Traitement Automatique du Langage

Notes de Cours

Master M2 — 2025–2026

Yaë Ulrich Gaba

« Les limites de mon langage sont les limites de mon monde. »
— Ludwig Wittgenstein

25 mars 2026

Table des matières

Préface

Objectifs de ce cours

Le *Traitement Automatique du Langage* (TAL), ou *Natural Language Processing* (NLP), constitue l'un des domaines les plus dynamiques de l'intelligence artificielle contemporaine. Ce cours de niveau Master/Doctorat vise à fournir une formation complète, allant des fondements mathématiques classiques jusqu'aux modèles de langue de dernière génération.

L'objectif principal est triple :

1. **Comprendre les fondements théoriques** — modélisation probabiliste du langage, théorie de l'information, algèbre linéaire pour les représentations vectorielles.
2. **Maîtriser les architectures modernes** — réseaux récurrents, mécanismes d'attention, Transformers, modèles pré-entraînés (BERT, GPT, T5).
3. **Développer un esprit critique** — évaluation rigoureuse, considérations éthiques, biais algorithmiques, sécurité des systèmes génératifs.

Public visé

Ce cours s'adresse aux étudiants de Master 2 et de Doctorat en informatique, en science des données ou en linguistique computationnelle. Les prérequis suivants sont supposés :

- **Mathématiques** : algèbre linéaire (espaces vectoriels, décompositions matricielles), calcul différentiel multivarié, probabilités et statistiques.
- **Informatique** : programmation en Python, notions d'algorithmique, familiarité avec les bibliothèques scientifiques (NumPy, PyTorch ou TensorFlow).
- **Apprentissage automatique** : régression, classification, descente de gradient, réseaux de neurones de base.

Organisation du cours

Le cours est organisé en onze chapitres, suivant une progression logique depuis les représentations classiques du texte jusqu'aux systèmes génératifs les plus récents :

Chapitre 1 — Introduction au TAL. Panorama historique, tâches fondamentales, défis spécifiques au langage naturel.

Chapitre 2 — Représentations classiques du texte. Modèles sac-de-mots, TF-IDF, n-grammes, représentations creuses.

Chapitre 3 — Plongements de mots. Word2Vec (CBOW, Skip-gram), GloVe, FastText ; propriétés géométriques des espaces de plongements.

Chapitre 4 — Modèles de séquences. Réseaux récurrents (RNN), LSTM, GRU ; problèmes de gradient évanescent et explosif.

Chapitre 5 — Attention et Transformers. Mécanisme d'attention, auto-attention, architecture Transformer complète.

Chapitre 6 — Modèles pré-entraînés. BERT, GPT, T5 ; paradigme pré-entraînement/adaptation.

Chapitre 7 — Ajustement fin et instruction tuning. Stratégies d'adaptation, LoRA, RLHF, alignement.

Chapitre 8 — Génération de texte et décodage. Décodage glouton, recherche en faisceau, échantillonnage, top-k, top-p.

Chapitre 9 — Génération augmentée par récupération (RAG). Recherche de passages, intégration de connaissances externes, pipelines RAG.

Chapitre 10 — Évaluation des grands modèles de langue. Métriques automatiques, évaluation humaine, benchmarks, robustesse.

Chapitre 11 — Éthique, biais et sécurité. Biais dans les données et les modèles, toxicité, vie privée, régulation.

Conventions typographiques

- Les **définitions** formelles sont encadrées dans des environnements `definition`.
- Les **théorèmes**, **propositions** et **lemmes** sont numérotés et, lorsque pertinent, accompagnés d'une preuve.
- Les blocs **intuition** fournissent une compréhension qualitative avant la formalisation.
- Les blocs **attention** signalent les erreurs fréquentes.
- Les blocs **bonne pratique** résument les recommandations concrètes.
- Le code Python utilise principalement les bibliothèques `transformers` (Hugging Face), `torch` et `scikit-learn`.

Notations mathématiques

Nous adoptons les notations suivantes tout au long du cours :

Notation	Signification
$\mathbb{R}^d$	Espace euclidien de dimension d
$\mathbb{N}$	Ensemble des entiers naturels
$\mathbf{x}, \mathbf{W}$	Vecteurs (minuscule gras), matrices (majuscule gras)
$\langle \mathbf{u}, \mathbf{v} \rangle$	Produit scalaire de $\mathbf{u}$ et $\mathbf{v}$
$\|\mathbf{x}\|$	Norme euclidienne de $\mathbf{x}$
$\|S\|$	Cardinalité d'un ensemble S
$\mathbb{E}[X]$	Espérance de la variable aléatoire X
$P(A \mid B)$	Probabilité conditionnelle de A sachant B
$\mathrm{KL}(p\|q)$	Divergence de Kullback-Leibler de p vers q
$\sigma(\cdot)$	Fonction sigmoïde logistique
$\mathrm{softmax}(\cdot)$	Fonction softmax
$\arg\max_x f(x)$	Argument maximisant f
$\arg\min_x f(x)$	Argument minimisant f
$\mathcal{V}$	Vocabulaire (ensemble fini de tokens)
$V = \|\mathcal{V}\|$	Taille du vocabulaire
$\mathbf{E} \in \mathbb{R}^{V \times d}$	Matrice de plongements
T	Longueur d'une séquence

Logiciels et environnement

Les travaux pratiques et exemples de code s'appuient sur l'écosystème suivant :

- **Python** ≥ 3.10
- **PyTorch** ≥ 2.0
- **Hugging Face Transformers** ≥ 4.35
- **Hugging Face Datasets**
- **scikit-learn** pour les méthodes classiques
- **NLTK** et **spaCy** pour le prétraitement linguistique
- **Matplotlib** / **seaborn** pour la visualisation

Environnement reproductible

Nous recommandons d'utiliser un environnement virtuel (`venv` ou `conda`) et de fixer les graines aléatoires (`torch.manual_seed`, `random.seed`, `numpy.random.seed`) pour garantir la reproductibilité des expériences.

Perspective historique

Le TAL a connu plusieurs révolutions paradigmatiques. Dans les années 1950, les premières approches étaient fondées sur des règles linguistiques codées manuellement. Les années 1990 ont vu l'essor des méthodes statistiques (modèles de Markov cachés, modèles n-grammes). Les années 2010 ont été dominées par l'apprentissage profond (plongements de mots, réseaux récurrents). Depuis 2017, l'architecture Transformer a inauguré l'ère des grands modèles de langue (LLM), transformant radicalement le paysage du domaine.

Pourquoi le langage est-il difficile ?

Le langage naturel est intrinsèquement ambigu (polysmie, anaphores, ellipses), contextuel (le sens dépend du contexte pragmatique) et compositional (le sens d'une phrase dépend de la structure syntaxique et du sens de ses constituants). Ces propriétés rendent le TAL fondamentalement différent du traitement de données structurées.

Comment utiliser ce cours

Chaque chapitre est conçu pour être auto-suffisant, tout en s'appuyant sur les concepts des chapitres précédents. Nous recommandons une lecture séquentielle pour une première approche, puis un usage ciblé comme référence.

Les exercices proposés à la fin de chaque chapitre sont classés en trois niveaux de difficulté :

- ⋆ Exercices de compréhension et de vérification.
- ⋆⋆ Exercices d'approfondissement nécessitant une réflexion mathématique ou une implémentation.
- ⋆⋆⋆ Exercices de recherche ouverts, pouvant servir de point de départ pour un projet ou un mémoire.

Remerciements

Ce cours a bénéficié des contributions de nombreux collègues, chercheurs et étudiants. Nous remercions en particulier les communautés open source de Hugging Face, PyTorch et spaCy, dont les outils rendent l'enseignement du TAL à la fois rigoureux et accessible.

Nous remercions également les étudiants des promotions précédentes dont les questions et retours ont grandement amélioré la qualité de ce matériel pédagogique.

[Author Name], mars 2026

Chapitre 1

Introduction au TAL

En 1950, Alan Turing pose la question fondatrice de l'intelligence artificielle : une machine peut-elle « penser » ? Et il propose un critère opérationnel : si un humain, en conversant avec une machine par texte interposé, ne parvient pas à la distinguer d'un autre humain, alors la machine « pense » au sens fonctionnel du terme. Ce « test de Turing » est fondamentalement un défi de *traitement du langage naturel* : pour le réussir, une machine doit comprendre la syntaxe, la sémantique, le contexte, l'ironie, le sous-entendu. Soixante-quinze ans plus tard, les modèles de langage de grande taille (GPT, Claude, LLaMA) semblent s'en approcher — mais les défis fondamentaux restent intacts.

Pourquoi étudier le TAL ?

Le langage est le principal vecteur de communication humaine. Enseigner aux machines à comprendre, produire et raisonner sur le langage naturel est l'un des défis fondamentaux de l'intelligence artificielle. Les applications vont de la traduction automatique à l'extraction d'information, en passant par les agents conversationnels et la génération de texte.

1.1 Qu'est-ce que le TAL ?

Définition 1.1 (Traitement Automatique du Langage)**.** Le *Traitement Automatique du Langage* (TAL) est le sous-domaine de l'intelligence artificielle qui étudie les interactions entre les ordinateurs et le langage humain. Il englobe l'analyse, la compréhension et la génération de texte et de parole en langues naturelles.

Le TAL se distingue de la *linguistique computationnelle* par son orientation vers les applications, bien que les deux domaines partagent un socle théorique commun. Formellement, on peut considérer le langage comme un système génératif défini sur un vocabulaire $\mathcal{V}$:

$$\mathcal{L} \subseteq \mathcal{V}^* = \bigcup_{n=0}^{\infty} \mathcal{V}^n$$

où $\mathcal{V}^*$ désigne l'ensemble de toutes les séquences finies sur $\mathcal{V}$.

1.2 Niveaux d'analyse linguistique

Le traitement du langage naturel opère à plusieurs niveaux d'analyse :

1. **Phonétique et phonologie** : étude des sons du langage (principalement pour le traitement de la parole).

2. **Morphologie** : structure interne des mots (préfixes, suffixes, flexions). Par exemple, « anti-constitutionnel »se décompose en `anti-` + `constitution` + `-nel`.

3. **Syntaxe** : structure des phrases, relations grammaticales.

4. **Sémantique** : sens des mots et des phrases.

5. **Pragmatique** : sens en contexte, intentions du locuteur.

6. **Discours** : cohérence et structure au-delà de la phrase.

Remarque 1.2. En TAL moderne, les modèles profonds apprennent implicitement ces niveaux d'analyse à partir des données, sans segmentation explicite. Cependant, comprendre ces niveaux reste essentiel pour diagnostiquer les erreurs et concevoir des évaluations pertinentes.

1.3 Tâches fondamentales du TAL

1.3.1 Classification de texte

La classification de texte consiste à assigner une étiquette $y \in \mathcal{Y}$ à un document $\mathbf{x} = (x_1, x_2, \ldots, x_T)$. Le modèle apprend une fonction $f : \mathcal{V}^* \to \mathcal{Y}$ à partir d'un ensemble d'entraînement $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$.

Applications : analyse de sentiments, détection de spam, classification thématique.

1.3.2 Étiquetage de séquences

L'étiquetage de séquences assigne une étiquette $y_t \in \mathcal{Y}$ à chaque token x_t d'une séquence :

$$f : \mathcal{V}^T \to \mathcal{Y}^T$$

Applications : reconnaissance d'entités nommées (NER), étiquetage morpho-syntaxique (POS tagging).

Exemple 1.3. Considérons la phrase « Marie habite à Paris »avec l'étiquetage NER suivant :

Marie	habite	à	Paris
`B-PER`	`O`	`O`	`B-LOC`

1.3.3 Traduction automatique

La traduction automatique est une tâche de transduction séquence-à-séquence :

$$f : \mathcal{V}_{\text{source}}^* \to \mathcal{V}_{\text{cible}}^*$$

C'est historiquement l'une des tâches motrices du TAL.

1.3.4 Question-réponse

Soit un contexte C et une question Q, le système doit extraire ou générer une réponse A :

$$f : (C, Q) \to A$$

1.3.5 Résumé automatique

Le résumé automatique produit une version condensée S d'un document D, préservant l'information essentielle : $f : D \to S$ avec $|S| \ll |D|$.

