & Ba, J. (2017). Adam : A Method for Stochastic Optimization [arXiv :1412.6980 [cs]], (arXiv :1412.6980). http://arxiv.org/abs/1412.6980

Le tutoriel Python. (2023). https://docs.python.org/3/tutorial/index.html

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoya11, V., & Zettlemoyer, L. (2019). BART : Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension [arXiv :1910.13461 [cs, stat]], (arXiv :1910.13461). http://arxiv.org/abs/1910.13461

Lu, A., Huang, H., Hu, Y., Zbijewski, W., Unberath, M., Siewerdsen, J. H., Weiss, C. R., & Sisniega, A. (2023). Deformable motion compensation for intraprocedural vascular cone-beam CT with sequential projection domain targeting and vessel-enhancing autofocus. *Medical Imaging 2023 : Image-Guided Procedures, Robotic Interventions, and Modeling*, *12466*, 169-174.

MacWhinney, B. (2007). The talkbank project. *Creating and Digitizing Language Corpora : Volume 1 : Synchronic Databases*, 163-180.

MacWhinney, B., Fromm, D., Forbes, M., & Holland, A. (2011). AphasiaBank : Methods for studying discourse. *Aphasiology*, *25*(11), 1286-1307. https://doi.org/10.1080/02687038.2011.589893

Parameter Importance | Weights & Biases Documentation. (s. d.). https://docs.wandb.ai/guides/app/features/panels/parameter-importance

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Te1i, A., Chilamkurthy, S., Steiner, B., Fang, L.,

... Garnett, R. (2019). PyTorch : An Imperative Style, High-Performance Deep Learning Library. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2022). Robust Speech Recognition via Large-Scale Weak Supervision [arXiv :2212.04356 [cs, eess]], (arXiv :2212.04356). http://arxiv.org/abs/2212.04356

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.

Smaïli, K., Langlois, D., & Pribil, P. (2022). Language rehabilitation of people with BROCA aphasia using deep neural machine translation. *Fifth International Conference Computational Linguistics in Bulgaria*, 162.

Stack Overflow Developer Survey. (2022). https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022

van Rijsbergen, C. J. (2002, janvier 3). *Information Retrieval*. Butterworth.

Vasilev, I., Slater, D., Spacagna, G., Roelants, P., & Zocca, V. (2019). *Python Deep Learning : Exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow, 2nd Edition* [Google-Books-ID : ESKEDwAAQBAJ]. Packt Publishing Ltd.

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*.

Annexe A

Dépendances et bibliothèques

```
lightning==2.0.2
torch==2.0.0
pytorch_memlab==0.2.4
PyYAML==6.0
tokenizers==0.13.3
torchdata==0.6.0
torchmetrics==0.11.4
torchtext==0.15.1
torchview==0.2.6
tqdm==4.64.1
beautifulsoup4==4.11.1
openai==0.27.2
pandas==1.5.3
PyHyphen==4.0.3
python-dotenv==1.0.0
Requests==2.30.0
scikit_learn==1.2.0
tokenizers==0.13.3
tqdm==4.64.1
evaluate==0.4.0
```