1.4 Fondements probabilistes

La modélisation probabiliste du langage est au cœur du TAL. Un *modèle de langue* définit une distribution de probabilité sur les séquences de tokens.

Définition 1.4 (Modèle de langue)**.** Un *modèle de langue* est une distribution de probabilité P sur $\mathcal{V}^*$ telle que :

$$P(\mathbf{x}) = P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

où la factorisation découle de la règle de la chaîne des probabilités.

Théorème 1.5 (Règle de la chaîne)**.** *Pour toute distribution jointe $P(x_1, \ldots, x_T)$:*

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$$

où $x_{<t} = (x_1, \ldots, x_{t-1})$ désigne le contexte *au pas t.*

Démonstration. Par applications successives de la définition de la probabilité conditionnelle $P(A \mid B) = P(A, B)/P(B)$:

$$\begin{aligned} P(x_1, x_2, \ldots, x_T) &= P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_1, x_2) \cdots \\ &= \prod_{t=1}^{T} P(x_t \mid x_{<t}). \end{aligned}$$

□

1.5 Théorie de l'information et langage

Définition 1.6 (Entropie)**.** L'*entropie* d'une variable aléatoire discrète X à valeurs dans $\mathcal{X}$ est :

$$H(X) = -\sum_{x \in \mathcal{X}} P(x) \log_2 P(x)$$

Elle mesure l'incertitude moyenne associée à X.

Définition 1.7 (Entropie croisée)**.** L'*entropie croisée* entre la distribution réelle p et un modèle q est :

$$H(p, q) = -\sum_{x} p(x) \log q(x) = H(p) + \mathrm{KL}(p \| q)$$

Proposition 1.8 (Borne inférieure)**.** Pour tout modèle q, on a $H(p, q) \geq H(p)$, avec égalité si et seulement si $q = p$.

La *perplexité* d'un modèle de langue q est définie par :

$$\mathrm{PPL}(q) = 2^{H(p,q)}$$

Métriques informationnelles clés

$$H(X) = -\sum_{x} P(x) \log P(x) \qquad \text{(entropie)} \tag{1.1}$$

$$\mathrm{KL}(p\|q) = \sum_{x} p(x) \log \frac{p(x)}{q(x)} \qquad \text{(divergence KL)} \tag{1.2}$$

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log q(x_t \mid x_{<t})\right) \qquad \text{(perplexité)} \tag{1.3}$$

1.6 Pipeline classique de TAL

Un pipeline de TAL classique comprend les étapes suivantes :

1. **Collecte de données** : corpus textuels, web scraping, APIs.
2. **Prétraitement** :
   - Tokenisation (découpage en unités lexicales)
   - Normalisation (mise en minuscules, suppression d'accents)
   - Suppression des mots vides (*stopwords*)
   - Racinisation (*stemming*) ou lemmatisation
3. **Représentation** : conversion du texte en vecteurs numériques.
4. **Modélisation** : entraînement d'un modèle.
5. **Évaluation** : mesure de la performance sur un jeu de test.
6. **Déploiement** : mise en production du modèle.

Prétraitement et modèles modernes

Avec les modèles pré-entraînés (BERT, GPT), le prétraitement est généralement minimal : le tokeniseur du modèle (BPE, WordPiece) remplace les étapes classiques de stemming et de suppression de stopwords. Appliquer un prétraitement agressif peut même dégrader les performances.

1.7 Tokenisation

Définition 1.9 (Tokenisation)**.** La *tokenisation* est le processus de découpage d'un texte en unités élémentaires appelées *tokens*. Un token peut être un mot, un sous-mot, un caractère ou un byte.

1.7.1 Tokenisation par sous-mots

Les algorithmes modernes de tokenisation par sous-mots (BPE, WordPiece, Unigram) offrent un compromis entre la tokenisation par mots (vocabulaire très grand) et par caractères (séquences très longues).

Byte Pair Encoding (BPE)

1. Initialiser le vocabulaire avec tous les caractères individuels.
2. Répéter k fois :
   (a) Compter toutes les paires de symboles adjacents dans le corpus.
   (b) Fusionner la paire la plus fréquente en un nouveau symbole.
   (c) Ajouter ce symbole au vocabulaire.
3. Retourner le vocabulaire final de taille $|\mathcal{V}_0| + k$.

Tokenisation avec Hugging Face

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
text = "Natural language processing is fascinating."
tokens = tokenizer.tokenize(text)
print(tokens)
# ['natural', 'language', 'processing', 'is', 'fascinating', '.']

ids = tokenizer.encode(text, return_tensors="pt")
print(ids)
# tensor([[ 101, 3019, 2653, 6364, 2003, 15281, 1012,  102]])
```

1.8 Bref historique du TAL

Période	Paradigme	Exemples
1950–1970	Règles symboliques	ELIZA, SHRDLU
1970–1990	Grammaires formelles	ATN, LFG, HPSG
1990–2010	Méthodes statistiques	HMM, CRF, n-grammes
2010–2017	Apprentissage profond	Word2Vec, LSTM, Seq2Seq
2017–	Transformers & LLM	BERT, GPT, T5, LLaMA

Remarque 1.10. Le passage du paradigme statistique au paradigme neuronal ne représente pas une rupture totale : les concepts d'entropie croisée, de modèle de langue et de régularisation restent au cœur des approches modernes.

1.9 Défis ouverts

Malgré les progrès spectaculaires, de nombreux défis restent ouverts :

- **Compréhension réelle vs. corrélations statistiques** : les LLM capturent-ils le *sens* ou simplement des régularités de surface ?
- **Langues peu dottées** : la majorité des ressources sont disponibles en anglais ; environ 7 000 langues sont parlées dans le monde.
- **Robustesse** : sensibilité aux perturbations adversariales, aux néologismes, aux domaines hors distribution.
- **Efficacité** : les LLM requièrent des ressources computationnelles considérables.
- **Éthique** : biais, toxicité, confidentialité, désinformation.

1.10 Exercices

Exercice 1.1 ($\star$)**.** Calculez l'entropie d'un modèle de langue uniforme sur un vocabulaire de taille $V = 50\,000$. Quelle est la perplexité correspondante ?

Exercice 1.2 ($\star$)**.** Soit la phrase « Le chat mange la souris ». Appliquez la règle de la chaîne pour exprimer $P(\text{Le, chat, mange, la, souris})$ comme un produit de probabilités conditionnelles.

Exercice 1.3 ($\star\star$)**.** Implémentez un tokeniseur BPE simple en Python. Partez d'un petit corpus de 10 phrases et montrez l'évolution du vocabulaire après 20 fusions.

Exercice 1.4 ($\star\star$)**.** Montrez que la divergence de Kullback-Leibler $\mathrm{KL}(p\|q)$ est toujours positive ou nulle (inégalité de Gibbs). *Indication :* utilisez l'inégalité $\log x \leq x - 1$.

Exercice 1.5 ($\star\star\star$)**.** Comparez la perplexité d'un modèle n-gramme (pour $n = 1, 2, 3$) et d'un modèle neuronal (GPT-2) sur un corpus de votre choix. Analysez les résultats en fonction de la taille du corpus d'entraînement.

Chapitre 2

Représentations Textuelles Classiques

Un algorithme ne lit pas. Il calcule. Pour qu'un classifieur, un moteur de recherche ou un système de traduction puisse travailler sur du texte, il faut d'abord transformer les mots en nombres — ou plus précisément, en *vecteurs*. Comment effectuer cette transformation ? C'est l'une des questions fondatrices du TAL, et les réponses ont évolué considérablement : des représentations « sac de mots » des années 1990 aux plongements denses de Word2Vec, en passant par la pondération TF-IDF.

Du texte aux vecteurs

Les algorithmes d'apprentissage automatique opèrent sur des vecteurs numériques, pas sur du texte brut. Ce chapitre présente les méthodes classiques pour transformer un document textuel en une représentation vectorielle exploitable par un classifieur ou un moteur de recherche.

2.1 Représentation sac-de-mots

Définition 2.1 (Sac-de-mots (Bag of Words))**.** Le modèle *sac-de-mots* représente un document d comme un vecteur $\mathbf{x}_d \in \mathbb{N}^V$ où la j-ème composante est le nombre d'occurrences du terme t_j dans d :

$$x_{d,j} = \text{count}(t_j, d)$$

Ce modèle ignore complètement l'ordre des mots. La phrase « le chat mange la souris »et « la souris mange le chat »ont la même représentation.

Remarque 2.2. Malgré sa simplicité, le modèle sac-de-mots reste utile comme ligne de base (*baseline*) et dans des applications où l'ordre des mots est moins important (classification thématique, filtrage de documents).

2.2 Pondération TF-IDF

La fréquence brute ne reflète pas l'importance discriminative d'un terme. Les mots très fréquents (“le”, “de”, “et”) dominent la représentation sans apporter d'information sémantique.

Définition 2.3 (TF – Term Frequency)**.** La *fréquence du terme* t dans le document d est :

$$\mathrm{tf}(t, d) = \frac{\mathrm{count}(t, d)}{\sum_{t' \in d} \mathrm{count}(t', d)}$$

Définition 2.4 (IDF – Inverse Document Frequency)**.** La *fréquence inverse de document* du terme t dans un corpus $\mathcal{D}$ de N documents est :

$$\mathrm{idf}(t, \mathcal{D}) = \log \frac{N}{|\{d \in \mathcal{D} : t \in d\}|}$$

Définition 2.5 (TF-IDF)**.** La pondération *TF-IDF* combine les deux mesures :

$$\text{tf-idf}(t, d, \mathcal{D}) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, \mathcal{D})$$

Théorème 2.6 (Propriétés de TF-IDF)**.** *Soit* $\mathbf{v}_d \in \mathbb{R}^V$ *le vecteur TF-IDF du document* d*. Alors :*

1. *tf-idf*$(t, d) = 0$ *si* $t \notin d$ *;*
2. *tf-idf*(t, d) *est élevé si* t *est fréquent dans* d *mais rare dans le corpus ;*
3. *idf*$(t) = 0$ *si* t *apparaît dans tous les documents.*

Variantes de TF-IDF

$$\mathrm{tf}_{\log}(t, d) = 1 + \log(\mathrm{count}(t, d)) \qquad \text{(tf logarithmique)} \tag{2.1}$$

$$\mathrm{idf}_{\text{smooth}}(t) = \log \frac{N + 1}{\mathrm{df}(t) + 1} + 1 \qquad \text{(idf lissé)} \tag{2.2}$$

$$\text{tf-idf}_{\text{norm}} = \frac{\text{tf-idf}(t, d)}{\|\mathbf{v}_d\|} \qquad \text{(normalisé } L_2\text{)} \tag{2.3}$$

TF-IDF avec scikit-learn

```python
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "Le chat dort sur le tapis",
    "Le chien court dans le jardin",
    "Le chat et le chien jouent ensemble"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
print(f"Forme de la matrice : {X.shape}")
print(f"Vocabulaire : {vectorizer.get_feature_names_out()}")
print(f"Matrice TF-IDF (dense) :\n{X.toarray().round(2)}")
```

2.3 Modèles n-grammes

Définition 2.7 (n-gramme)**.** Un *n-gramme* est une sous-séquence contiguë de n tokens extraite d'un texte. Pour une séquence $(x_1, \ldots, x_T)$, les n-grammes sont :

$$\{(x_t, x_{t+1}, \ldots, x_{t+n-1}) : t = 1, \ldots, T - n + 1\}$$

2.3.1 Modèle de langue n-gramme

Un modèle de langue n-gramme approche la probabilité conditionnelle par une hypothèse de Markov d'ordre $n - 1$:

Définition 2.8 (Modèle de langue n-gramme)**.**

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1})$$

estimée par maximum de vraisemblance :

$$\hat{P}(x_t \mid x_{t-n+1}^{t-1}) = \frac{\text{count}(x_{t-n+1}, \ldots, x_t)}{\text{count}(x_{t-n+1}, \ldots, x_{t-1})}$$

Problème des zéros

Si un n-gramme n'apparaît jamais dans le corpus d'entraînement, sa probabilité estimée est nulle, ce qui rend la perplexité infinie. Les techniques de lissage (Laplace, Kneser-Ney) sont indispensables.

2.3.2 Lissage de Laplace (add-α)

Définition 2.9 (Lissage de Laplace)**.** Le lissage de Laplace ajoute un pseudo-comptage $\alpha > 0$:

$$\hat{P}_\alpha(x_t \mid x_{t-n+1}^{t-1}) = \frac{\text{count}(x_{t-n+1}^{t}) + \alpha}{\text{count}(x_{t-n+1}^{t-1}) + \alpha V}$$

où $V = |\mathcal{V}|$.

2.3.3 Lissage de Kneser-Ney

Définition 2.10 (Lissage de Kneser-Ney)**.** Le lissage de Kneser-Ney utilise un discount absolu δ et une distribution de continuation :

$$P_{\text{KN}}(w \mid h) = \frac{\max(\text{count}(h, w) - \delta, 0)}{\text{count}(h)} + \lambda(h) \cdot P_{\text{cont}}(w)$$

où $P_{\text{cont}}(w) \propto |\{h' : \text{count}(h', w) > 0\}|$ est la probabilité de continuation et $\lambda(h)$ est un facteur de normalisation.

2.4 Similarité entre documents

Définition 2.11 (Similarité cosinus)**.** La *similarité cosinus* entre deux vecteurs $\mathbf{u}, \mathbf{v} \in \mathbb{R}^V$ est :

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|}$$

Proposition 2.12 (Propriétés de la similarité cosinus)**.**

1. $\cos(\mathbf{u}, \mathbf{v}) \in [-1, 1]$
2. $\cos(\mathbf{u}, \mathbf{v}) = 1 \iff \mathbf{u} = \lambda \mathbf{v}$ pour $\lambda > 0$
3. Pour des vecteurs TF-IDF (composantes ≥ 0) : $\cos(\mathbf{u}, \mathbf{v}) \in [0, 1]$

Définition 2.13 (Distance de Jaccard)**.** Pour deux ensembles de termes A et B :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \qquad d_J(A, B) = 1 - J(A, B)$$

2.5 Matrice documents-termes et SVD

La matrice documents-termes $\mathbf{M} \in \mathbb{R}^{N \times V}$ est généralement très creuse ($> 99\%$ de zéros). La décomposition en valeurs singulières (SVD) permet de réduire la dimensionnalité.

Définition 2.14 (Analyse sémantique latente (LSA))**.** L'analyse sémantique latente (LSA) applique la SVD tronquée à la matrice TF-IDF :

$$\mathbf{M} \approx \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$$

où $k \ll \min(N, V)$. Les documents sont alors représentés dans $\mathbb{R}^k$ par les lignes de $\mathbf{U}_k \mathbf{\Sigma}_k$.

Théorème 2.15 (Optimalité de la SVD tronquée (Eckart-Young))**.** *La matrice de rang k qui minimise l'erreur de Frobenius $\left\|\mathbf{M} - \hat{\mathbf{M}}\right\|_F$ est donnée par la SVD tronquée $\hat{\mathbf{M}} = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$.*

LSA avec scikit-learn

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

vectorizer = TfidfVectorizer(max_features=10000)
X_tfidf = vectorizer.fit_transform(corpus)

svd = TruncatedSVD(n_components=100, random_state=42)
X_lsa = svd.fit_transform(X_tfidf)
print(f"Variance expliquee : {svd.explained_variance_ratio_.sum():.2%}")
print(f"Dimension reduite : {X_lsa.shape}")
```

2.6 Représentations par caractères n-grammes

Les n-grammes de caractères capturent la morphologie et sont robustes aux fautes d'orthographe :

Exemple 2.16. Le mot « bonjour »avec $n = 3$ produit les trigrammes : `#bo`, `bon`, `onj`, `njo`, `jou`, `our`, `ur#` (où `#` marque les frontières).

2.7 Modèle BM25

Définition 2.17 (BM25)**.** Le score BM25 du document d pour la requête $q = \{q_1, \ldots, q_m\}$ est :

$$\text{BM25}(d, q) = \sum_{i=1}^{m} \text{idf}(q_i) \cdot \frac{\text{tf}(q_i, d) \cdot (k_1 + 1)}{\text{tf}(q_i, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

où k_1 et b sont des hyperparamètres (typiquement $k_1 = 1.2$, $b = 0.75$) et avgdl est la longueur moyenne des documents.

BM25 en pratique

BM25 reste une méthode de référence pour la recherche d'information (*information retrieval*). Même les systèmes RAG modernes utilisent souvent BM25 comme première étape de récupération avant un reranking neuronal.

2.8 Limites des représentations creuses

Les méthodes présentées dans ce chapitre souffrent de plusieurs limitations :

1. **Haute dimensionnalité** : V peut atteindre 10^5 à 10^6.
2. **Parcimonie** : les vecteurs sont très creux, rendant les mesures de distance peu fiables.
3. **Absence de sémantique** : les mots synonymes ont des composantes orthogonales (“voiture” $\perp$ “automobile”).
4. **Indépendance contextuelle** : chaque mot a une représentation fixe indépendante du contexte.

Ces limitations motivent les plongements denses (chapitre 3), qui représentent les mots dans un espace de faible dimension où la proximité géométrique reflète la similarité sémantique.

2.9 Application : classification naïve de Bayes

Définition 2.18 (Classifieur naïf de Bayes multinomial)**.** Soit un document représenté par ses comptes de mots $\mathbf{x} = (x_1, \ldots, x_V)$. Le classifieur naïf de Bayes multinomial prédit :

$$\hat{y} = \arg\max_{c \in \mathcal{Y}} \left[\log P(c) + \sum_{j=1}^{V} x_j \log P(t_j \mid c) \right]$$

avec $P(t_j \mid c) = \frac{\text{count}(t_j, c) + \alpha}{\sum_{j'} \text{count}(t_{j'}, c) + \alpha V}$.

Classification naïve de Bayes

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups(subset='train', categories=['sci.space',
↪  'rec.sport.baseball'])
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer(max_features=10000)),
    ('clf', MultinomialNB(alpha=0.1))
])
pipeline.fit(data.data, data.target)
print(f"Accuracy (train) : {pipeline.score(data.data, data.target):.2%}")
```

2.10 Exercices

Exercice 2.1 ($\star$)**.** Calculez les vecteurs TF-IDF pour le corpus suivant : d_1 = "le chat dort", d_2 = "le chien dort", d_3 = "le chat et le chien".

Exercice 2.2 ($\star$)**.** Montrez que la similarité cosinus est invariante par changement d'échelle positif : $\cos(\lambda \mathbf{u}, \mu \mathbf{v}) = \cos(\mathbf{u}, \mathbf{v})$ pour $\lambda, \mu > 0$.

Exercice 2.3 ($\star\star$)**.** Implémentez un modèle de langue bigramme avec lissage de Laplace. Calculez la perplexité sur un corpus de test.

Exercice 2.4 ($\star\star$)**.** Montrez que le théorème d'Eckart-Young implique que la SVD tronquée donne la meilleure approximation de rang k au sens de la norme de Frobenius.

Exercice 2.5 ($\star\star\star$)**.** Comparez les performances de TF-IDF + régression logistique, TF-IDF + SVM, et LSA + régression logistique sur la classification de sentiments (dataset IMDB). Analysez l'impact de la dimension k de LSA.

Chapitre 3

Plongements de Mots

En 2013, Tomas Mikolov et son équipe chez Google publient un article qui va transformer le traitement du langage naturel : *Efficient Estimation of Word Representations in Vector Space*. L'idée est d'entraîner un réseau de neurones superficiel à prédire les mots à partir de leur contexte (ou inversement), et d'utiliser les poids appris comme représentations vectorielles des mots. Le résultat, Word2Vec, révèle une structure géométrique stupéfiante : les analogies sémantiques deviennent des *opérations vectorielles*. Le vecteur « roi »− « homme »+ « femme »pointe vers « reine ». La géométrie capture le sens.

Mais Word2Vec n'est pas né de rien. Il s'inscrit dans une longue tradition fondée sur l'*hypothèse distributionnelle* de J.R. Firth (1957) : « on connaît un mot par la compagnie qu'il fréquente. »Ce chapitre retrace le chemin qui mène des matrices de co-occurrence aux plongements denses, en passant par l'analyse sémantique latente et GloVe.

L'hypothèse distributionnelle

« You shall know a word by the company it keeps »(J.R. Firth, 1957). Les plongements de mots (*word embeddings*) sont des représentations vectorielles denses qui capturent la similarité sémantique : les mots apparaissant dans des contextes similaires ont des vecteurs proches.

3.1 De la représentation creuse à la représentation dense

Rappelons qu'une représentation one-hot encode chaque mot w_i comme un vecteur $\mathbf{e}_i \in \mathbb{R}^V$ avec $(\mathbf{e}_i)_j = \delta_{ij}$. Cette représentation souffre de deux défauts majeurs :

1. **Dimensionnalité** : V peut atteindre 10^5 à 10^6.

2. **Orthogonalité** : $\langle \mathbf{e}_i, \mathbf{e}_j \rangle = 0$ pour $i \neq j$, même pour des synonymes.

L'objectif des plongements est d'apprendre une fonction $\phi : \mathcal{V} \to \mathbb{R}^d$ avec $d \ll V$ (typiquement $d \in [100, 300]$) telle que la proximité géométrique reflète la similarité sémantique.

3.2 Word2Vec

3.2.1 Architecture CBOW

Définition 3.1 (Continuous Bag of Words (CBOW))**.** Le modèle CBOW prédit le mot central w_t à partir de son contexte $C_t = \{w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}\}$ où c est la taille de la fenêtre. L'objectif est de maximiser :

$$\mathcal{L}_{\text{CBOW}} = \sum_{t=1}^{T} \log P(w_t \mid C_t)$$

avec :

$$P(w_t \mid C_t) = \frac{\exp(\langle \mathbf{v}_{w_t}, \bar{\mathbf{u}}_{C_t} \rangle)}{\sum_{w \in \mathcal{V}} \exp(\langle \mathbf{v}_w, \bar{\mathbf{u}}_{C_t} \rangle)}$$

où $\bar{\mathbf{u}}_{C_t} = \frac{1}{2c} \sum_{w \in C_t} \mathbf{u}_w$ est la moyenne des vecteurs de contexte.

3.2.2 Architecture Skip-gram

Définition 3.2 (Skip-gram)**.** Le modèle Skip-gram prédit les mots du contexte à partir du mot central. L'objectif est de maximiser :

$$\mathcal{L}_{\text{SG}} = \sum_{t=1}^{T} \sum_{\substack{j=-c \\ j \neq 0}}^{c} \log P(w_{t+j} \mid w_t)$$

avec :

$$P(w_o \mid w_i) = \frac{\exp(\langle \mathbf{v}_{w_o}, \mathbf{u}_{w_i} \rangle)}{\sum_{w \in \mathcal{V}} \exp(\langle \mathbf{v}_w, \mathbf{u}_{w_i} \rangle)}$$

Coût du softmax

Le dénominateur du softmax nécessite une somme sur tout le vocabulaire V, ce qui est prohibitif. Deux approximations sont couramment utilisées : l'échantillonnage négatif et le softmax hiérarchique.

3.2.3 Échantillonnage négatif (NEG)

Définition 3.3 (Negative Sampling)**.** L'objectif avec échantillonnage négatif remplace le softmax par :

$$\mathcal{L}_{\text{NEG}} = \log \sigma(\langle \mathbf{v}_{w_o}, \mathbf{u}_{w_i} \rangle) + \sum_{k=1}^{K} \mathbb{E}_{w_k \sim P_n} \left[\log \sigma(-\langle \mathbf{v}_{w_k}, \mathbf{u}_{w_i} \rangle) \right]$$

où K est le nombre d'exemples négatifs et $P_n(w) \propto \text{count}(w)^{3/4}$.

Théorème 3.4 (Équivalence NEG et PMI)**.** *Levy et Goldberg (2014) ont montré que Skip-gram avec échantillonnage négatif factorise implicitement une matrice de PMI décalée :*

$$\mathbf{u}_w^\top \mathbf{v}_c = PMI(w, c) - \log K$$

où $PMI(w, c) = \log \frac{P(w,c)}{P(w)P(c)}$ *est l'information mutuelle ponctuelle.*

3.3 GloVe

Définition 3.5 (GloVe)**.** Le modèle GloVe (Pennington et al., 2014) apprend les plongements en minimisant l'erreur quadratique pondérée sur la matrice de co-occurrence :

$$\mathcal{L}_{\text{GloVe}} = \sum_{i,j=1}^{V} f(X_{ij}) \left(\langle \mathbf{u}_i, \mathbf{v}_j \rangle + b_i + c_j - \log X_{ij} \right)^2$$

où X_{ij} est le nombre de co-occurrences, b_i, c_j sont des biais, et f est une fonction de pondération :

$$f(x) = \begin{cases} (x/x_{\max})^\alpha & \text{si } x < x_{\max} \\ 1 & \text{sinon} \end{cases}$$

avec typiquement $x_{\max} = 100$ et $\alpha = 0.75$.

Remarque 3.6. GloVe combine les avantages des méthodes globales (factorisation matricielle) et locales (fenêtre de contexte). L'objectif GloVe est équivalent à une factorisation de la matrice de log-co-occurrence.

3.4 FastText

Définition 3.7 (FastText)**.** FastText (Bojanowski et al., 2017) enrichit Skip-gram en représentant chaque mot comme la somme de ses n-grammes de caractères. Pour un mot w avec l'ensemble de n-grammes $\mathcal{G}_w$:

$$\mathbf{u}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g$$

où $\mathbf{z}_g \in \mathbb{R}^d$ est le vecteur du n-gramme g.

Avantages de FastText

- Gère les mots hors vocabulaire (OOV) par décomposition en n-grammes.
- Capture la morphologie (préfixes, suffixes).
- Particulièrement efficace pour les langues morphologiquement riches (turc, finnois, arabe).

3.5 Propriétés géométriques

3.5.1 Analogies vectorielles

L'une des propriétés les plus remarquables des plongements est leur capacité à capturer des relations sémantiques par des opérations arithmétiques :

Exemple 3.8.

$$\mathbf{v}_{\text{roi}} - \mathbf{v}_{\text{homme}} + \mathbf{v}_{\text{femme}} \approx \mathbf{v}_{\text{reine}}$$

Formellement, pour résoudre l'analogie $a : b :: c :?$, on cherche :

$$d^* = \arg\max_{d \in \mathcal{V} \setminus \{a,b,c\}} \cos(\mathbf{v}_d, \mathbf{v}_b - \mathbf{v}_a + \mathbf{v}_c)$$

Théorème 3.9 (Structure linéaire des analogies)**.** *Sous l'hypothèse que les plongements factorisent une matrice de PMI (théorème 3.4), les relations sémantiques régulières se manifestent comme des directions linéaires dans l'espace de plongement :*

$$\mathbf{v}_b - \mathbf{v}_a \approx \mathbf{v}_{b'} - \mathbf{v}_{a'}$$

pour toutes les paires (a, b) et (a', b') partageant la même relation.

3.5.2 Visualisation par t-SNE

Visualisation des plongements

```python
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from gensim.models import KeyedVectors

# Charger des plongements pre-entraines
wv = KeyedVectors.load_word2vec_format('GoogleNews-vectors.bin',
↪  binary=True)
words = ['king', 'queen', 'man', 'woman', 'prince', 'princess',
         'paris', 'france', 'berlin', 'germany', 'rome', 'italy']
vectors = np.array([wv[w] for w in words])

tsne = TSNE(n_components=2, random_state=42, perplexity=5)
coords = tsne.fit_transform(vectors)

plt.figure(figsize=(10, 8))
for i, word in enumerate(words):
    plt.scatter(coords[i, 0], coords[i, 1])
    plt.annotate(word, (coords[i, 0]+0.5, coords[i, 1]+0.5))
plt.title("Visualisation t-SNE des plongements Word2Vec")
plt.show()
```

3.6 Évaluation des plongements

3.6.1 Évaluation intrinsèque

- **Analogies** : précision sur des benchmarks d'analogies (Google analogy dataset, BATS).
- **Similarité** : corrélation de Spearman entre la similarité cosinus et les jugements humains (SimLex-999, WordSim-353).

3.6.2 Évaluation extrinsèque

Performance des plongements comme features dans des tâches en aval : classification de sentiments, NER, désambiguation lexicale.

Métriques d'évaluation des plongements

$$\text{Précision analogies} = \frac{\text{analogies correctes}}{\text{total analogies}} \tag{3.1}$$

$$\rho_{\text{Spearman}} = 1 - \frac{6\sum_i d_i^2}{n(n^2-1)} \qquad \text{(corrélation de rang)} \tag{3.2}$$

3.7 Biais dans les plongements

Les plongements appris sur de grands corpus textuels héritent des biais sociétaux présents dans les données.

Exemple 3.10. Bolukbasi et al. (2016) ont montré que :

$$\mathbf{v}_{\text{informaticien}} - \mathbf{v}_{\text{homme}} + \mathbf{v}_{\text{femme}} \approx \mathbf{v}_{\text{ménagère}}$$

Ce résultat reflète des stéréotypes de genre présents dans les corpus.

Définition 3.11 (Débiaisage par projection)**.** Le débiaisage de Bolukbasi et al. consiste à :

1. Identifier la direction de biais $\mathbf{g}$ (par ACP sur des paires de genre).
2. Projeter les mots neutres orthogonalement à $\mathbf{g}$: $\mathbf{v}'_w = \mathbf{v}_w - \langle \mathbf{v}_w, \mathbf{g} \rangle \mathbf{g}$.
3. Égaliser les paires définitionnelles.

3.8 Plongements contextuels vs. statiques

Les plongements statiques (Word2Vec, GloVe, FastText) assignent un vecteur unique à chaque mot, indépendamment du contexte. Cela pose problème pour les mots polysmiques :

Exemple 3.12. Le mot « avocat »désigne soit un fruit, soit un professionnel du droit. Un plongement statique représente ces deux sens par un seul vecteur, qui sera un compromis entre les deux.

Les plongements contextuels (ELMo, BERT, GPT) résolvent ce problème en produisant des représentations différentes selon le contexte. Ils seront étudiés au chapitre 6.

3.9 Implémentation complète

Entraînement Skip-gram avec PyTorch

```python
import torch
import torch.nn as nn

class SkipGram(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.target_embed = nn.Embedding(vocab_size, embed_dim)
        self.context_embed = nn.Embedding(vocab_size, embed_dim)

    def forward(self, target, context, negatives):
        # target: (B,), context: (B,), negatives: (B, K)
        t = self.target_embed(target)           # (B, d)
        c = self.context_embed(context)         # (B, d)
        n = self.context_embed(negatives)       # (B, K, d)

        # Positive score
        pos_score = torch.sum(t * c, dim=1)     # (B,)
        pos_loss = -torch.log(torch.sigmoid(pos_score) + 1e-10)

        # Negative scores
        neg_score = torch.bmm(n, t.unsqueeze(2)).squeeze(2)  # (B, K)
        neg_loss = -torch.log(torch.sigmoid(-neg_score) +
        ↪  1e-10).sum(dim=1)

        return (pos_loss + neg_loss).mean()

model = SkipGram(vocab_size=50000, embed_dim=300)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

3.10 Exercices

Exercice 3.1 ($\star$)**.** Montrez que la similarité cosinus entre deux vecteurs one-hot est nulle pour des mots différents.

Exercice 3.2 ($\star\star$)**.** Dérivez le gradient de l'objectif Skip-gram avec échantillonnage négatif par rapport aux vecteurs $\mathbf{u}_{w_i}$ et $\mathbf{v}_{w_o}$.

Exercice 3.3 ($\star\star$)**.** Montrez que l'objectif GloVe est équivalent à une factorisation de matrice pondérée de la matrice $\log X_{ij}$.

Exercice 3.4 ($\star\star$)**.** Entraînez des plongements Word2Vec (avec `gensim`) sur un corpus français (Wikipédia). Évaluez les analogies "roi – homme + femme = ?" et "Paris – France + Allemagne = ?".

Exercice 3.5 ($\star\star\star$)**.** Implémentez la méthode de débiaisage de Bolukbasi et al. (2016). Mesurez le biais de genre avant et après correction sur un ensemble de mots liés aux professions.

Chapitre 4

Modèles de Séquences

Le sac de mots jette l'ordre par la fenêtre : « le chat mange la souris »et « la souris mange le chat »ont la même représentation. Pour capturer le sens, il faut respecter la séquence. Les réseaux de neurones récurrents (RNN), introduits par Jeffrey Elman en 1990, proposent une solution élégante : un état caché qui se propage d'un mot au suivant, accumulant l'information contextuelle. Mais les RNN simples souffrent du problème du *gradient évanescent*. La solution viendra des cellules LSTM (Hochreiter et Schmidhuber, 1997) et GRU (Cho et al., 2014), qui introduisent des mécanismes de portes pour contrôler le flux d'information.

Modéliser l'ordre des mots

Contrairement au modèle sac-de-mots, les modèles de séquences traitent le texte comme une suite ordonnée de tokens. Les réseaux récurrents (RNN) maintiennent un état caché qui encode l'historique de la séquence, permettant de capturer les dépendances temporelles.

4.1 Réseaux de neurones récurrents (RNN)

Définition 4.1 (RNN – Elman Network)**.** Un réseau récurrent simple (Elman, 1990) traite une séquence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ en maintenant un état caché $\mathbf{h}_t \in \mathbb{R}^h$:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{4.1}$$

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \tag{4.2}$$

où $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, $\mathbf{W}_{xh} \in \mathbb{R}^{h\times d}$, $\mathbf{W}_{hy} \in \mathbb{R}^{o\times h}$ sont les matrices de poids.

Remarque 4.2. L'état caché $\mathbf{h}_t$ agit comme une « mémoire comprimée »de tout l'historique $(\mathbf{x}_1, \ldots, \mathbf{x}_t)$. En théorie, un RNN peut modéliser des dépendances de longueur arbitraire ; en pratique, cette capacité est sévèrement limitée.

4.2 Problème du gradient évanescent et explosif

Théorème 4.3 (Gradient évanescent)**.** *Lors de la rétropropagation à travers le temps (BPTT), le gradient de la perte $\mathcal{L}$ par rapport à $\mathbf{h}_k$ ($k \ll t$) satisfait :*

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}$$

où chaque Jacobien $\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = diag(\tanh'(\cdot))\mathbf{W}_{hh}$.

Si $\|\mathbf{W}_{hh}\| < 1/\gamma$ *(où* $\gamma = \max|\tanh'(\cdot)|$*), le produit des Jacobiens décroît exponentiellement, rendant l'apprentissage des dépendances longues impossible.*

Démonstration. On a $\left\|\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}\right\| \leq \gamma \|\mathbf{W}_{hh}\|$ où $\gamma \leq 1$ pour tanh. Donc :

$$\left\|\prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}\right\| \leq (\gamma \|\mathbf{W}_{hh}\|)^{t-k}$$

Si $\gamma \|\mathbf{W}_{hh}\| < 1$, cette quantité tend vers 0 exponentiellement vite quand $t - k \to \infty$. □

Gradient clipping

Pour contrer le gradient explosif, on utilise le *gradient clipping* :

$$\hat{\mathbf{g}} = \begin{cases} \mathbf{g} & \text{si } \|\mathbf{g}\| \leq \theta \\ \theta \cdot \frac{\mathbf{g}}{\|\mathbf{g}\|} & \text{sinon} \end{cases}$$

où θ est le seuil de clipping (typiquement $\theta \in [1, 5]$).

4.3 Long Short-Term Memory (LSTM)

Définition 4.4 (LSTM)**.** Le réseau LSTM (Hochreiter & Schmidhuber, 1997) introduit une cellule mémoire $\mathbf{c}_t$ et trois portes (gates) pour contrôler le flux d'information :

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \text{(porte d'oubli)} \tag{4.3}$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \text{(porte d'entrée)} \tag{4.4}$$
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) \quad \text{(candidat)} \tag{4.5}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \text{(mise à jour)} \tag{4.6}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \text{(porte de sortie)} \tag{4.7}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \text{(état caché)} \tag{4.8}$$

où $\odot$ dénote le produit de Hadamard (élément par élément).

Proposition 4.5 (Gradient dans les LSTM)**.** Dans un LSTM, le gradient de la perte par rapport à la cellule mémoire $\mathbf{c}_k$ contient un terme :

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} = \prod_{i=k+1}^{t} (\text{diag}(\mathbf{f}_i) + \ldots)$$

La porte d'oubli $\mathbf{f}_i$ permet au gradient de se propager sans atténuation quand $\mathbf{f}_i \approx \mathbf{1}$, résolvant partiellement le problème du gradient évanescent.

4.4 Gated Recurrent Unit (GRU)

Définition 4.6 (GRU)**.** Le GRU (Cho et al., 2014) simplifie le LSTM en fusionnant la cellule mémoire et l'état caché, et en n'utilisant que deux portes :

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z) \quad \text{(porte de mise à jour)} \tag{4.9}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r) \quad \text{(porte de réinitialisation)} \tag{4.10}$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h) \quad \text{(candidat)} \tag{4.11}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \quad \text{(interpolation)} \tag{4.12}$$

Remarque 4.7. Le GRU a moins de paramètres que le LSTM (3 matrices de portes contre 4) et offre des performances comparables dans de nombreuses tâches. Le choix entre LSTM et GRU est souvent empirique.

4.5 RNN bidirectionnels

Définition 4.8 (BiRNN)**.** Un RNN bidirectionnel traite la séquence dans les deux directions et concatène les états cachés :

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] \in \mathbb{R}^{2h}$$

où $\overrightarrow{\mathbf{h}}_t$ est l'état de gauche à droite et $\overleftarrow{\mathbf{h}}_t$ de droite à gauche.

4.6 Architecture encodeur-décodeur (Seq2Seq)

Définition 4.9 (Seq2Seq)**.** L'architecture encodeur-décodeur (Sutskever et al., 2014) utilise un RNN encodeur pour compresser la séquence source en un vecteur de contexte $\mathbf{c} = \mathbf{h}_T^{\text{enc}}$, et un RNN décodeur pour générer la séquence cible :

$$\text{Encodeur :} \quad \mathbf{h}_t^{\text{enc}} = f_{\text{enc}}(\mathbf{x}_t, \mathbf{h}_{t-1}^{\text{enc}}) \tag{4.13}$$

$$\text{Décodeur :} \quad \mathbf{h}_t^{\text{dec}} = f_{\text{dec}}(\mathbf{y}_{t-1}, \mathbf{h}_{t-1}^{\text{dec}}) \tag{4.14}$$

$$P(y_t \mid y_{<t}, \mathbf{x}) = \text{softmax}(\mathbf{W}_o \mathbf{h}_t^{\text{dec}}) \tag{4.15}$$

Goulot d'étranglement

Compresser toute la séquence source dans un seul vecteur $\mathbf{c}$ crée un goulot d'étranglement informationnel. Ce problème est résolu par le mécanisme d'attention (chapitre 5).

4.7 Implémentation avec PyTorch

LSTM pour la classification de texte

```
import torch
import torch.nn as nn
```

```python
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_classes,
                 num_layers=2, dropout=0.3, bidirectional=True):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
        ↪  padding_idx=0)
        self.lstm = nn.LSTM(
            embed_dim, hidden_dim, num_layers=num_layers,
            batch_first=True, dropout=dropout,
            bidirectional=bidirectional
        )
        direction_factor = 2 if bidirectional else 1
        self.classifier = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(hidden_dim * direction_factor, num_classes)
        )

    def forward(self, input_ids, lengths):
        embeds = self.embedding(input_ids)       # (B, T, d)
        packed = nn.utils.rnn.pack_padded_sequence(
            embeds, lengths.cpu(), batch_first=True, enforce_sorted=False
        )
        _, (hidden, _) = self.lstm(packed)       # hidden: (layers*dirs,
        ↪  B, h)
        # Concatenate last forward and backward hidden states
        hidden = torch.cat([hidden[-2], hidden[-1]], dim=1)  # (B, 2h)
        return self.classifier(hidden)           # (B, num_classes)

model = LSTMClassifier(
    vocab_size=30000, embed_dim=256,
    hidden_dim=128, num_classes=2
)
```

Encodeur-décodeur Seq2Seq

```python
class Seq2SeqEncoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers=2):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, num_layers,
                          batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, hidden_dim)

    def forward(self, src):
        embedded = self.embedding(src)
        outputs, hidden = self.rnn(embedded)
        # Combine bidirectional hidden states
        hidden = torch.cat([hidden[-2], hidden[-1]], dim=1)
        hidden = torch.tanh(self.fc(hidden)).unsqueeze(0)
        return outputs, hidden
```

```python
class Seq2SeqDecoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, num_layers,
                          batch_first=True)
        self.fc_out = nn.Linear(hidden_dim, vocab_size)

    def forward(self, input_token, hidden):
        embedded = self.embedding(input_token).unsqueeze(1)
        output, hidden = self.rnn(embedded, hidden)
        prediction = self.fc_out(output.squeeze(1))
        return prediction, hidden
```

4.8 Teacher forcing et scheduled sampling

Définition 4.10 (Teacher forcing)**.** Pendant l'entraînement, le *teacher forcing* consiste à fournir au décodeur le token de référence y_{t-1} plutôt que sa propre prédiction $\hat{y}_{t-1}$. Cela accélère la convergence mais crée un décalage (*exposure bias*) entre entraînement et inférence.

Définition 4.11 (Scheduled sampling)**.** Le *scheduled sampling* (Bengio et al., 2015) atténue l'exposure bias en alternant aléatoirement entre teacher forcing et auto-régression :

$$\text{input}_t = \begin{cases} y_{t-1} & \text{avec probabilité } \epsilon_t \\ \hat{y}_{t-1} & \text{avec probabilité } 1 - \epsilon_t \end{cases}$$

où ϵ_t décroît progressivement de 1 vers 0.

4.9 Exercices

Exercice 4.1 ($\star$)**.** Comptez le nombre total de paramètres d'un LSTM avec $d = 256$ (entrée) et $h = 512$ (état caché).

Exercice 4.2 ($\star$)**.** Expliquez pourquoi un RNN bidirectionnel ne peut pas être utilisé pour la modélisation de langue auto-régressive.

Exercice 4.3 ($\star\star$)**.** Démontrez que le gradient dans un RNN simple décroît exponentiellement avec la distance temporelle sous les conditions du théorème du gradient évanescent.

Exercice 4.4 ($\star\star$)**.** Implémentez un modèle de langue caractère par caractère avec un LSTM à 2 couches. Entraînez-le sur un corpus littéraire et générez du texte par échantillonnage.

Exercice 4.5 ($\star\star\star$)**.** Comparez les performances d'un GRU et d'un LSTM sur une tâche de NER (dataset CoNLL-2003). Analysez la vitesse de convergence et la capacité à capturer les dépendances longues.

Chapitre 5

Mécanisme d'Attention et Transformers

« Attention Is All You Need. »Ce titre, choisi par Vaswani et al. pour leur article de 2017, est devenu l'un des plus célèbres de l'histoire de l'intelligence artificielle. Le *Transformer*, l'architecture qu'ils proposent, abandonne entièrement la récurrence au profit d'un mécanisme d'*attention* qui permet à chaque mot de la séquence de « regarder »directement tous les autres mots, sans restriction de distance. Le résultat : un parallélisme massif lors de l'entraînement, une capacité à capturer les dépendances à longue portée, et des performances qui ont rapidement surpassé tous les modèles existants. Le Transformer est la brique de base de GPT, BERT, et de toute la révolution des grands modèles de langage.

Attention : regarder là où c'est pertinent

Le mécanisme d'attention permet au modèle de se concentrer sur les parties pertinentes de l'entrée à chaque étape de génération, éliminant le goulot d'étranglement de l'encodeur-décodeur. L'architecture Transformer, fondée entièrement sur l'attention, a révolutionné le TAL depuis 2017.

5.1 Attention dans les modèles Seq2Seq

Définition 5.1 (Attention de Bahdanau)**.** L'attention additive (Bahdanau et al., 2015) calcule un vecteur de contexte $\mathbf{c}_t$ comme moyenne pondérée des états de l'encodeur :

$$e_{t,s} = \mathbf{v}_a^\top \tanh(\mathbf{W}_a \mathbf{h}_s^{\text{enc}} + \mathbf{U}_a \mathbf{h}_{t-1}^{\text{dec}}) \tag{5.1}$$

$$\alpha_{t,s} = \frac{\exp(e_{t,s})}{\sum_{s'=1}^{S} \exp(e_{t,s'})} \tag{5.2}$$

$$\mathbf{c}_t = \sum_{s=1}^{S} \alpha_{t,s} \mathbf{h}_s^{\text{enc}} \tag{5.3}$$

où $\alpha_{t,s}$ est le poids d'attention sur la position source s au pas de décodage t.

Définition 5.2 (Attention de Luong)**.** L'attention multiplicative (Luong et al., 2015) utilise un score plus simple :

$$e_{t,s} = (\mathbf{h}_t^{\text{dec}})^\top \mathbf{W}_a \mathbf{h}_s^{\text{enc}}$$

Variantes : *dot* ($e_{t,s} = (\mathbf{h}_t^{\text{dec}})^\top \mathbf{h}_s^{\text{enc}}$), *general*, *concat*.

5.2 Auto-attention (Self-Attention)

Définition 5.3 (Auto-attention par produit scalaire)**.** L'auto-attention (*scaled dot-product attention*) opère sur des vecteurs de requête (*query*) $\mathbf{Q}$, clé (*key*) $\mathbf{K}$ et valeur (*value*) $\mathbf{V}$:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

où d_k est la dimension des clés et le facteur $1/\sqrt{d_k}$ stabilise les gradients.

Théorème 5.4 (Justification du facteur d'échelle)**.** *Soient $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ des vecteurs aléatoires avec composantes i.i.d. de moyenne 0 et variance 1. Alors :*

$$\mathbb{E}[\langle \mathbf{q}, \mathbf{k} \rangle] = 0, \qquad Var[\langle \mathbf{q}, \mathbf{k} \rangle] = d_k$$

Sans le facteur $1/\sqrt{d_k}$, la variance du produit scalaire croît avec d_k, poussant le softmax vers des régions saturées où les gradients sont quasi-nuls.

Démonstration. $\langle \mathbf{q}, \mathbf{k} \rangle = \sum_{i=1}^{d_k} q_i k_i$. Puisque les composantes sont indépendantes de moyenne nulle : $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i]\mathbb{E}[k_i] = 0$ et $\text{Var}[q_i k_i] = \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = 1$. Par indépendance : $\text{Var}[\langle \mathbf{q}, \mathbf{k} \rangle] = d_k$. □

5.3 Attention multi-têtes

Définition 5.5 (Multi-Head Attention)**.** L'attention multi-têtes projette $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ dans H sous-espaces différents et concatène les résultats :

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{5.4}$$

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1; \ldots; \text{head}_H]\mathbf{W}^O \tag{5.5}$$

où $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$, $\mathbf{W}^O \in \mathbb{R}^{Hd_v \times d}$, avec $d_k = d_v = d/H$.

Remarque 5.6. Chaque tête d'attention peut apprendre à se concentrer sur différents types de relations : syntaxiques, sémantiques, positionnelles, etc.

5.4 Architecture Transformer

Définition 5.7 (Bloc Transformer (encodeur))**.** Un bloc encodeur Transformer consiste en :

1. Auto-attention multi-têtes avec connexion résiduelle et normalisation de couche :

$$\mathbf{Z} = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{X}, \mathbf{X}, \mathbf{X}))$$

2. Réseau feed-forward position-wise avec connexion résiduelle :

$$\text{FFN}(\mathbf{z}) = \max(0, \mathbf{z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{Z} + \text{FFN}(\mathbf{Z}))$$

Définition 5.8 (Bloc Transformer (décodeur))**.** Un bloc décodeur Transformer ajoute une couche d'attention croisée (*cross-attention*) entre l'auto-attention masquée et le FFN :

1. Auto-attention masquée (causale)
2. Attention croisée : MultiHead($\mathbf{H}^{\text{dec}}, \mathbf{H}^{\text{enc}}, \mathbf{H}^{\text{enc}}$)
3. Réseau feed-forward

Chaque sous-couche est suivie d'une connexion résiduelle et d'une normalisation.

5.5 Encodage positionnel

L'auto-attention est invariante par permutation. Pour injecter l'information de position, on ajoute un encodage positionnel aux plongements d'entrée.

Définition 5.9 (Encodage positionnel sinusoïdal)**.** L'encodage positionnel de Vaswani et al. (2017) :

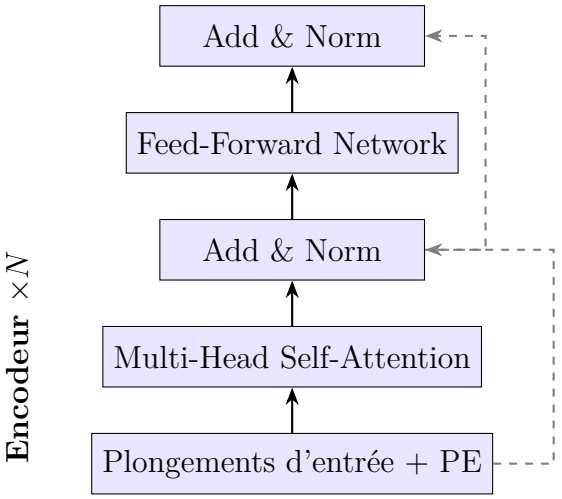
$$\text{PE}(\text{pos}, 2i) = \sin\Big(\frac{\text{pos}}{10000^{2i/d}}\Big) \tag{5.6}$$

$$\text{PE}(\text{pos}, 2i+1) = \cos\Big(\frac{\text{pos}}{10000^{2i/d}}\Big) \tag{5.7}$$

où pos est la position et i l'indice de dimension.

Proposition 5.10 (Propriété de translation)**.** L'encodage sinusoïdal permet de représenter les décalages de position comme des transformations linéaires : il existe une matrice $\mathbf{M}_k$ telle que $\text{PE}(\text{pos}+k) = \mathbf{M}_k \cdot \text{PE}(\text{pos})$.

5.6 Diagramme de l'architecture Transformer



5.7 Complexité computationnelle

Proposition 5.11 (Complexité de l'auto-attention)**.** Pour une séquence de longueur T et dimension d :

- **Auto-attention** : $O(T^2 d)$ en temps et $O(T^2 + Td)$ en mémoire.
- **RNN** : $O(Td^2)$ en temps (séquentiel).
- **FFN** : $O(Td^2)$ en temps (parallélisable).

L'auto-attention est avantageuse quand $T < d$ mais devient prohibitive pour les longues séquences ($T > 4096$).

Nombre de paramètres d'un Transformer

Pour un Transformer à N couches, dimension d, H têtes, FFN de dimension d_{ff} :

$$\text{Params par couche} = 4d^2 + 2d \cdot d_{\text{ff}} + \text{biais}$$

$$\text{Total} \approx N(4d^2 + 2d \cdot d_{\text{ff}}) + V \cdot d$$

5.8 Variantes d'attention efficace

Pour pallier la complexité quadratique, plusieurs variantes ont été proposées :

- **Sparse attention** (Longformer, BigBird) : attention locale + globale, $O(T\sqrt{T})$.
- **Linear attention** (Performers) : approximation par noyaux, $O(Td^2)$.
- **Flash Attention** (Dao et al., 2022) : optimisation au niveau matériel, exact mais avec accès mémoire optimisés.

5.9 Visualisation de l'attention

Visualisation des poids d'attention avec Hugging Face

```python
from transformers import AutoTokenizer, AutoModel
import torch
import matplotlib.pyplot as plt
import seaborn as sns

model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name, output_attentions=True)

text = "The cat sat on the mat because it was tired"
inputs = tokenizer(text, return_tensors="pt")
with torch.no_grad():
    outputs = model(**inputs)

# Attention weights: tuple of (num_layers) tensors of shape (B, H, T, T)
attention = outputs.attentions
layer, head = 6, 3
tokens = tokenizer.convert_ids_to_tokens(inputs["input_ids"][0])
attn_weights = attention[layer][0, head].numpy()
```

```
plt.figure(figsize=(10, 8))
sns.heatmap(attn_weights, xticklabels=tokens, yticklabels=tokens,
↪  cmap="viridis")
plt.title(f"Attention - Couche {layer}, Tete {head}")
plt.tight_layout()
plt.show()
```

5.10 Implémentation complète d'un bloc Transformer

Bloc Transformer encodeur en PyTorch

```
import torch
import torch.nn as nn
import math

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        B, T, d = Q.shape
        q = self.W_q(Q).view(B, T, self.num_heads, self.d_k).transpose(1,
        ↪  2)
        k = self.W_k(K).view(B, -1, self.num_heads,
        ↪  self.d_k).transpose(1, 2)
        v = self.W_v(V).view(B, -1, self.num_heads,
        ↪  self.d_k).transpose(1, 2)

        scores = torch.matmul(q, k.transpose(-2, -1)) /
        ↪  math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, v)
        out = out.transpose(1, 2).contiguous().view(B, T, -1)
        return self.W_o(out)

class TransformerEncoderBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
```

```
        super().__init__()
        self.attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff), nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        x = self.norm1(x + self.dropout(self.attn(x, x, x, mask)))
        x = self.norm2(x + self.dropout(self.ffn(x)))
        return x
```

5.11 Exercices

Exercice 5.1 ($\star$)**.** Vérifiez que $\mathrm{Var}[\langle \mathbf{q}, \mathbf{k} \rangle] = d_k$ lorsque les composantes sont i.i.d. $\mathcal{N}(0, 1)$.

Exercice 5.2 ($\star$)**.** Calculez le nombre de paramètres d'un bloc Transformer encodeur avec $d = 512$, $H = 8$, $d_{\text{ff}} = 2048$.

Exercice 5.3 ($\star\star$)**.** Montrez que l'attention par produit scalaire sans facteur d'échelle peut conduire à des gradients quasi-nuls du softmax pour de grandes valeurs de d_k.

Exercice 5.4 ($\star\star$)**.** Implémentez un masque causal pour l'auto-attention du décodeur et vérifiez qu'il empêche le modèle de "voir le futur".

Exercice 5.5 ($\star\star\star$)**.** Implémentez un Transformer encodeur-décodeur complet pour la traduction français-anglais. Comparez avec un modèle Seq2Seq à base de LSTM avec attention de Bahdanau sur le même jeu de données.

Chapitre 6

Modèles Pré-entraînés

En 2018, trois articles transforment le TAL : ELMo (Peters et al.), GPT (Radford et al.) et surtout BERT (Devlin et al.). Leur idée commune : pré-entraîner un grand modèle de langage sur des milliards de mots non étiquetés, puis l'adapter (*fine-tune*) sur la tâche cible avec très peu de données supervisées. Ce paradigme « pré-entraînement / adaptation »révolutionne la discipline : les représentations apprises capturent la syntaxe, la sémantique, et même des connaissances factuelles du monde, rendant les approches spécialisées antérieures largement obsolètes. C'est le début de l'ère des *foundation models*.

Le paradigme pré-entraînement / adaptation

Plutôt que d'entraîner un modèle de zéro pour chaque tâche, on pré-entraîne un grand modèle sur des données non étiquetées, puis on l'adapte (*fine-tune*) sur des données spécifiques. Ce paradigme, inauguré par ELMo (2018) et popularisé par BERT (2018), a transformé le TAL.

6.1 Apprentissage de représentations contextuelles

Définition 6.1 (Représentation contextuelle)**.** Une *représentation contextuelle* est une fonction $\phi : \mathcal{V}^T \times \{1, \dots, T\} \to \mathbb{R}^d$ qui associe à chaque token x_t un vecteur $\mathbf{h}_t$ dépendant du contexte $(x_1, \dots, x_T)$:

$$\mathbf{h}_t = \phi(\mathbf{x}, t) \neq \phi(\mathbf{x}', t) \quad \text{si } \mathbf{x} \neq \mathbf{x}'$$

même si $x_t = x'_t$.

Cela contraste avec les plongements statiques où $\phi(w)$ est indépendant du contexte.

6.2 ELMo : contexte par RNN bidirectionnel

Définition 6.2 (ELMo)**.** ELMo (Embeddings from Language Models, Peters et al., 2018) produit des représentations contextuelles en combinant les couches d'un BiLSTM pré-entraîné comme modèle de langue bidirectionnel :

$$\text{ELMo}_t = \gamma \sum_{\ell=0}^{L} s_\ell \mathbf{h}_t^\ell$$

où $\mathbf{h}_t^\ell$ est l'état caché de la couche ℓ à la position t, s_ℓ sont des poids appris spécifiques à la tâche, et γ est un facteur d'échelle.

6.3 BERT : Bidirectional Encoder Representations from Transformers

6.3.1 Architecture

BERT utilise un Transformer encodeur (auto-attention bidirectionnelle) avec les configurations suivantes :

	BERT-base	**BERT-large**
Couches N	12	24
Dimension d	768	1024
Têtes H	12	16
Paramètres	110M	340M

6.3.2 Objectifs de pré-entraînement

Définition 6.3 (Masked Language Model (MLM))**.** L'objectif MLM masque aléatoirement 15% des tokens d'entrée et entraîne le modèle à les prédire :

$$\mathcal{L}_{\text{MLM}} = -\sum_{t \in \mathcal{M}} \log P(x_t \mid \mathbf{x}_{\setminus \mathcal{M}})$$

où $\mathcal{M}$ est l'ensemble des positions masquées. Parmi les tokens sélectionnés : 80% sont remplacés par `[MASK]`, 10% par un token aléatoire, 10% inchangés.

Définition 6.4 (Next Sentence Prediction (NSP))**.** L'objectif NSP entraîne le modèle à prédire si la phrase B suit la phrase A dans le corpus original :

$$\mathcal{L}_{\text{NSP}} = -\log P(\text{IsNext} \mid \texttt{[CLS]}, A, \texttt{[SEP]}, B)$$

Limites du MLM

Le masquage crée un décalage entre pré-entraînement (tokens `[MASK]` présents) et inférence (pas de `[MASK]`). Des travaux ultérieurs (RoBERTa, ALBERT) ont montré que le NSP n'est pas nécessaire et que l'entraînement bénéficie de séquences plus longues et de plus de données.

6.3.3 Tokenisation WordPiece

BERT utilise WordPiece, une variante de BPE qui maximise la vraisemblance :

$$\text{score}(a, b) = \frac{\text{count}(ab)}{\text{count}(a) \cdot \text{count}(b)}$$

6.4 GPT : Generative Pre-trained Transformer

Définition 6.5 (GPT)**.** GPT (Radford et al., 2018) utilise un Transformer décodeur (auto-attention causale) pré-entraîné avec un objectif auto-régressif :

$$\mathcal{L}_{\text{GPT}} = -\sum_{t=1}^{T} \log P(x_t \mid x_1, \ldots, x_{t-1})$$

Modèle	Paramètres	Couches	Contexte
GPT-1	117M	12	512
GPT-2	1.5B	48	1024
GPT-3	175B	96	2048
GPT-4	non divulgué	—	128K

Remarque 6.6. GPT-3 a démontré des capacités émergentes en *few-shot learning* : la capacité de résoudre de nouvelles tâches à partir de quelques exemples présentés dans le prompt, sans mise à jour des poids.

6.5 T5 : Text-to-Text Transfer Transformer

Définition 6.7 (T5)**.** T5 (Raffel et al., 2020) reformule toutes les tâches de TAL comme des problèmes texte-vers-texte. Le modèle est un Transformer encodeur-décodeur pré-entraîné avec un objectif de débruitage (*span corruption*) : des segments contigus de tokens sont remplacés par des sentinelles, et le décodeur doit reconstruire les segments masqués.

Comparaison des architectures

	BERT	GPT	T5
Architecture	Encodeur	Décodeur	Enc-Déc
Attention	Bidirectionnelle	Causale	Bid. + Causale
Objectif	MLM + NSP	Auto-régressif	Span corruption
Utilisation	Compréhension	Génération	Les deux

6.6 Lois d'échelle (Scaling Laws)

Théorème 6.8 (Lois d'échelle de Kaplan et al.)**.** *La perte de test L d'un modèle de langue suit des lois de puissance par rapport au nombre de paramètres N, à la taille du dataset D et au budget de calcul C :*

$$L(N) \propto N^{-\alpha_N} \qquad \alpha_N \approx 0.076 \tag{6.1}$$

$$L(D) \propto D^{-\alpha_D} \qquad \alpha_D \approx 0.095 \tag{6.2}$$

$$L(C) \propto C^{-\alpha_C} \qquad \alpha_C \approx 0.050 \tag{6.3}$$

(pour les modèles de type GPT sur des données en anglais).

Remarque 6.9. Les lois Chinchilla (Hoffmann et al., 2022) montrent que l'allocation optimale du budget calcul suit $N \propto D$: il faut augmenter la taille du modèle et des données à la même vitesse.

6.7 Utilisation avec Hugging Face

BERT pour la classification de texte

```
from transformers import AutoTokenizer,
↪  AutoModelForSequenceClassification
from transformers import pipeline

# Pipeline de sentiment
clf = pipeline("sentiment-analysis",
↪  model="nlptown/bert-base-multilingual-uncased-sentiment")
result = clf("Ce cours de NLP est vraiment excellent !")
print(result)
# [{'label': '5 stars', 'score': 0.73}]

# Utilisation directe
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model =
↪  AutoModelForSequenceClassification.from_pretrained("bert-base-uncased",
↪  num_labels=2)
inputs = tokenizer("This is a great course!", return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits  # (1, 2)
```

GPT-2 pour la génération de texte

```
from transformers import pipeline

generator = pipeline("text-generation", model="gpt2")
text = generator(
    "Natural language processing is",
    max_length=50,
    num_return_sequences=1,
    temperature=0.7
)
print(text[0]["generated_text"])
```

T5 pour le résumé

```
from transformers import pipeline

summarizer = pipeline("summarization", model="t5-base")
article = """
Natural language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the
interactions between computers and human language. The goal is to
enable computers to understand, interpret, and generate human language.
"""
summary = summarizer(article, max_length=30, min_length=10)
```

```
print(summary[0]["summary_text"])
```

6.8 Variantes et évolution

RoBERTa (Liu et al., 2019) : supprime NSP, entraîne plus longtemps avec plus de données et des séquences plus longues.

ALBERT (Lan et al., 2020) : factorisation des plongements et partage de paramètres inter-couches pour réduire la taille.

DeBERTa (He et al., 2021) : attention découplée avec position relative.

LLaMA (Touvron et al., 2023) : famille de modèles ouverts de Meta, avec RMSNorm, RoPE, et SwiGLU.

Mistral / Mixtral (2023–2024) : attention groupée (GQA), mélange d'experts (MoE).

6.9 Extraction de features et probing

Définition 6.10 (Probing classifiers)**.** Les *probing classifiers* sont des classifieurs simples (régression logistique) entraînés sur les représentations gelées d'un modèle pré-entraîné pour évaluer quelles informations linguistiques sont capturées par chaque couche.

Des études ont montré que les couches inférieures capturent la morphologie et la syntaxe, tandis que les couches supérieures encodent davantage d'information sémantique.

6.10 Exercices

Exercice 6.1 ($\star$)**.** Comparez le nombre de paramètres de BERT-base et GPT-2 (117M). Quelles différences architecturales expliquent les écarts ?

Exercice 6.2 ($\star$)**.** Expliquez pourquoi BERT ne peut pas être utilisé directement pour la génération de texte auto-régressive.

Exercice 6.3 ($\star\star$)**.** Implémentez un probing classifier pour évaluer si les représentations BERT de la couche 6 encodent l'information de POS tagging. Utilisez le dataset Universal Dependencies.

Exercice 6.4 ($\star\star$)**.** Reproduisez l'expérience de masquage de BERT : calculez la perplexité pseudo-log-vraisemblance $\mathrm{PLL}(\mathbf{x}) = \sum_{t=1}^{T} \log P(x_t \mid \mathbf{x}_{\setminus t})$ sur un ensemble de phrases.

Exercice 6.5 ($\star\star\star$)**.** Étudiez l'impact des lois d'échelle : entraînez des modèles de langue Transformer de tailles croissantes (1M, 10M, 100M paramètres) sur le même corpus et tracez la courbe perte vs. paramètres en échelle log-log.

Chapitre 7

Fine-tuning et Apprentissage par Instruction

En 2018, GPT et BERT démontrent qu'un modèle pré-entraîné sur d'énormes corpus de texte acquiert une compréhension générale du langage, transférable à des tâches spécifiques. Mais comment passer du généraliste au spécialiste ? Le *fine-tuning* — l'ajustement des poids du modèle sur un jeu de données ciblé — est la réponse classique. Pourtant, à mesure que les modèles grossissent (des milliards de paramètres), le fine-tuning complet devient prohibitif. Deux révolutions se succèdent alors : d'abord les méthodes « efficientes en paramètres » comme LoRA (Hu et al., 2022), qui ne modifient qu'une infime fraction des poids ; puis l'*instruction tuning* et le RLHF (Reinforcement Learning from Human Feedback), qui alignent le modèle non plus sur une tâche mais sur des *préférences humaines*. C'est ce dernier ingrédient qui transforme un modèle de langage en assistant conversationnel. Ce chapitre parcourt cette trajectoire, du fine-tuning classique à l'alignement par renforcement.

Adapter un modèle généraliste

Un modèle pré-entraîné est un « généraliste »du langage. Le fine-tuning et l'instruction tuning le transforment en spécialiste d'une tâche ou d'un comportement souhaité, souvent avec des données étiquetées limitées.

7.1 Fine-tuning classique

Définition 7.1 (Fine-tuning)**.** Le *fine-tuning* consiste à ré-entraîner un modèle pré-entraîné θ_{pre} sur un jeu de données spécifique à la tâche $\mathcal{D}_{\text{task}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$:

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), y_i), \quad \theta_0 = \theta_{\text{pre}}$$

Stratégies de fine-tuning

- **Taux d'apprentissage** : utiliser un learning rate faible (2×10^{-5} à 5×10^{-5} pour BERT).
- **Warmup** : augmentation progressive du learning rate sur les premières itérations.

- **Époques** : 2–4 époques suffisent généralement.
- **Gel partiel** : geler les couches inférieures et ne fine-tuner que les couches supérieures pour de petits datasets.

Fine-tuning BERT avec Hugging Face Trainer

```
from transformers import (AutoTokenizer,
↪  AutoModelForSequenceClassification,
                          Trainer, TrainingArguments)
from datasets import load_dataset

dataset = load_dataset("imdb")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

def tokenize_fn(examples):
    return tokenizer(examples["text"], truncation=True,
    ↪  padding="max_length",
                     max_length=256)

tokenized = dataset.map(tokenize_fn, batched=True)
model = AutoModelForSequenceClassification.from_pretrained(
    "bert-base-uncased", num_labels=2
)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=32,
    learning_rate=2e-5,
    warmup_steps=500,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized["train"],
    eval_dataset=tokenized["test"],
)
trainer.train()
```

7.2 Méthodes d'adaptation efficace en paramètres (PEFT)

Le fine-tuning complet de grands modèles est coûteux en mémoire et en calcul. Les méthodes PEFT (*Parameter-Efficient Fine-Tuning*) ne mettent à jour qu'une petite fraction des paramètres.

7.2.1 LoRA : Low-Rank Adaptation

Définition 7.2 (LoRA)**.** LoRA (Hu et al., 2022) injecte des matrices de faible rang dans les couches d'attention. Pour une matrice de poids $\mathbf{W}_0 \in \mathbb{R}^{d \times d}$, la mise à jour est :

$$\mathbf{W} = \mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \mathbf{BA}$$

où $\mathbf{B} \in \mathbb{R}^{d \times r}$, $\mathbf{A} \in \mathbb{R}^{r \times d}$ avec $r \ll d$ (typiquement $r \in [4, 64]$). Seuls $\mathbf{A}$ et $\mathbf{B}$ sont entraînés ; $\mathbf{W}_0$ est gelé.

Proposition 7.3 (Complexité paramétrique de LoRA)**.** Le nombre de paramètres entraînables pour une couche est $2rd$ au lieu de d^2. Le ratio de compression est :

$$\frac{2rd}{d^2} = \frac{2r}{d}$$

Pour $d = 4096$ et $r = 16$: 0.78% des paramètres originaux.

LoRA avec PEFT

```python
from peft import LoraConfig, get_peft_model, TaskType
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf")

lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=16,
    lora_alpha=32,
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
)

model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
# trainable params: 4,194,304 || all params: 6,742,609,920 || trainable%:
↪  0.062
```

7.2.2 QLoRA

Définition 7.4 (QLoRA)**.** QLoRA (Dettmers et al., 2023) combine LoRA avec la quantification 4-bit du modèle de base, permettant le fine-tuning de modèles de 65B paramètres sur un seul GPU de 48 Go.

7.2.3 Autres méthodes PEFT

- **Prefix tuning** (Li & Liang, 2021) : ajout de vecteurs apprenables au début des clés et valeurs d'attention.
- **Adapters** (Houlsby et al., 2019) : insertion de petits réseaux feed-forward entre les couches du Transformer.
- **IA**^{3} (Liu et al., 2022) : vecteurs de mise à l'échelle appris pour les clés, valeurs et FFN.

7.3 Instruction tuning

Définition 7.5 (Instruction tuning)**.** L'*instruction tuning* entraîne un modèle de langue sur un ensemble d'instructions formulées en langage naturel, avec les réponses attendues :

$$\mathcal{D}_{\text{inst}} = \{(\text{instruction}_i, \text{réponse}_i)\}_{i=1}^{N}$$

Le modèle apprend à suivre des instructions générales plutôt qu'à résoudre une tâche unique.

Exemples de modèles : FLAN-T5 (Chung et al., 2022), InstructGPT (Ouyang et al., 2022).

7.4 RLHF : Apprentissage par renforcement à partir de retours humains

Définition 7.6 (RLHF)**.** Le RLHF (*Reinforcement Learning from Human Feedback*) aligne un modèle de langue avec les préférences humaines en trois étapes :

1. **SFT** (Supervised Fine-Tuning) : fine-tuning supervisé sur des démonstrations humaines.
2. **Modèle de récompense** : entraîner un modèle r_ϕ qui prédit la préférence humaine entre deux réponses :

   $$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(y_w, y_l)} \left[\log \sigma(r_\phi(y_w) - r_\phi(y_l))\right]$$

   où y_w est la réponse préférée et y_l la moins bonne.
3. **Optimisation PPO** : optimiser la politique π_θ par proximal policy optimization :

   $$\mathcal{L}_{\text{PPO}} = \mathbb{E}\left[r_\phi(y) - \beta \text{KL}(\pi_\theta \| \pi_{\text{ref}})\right]$$

   où le terme KL empêche la dérive par rapport au modèle de référence π_{ref}.

Objectif DPO (Direct Preference Optimization)

DPO (Rafailov et al., 2023) élimine le modèle de récompense explicite :

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right]$$

7.5 Alignement et sécurité

Définition 7.7 (Modèle aligné)**.** Un modèle de langue est dit *aligné* s'il satisfait les critères **HHH** (Helpful, Honest, Harmless) :

- **Utile** (*Helpful*) : répond aux demandes de manière informative.
- **Honnête** (*Honest*) : ne fabrique pas d'information, exprime son incertitude.
- **Inoffensif** (*Harmless*) : refuse les requêtes dangereuses, évite les contenus toxiques.

Red teaming et jailbreaking

Le *red teaming* consiste à tester systématiquement les limites d'un modèle aligné. Le *jailbreaking* désigne les techniques d'évasion des garde-fous (prompt injection, manipulation du rôle système). L'alignement est un processus continu, pas un état final.

7.6 Transfer learning et adaptation de domaine

Définition 7.8 (Adaptation de domaine)**.** L'*adaptation de domaine* consiste à pré-entraîner davantage un modèle sur des données du domaine cible avant le fine-tuning spécifique. Exemples : BioBERT (biomédical), SciBERT (scientifique), FinBERT (finance).

Théorème 7.9 (Borne de généralisation en transfer learning)**.** *Soit $\epsilon_S(\theta)$ l'erreur sur le domaine source et $\epsilon_T(\theta)$ l'erreur sur le domaine cible. Sous des hypothèses de divergence bornée $d_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T)$:*

$$\epsilon_T(\theta) \leq \epsilon_S(\theta) + d_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda$$

où λ est la somme des erreurs optimales sur les deux domaines.

7.7 Exercices

Exercice 7.1 ($\star$)**.** Calculez le nombre de paramètres entraînables avec LoRA ($r = 8$) appliqué aux 4 projections d'attention d'un modèle avec $d = 4096$ et $N = 32$ couches.

Exercice 7.2 ($\star\star$)**.** Implémentez le fine-tuning de BERT pour la classification de sentiments sur IMDB avec Hugging Face `Trainer`. Comparez les performances avec gel total, gel partiel (6 premières couches), et fine-tuning complet.

Exercice 7.3 ($\star\star$)**.** Dérivez l'objectif DPO à partir de l'objectif RLHF en montrant que la solution optimale de la politique sous contrainte KL a une forme analytique.

Exercice 7.4 ($\star\star$)**.** Comparez LoRA et le fine-tuning complet en termes de performance et de temps d'entraînement sur une tâche de NER.

Exercice 7.5 ($\star\star\star$)**.** Implémentez un pipeline RLHF simplifié : (1) SFT sur un petit jeu de données d'instructions, (2) modèle de récompense par préférences binaires, (3) optimisation avec PPO (utilisez `trl`).

Chapitre 8

Génération de Texte

Un modèle de langue qui prédit le mot suivant est, littéralement, un générateur de texte en puissance. Mais entre la distribution $P(x_t \mid x_{<t})$ et un texte cohérent, fluide et informatif, il y a un gouffre — celui de la *stratégie de décodage*. Prendre systématiquement le mot le plus probable (décodage glouton) produit un texte répétitif et ennuyeux. Échantillonner naïvement dans la distribution produit un texte incohérent. Les solutions — beam search, top-k, nucleus sampling (top-p), température — sont autant de compromis entre *qualité* et *diversité*. Ari Holtzman et ses collaborateurs, dans leur article "The Curious Case of Neural Text Degeneration" (2020), ont montré que le nucleus sampling résout élégamment le problème de dégénérescence. Ce chapitre explore ces stratégies et leurs fondements probabilistes.

Du modèle de langue au générateur

Un modèle de langue auto-régressif définit $P(x_t \mid x_{<t})$. Générer du texte revient à échantillonner séquentiellement dans cette distribution. Le choix de la stratégie de décodage influence radicalement la qualité, la diversité et la cohérence du texte produit.

8.1 Décodage glouton

Définition 8.1 (Décodage glouton)**.** Le décodage glouton (*greedy decoding*) sélectionne à chaque pas le token le plus probable :

$$x_t = \arg\max_{w \in \mathcal{V}} P(w \mid x_{<t})$$

Limites du décodage glouton

Le décodage glouton ne garantit pas la séquence globalement optimale. Il peut produire des répétitions et manque de diversité. La séquence la plus probable n'est pas toujours la plus naturelle.

8.2 Recherche en faisceau (Beam Search)

Définition 8.2 (Beam search)**.** La recherche en faisceau maintient les B séquences partielles les plus probables (*beams*) à chaque pas de décodage :

$$\mathcal{B}_t = \text{top-}B\left\{(s \oplus w, \text{score}(s) + \log P(w \mid s)) : s \in \mathcal{B}_{t-1}, w \in \mathcal{V}\right\}$$

où B est la largeur du faisceau.

Proposition 8.3 (Propriétés du beam search)**.**

1. Pour $B = 1$: équivalent au décodage glouton.
2. Pour $B = |\mathcal{V}|^T$: équivalent à la recherche exhaustive.
3. En pratique, $B \in [4, 10]$ offre un bon compromis.

8.2.1 Normalisation par la longueur

Le beam search favorise les séquences courtes (plus de termes $\log P < 0$ pour les longues). On normalise le score par la longueur :

$$\text{score}_{\text{norm}}(s) = \frac{1}{|s|^\alpha} \sum_{t=1}^{|s|} \log P(x_t \mid x_{<t})$$

où $\alpha \in [0.6, 1.0]$ est un hyperparamètre (Wu et al., 2016).

8.3 Échantillonnage stochastique

Définition 8.4 (Échantillonnage avec température)**.** L'échantillonnage avec température $\tau > 0$ modifie la distribution :

$$P_\tau(w \mid x_{<t}) = \frac{\exp(\text{logit}_w / \tau)}{\sum_{w'} \exp(\text{logit}_{w'} / \tau)}$$

- $\tau \to 0$: converge vers le décodage glouton.
- $\tau = 1$: distribution originale.
- $\tau > 1$: distribution plus uniforme (plus de diversité).

Définition 8.5 (Top-k sampling)**.** Le top-k sampling (Fan et al., 2018) restreint l'échantillonnage aux k tokens les plus probables :

$$P_{\text{top-}k}(w \mid x_{<t}) = \begin{cases} \frac{P(w|x_{<t})}{\sum_{w' \in \mathcal{V}_k} P(w'|x_{<t})} & \text{si } w \in \mathcal{V}_k \\ 0 & \text{sinon} \end{cases}$$

où $\mathcal{V}_k$ contient les k tokens les plus probables.

Définition 8.6 (Top-p sampling (nucleus))**.** Le top-p sampling (Holtzman et al., 2020) sélectionne le plus petit ensemble $\mathcal{V}_p$ tel que :

$$\sum_{w \in \mathcal{V}_p} P(w \mid x_{<t}) \geq p$$

puis échantillonne uniformément dans $\mathcal{V}_p$ (après renormalisation).

Combinaison des stratégies

En pratique, on combine souvent température, top-k et top-p. Par exemple, $\tau = 0.7$, $k = 50$, $p = 0.9$ offre un bon équilibre entre cohérence et diversité pour la génération créative.

8.4 Pénalités de répétition

Définition 8.7 (Pénalité de répétition)**.** Pour éviter les répétitions, on pénalise les tokens déjà générés :

$$\text{logit}'_w = \begin{cases} \text{logit}_w / \theta & \text{si } w \in x_{<t} \text{ et } \text{logit}_w > 0 \\ \text{logit}_w \cdot \theta & \text{si } w \in x_{<t} \text{ et } \text{logit}_w \leq 0 \end{cases}$$

où $\theta > 1$ est le facteur de pénalité.

8.5 Décodage contrasté

Définition 8.8 (Contrastive decoding)**.** Le décodage contrasté (Li et al., 2023) sélectionne les tokens qui maximisent la différence de log-probabilité entre un modèle expert π_{exp} et un modèle amateur π_{ama} :

$$x_t = \arg\max_{w \in \mathcal{V}_p} \left[\log \pi_{\text{exp}}(w \mid x_{<t}) - \alpha \log \pi_{\text{ama}}(w \mid x_{<t}) \right]$$

8.6 Décodage spéculatif

Définition 8.9 (Speculative decoding)**.** Le décodage spéculatif (Leviathan et al., 2023) utilise un petit modèle rapide (*draft model*) pour proposer K tokens, puis un grand modèle les vérifie en parallèle. Les tokens acceptés sont ceux dont la probabilité sous le grand modèle est suffisante, ce qui accélère la génération sans changer la distribution.

8.7 Métriques de qualité de génération

Métriques de génération

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n \right) \qquad \text{(précision } n\text{-grammes)} \tag{8.1}$$

$$\text{ROUGE-L} = F_1(\text{LCS}) \qquad \text{(plus longue sous-séquence commune)} \tag{8.2}$$

$$\text{METEOR} = F_1 \cdot (1 - \text{Penalty}) \qquad \text{(alignement flexible)} \tag{8.3}$$

$$\text{BERTScore} = F_1(\text{sim cosinus BERT}) \qquad \text{(similarité contextuelle)} \tag{8.4}$$

8.8 Implémentation

Stratégies de décodage avec Hugging Face

```
from transformers import AutoTokenizer, AutoModelForCausalLM

model_name = "gpt2-medium"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

prompt = "The future of artificial intelligence"
input_ids = tokenizer.encode(prompt, return_tensors="pt")

# Greedy decoding
greedy = model.generate(input_ids, max_length=50)

# Beam search
beam = model.generate(input_ids, max_length=50, num_beams=5,
                      length_penalty=0.8, no_repeat_ngram_size=3)

# Top-k + Top-p sampling
sampled = model.generate(input_ids, max_length=50, do_sample=True,
                         temperature=0.7, top_k=50, top_p=0.9,
                         repetition_penalty=1.2)

for name, ids in [("Greedy", greedy), ("Beam", beam), ("Sampled",
↪  sampled)]:
    print(f"{name}: {tokenizer.decode(ids[0],
    ↪  skip_special_tokens=True)}")
```

Décodage pas à pas

```
import torch
import torch.nn.functional as F

def generate_top_p(model, input_ids, max_new_tokens=50, temperature=0.8,
                   top_p=0.9):
    generated = input_ids.clone()
    for _ in range(max_new_tokens):
        with torch.no_grad():
            outputs = model(generated)
            logits = outputs.logits[:, -1, :] / temperature

        # Top-p filtering
        sorted_logits, sorted_indices = torch.sort(logits,
        ↪  descending=True)
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1),
        ↪  dim=-1)
        mask = cumulative_probs - F.softmax(sorted_logits, dim=-1) >=
        ↪  top_p
```

```
        sorted_logits[mask] = float('-inf')
        logits.scatter_(1, sorted_indices, sorted_logits)

        probs = F.softmax(logits, dim=-1)
        next_token = torch.multinomial(probs, num_samples=1)
        generated = torch.cat([generated, next_token], dim=-1)

        if next_token.item() == tokenizer.eos_token_id:
            break
    return generated
```

8.9 Génération contrôlée

Définition 8.10 (Génération contrôlée)**.** La *génération contrôlée* vise à orienter la sortie d'un modèle de langue selon des attributs spécifiés (ton, sujet, style) sans ré-entraînement. Approches :

- **PPLM** (Dathathri et al., 2020) : perturbation des gradients.
- **Classifier-free guidance** : interpolation entre distributions conditionnée et non conditionnée.
- **Constrained decoding** : contraintes lexicales dures.

8.10 Exercices

Exercice 8.1 ($\star$)**.** Montrez que le décodage glouton est un cas particulier du beam search avec $B = 1$.

Exercice 8.2 ($\star$)**.** Calculez la distribution top-p pour $p = 0.9$ étant donné les probabilités $(0.4, 0.3, 0.15, 0.1, 0.05)$.

Exercice 8.3 ($\star\star$)**.** Implémentez le beam search avec normalisation par la longueur. Testez avec différentes valeurs de α et B sur GPT-2.

Exercice 8.4 ($\star\star$)**.** Comparez quantitativement (BLEU, diversité) la génération par greedy, beam search ($B = 5$), et nucleus sampling ($p = 0.9$) sur un corpus de test.

Exercice 8.5 ($\star\star\star$)**.** Implémentez le décodage spéculatif avec un modèle draft (GPT-2 small) et un modèle cible (GPT-2 large). Mesurez l'accélération obtenue et vérifiez que la distribution de sortie est préservée.

Chapitre 9

RAG — Génération Augmentée par Récupération

Les grands modèles de langue sont remarquablement compétents, mais ils ont deux faiblesses structurelles : ils « hallucinent » (génèrent des informations plausibles mais fausses) et leurs connaissances sont figées à la date d'entraînement. En 2020, Patrick Lewis et ses collaborateurs chez Facebook AI Research proposent une solution élégante : la *Génération Augmentée par Récupération* (RAG). Le principe est de coupler le modèle génératif à un système de recherche documentaire : avant de générer une réponse, le modèle récupère des passages pertinents dans une base de connaissances externe et les utilise comme contexte. Cette architecture découple la *mémoire* (la base de documents, facilement mise à jour) du *raisonnement* (le modèle de langue). Le RAG est aujourd'hui au cœur des systèmes de question-réponse d'entreprise, des chatbots spécialisés et des assistants de recherche.

Pourquoi augmenter par la récupération ?

Les modèles de langue génératifs (LLM) souffrent d'hallucinations et de connaissances figées à la date d'entraînement. La Génération Augmentée par Récupération (RAG) résout ces problèmes en fournissant au modèle des documents pertinents extraits d'une base de connaissances externe avant la génération.

9.1 Principe du RAG

Définition 9.1 (Génération Augmentée par Récupération)**.** Un système **RAG** combine un module de *récupération* (retriever) et un module de *génération* (generator). Soit q une requête et $\mathcal{C}$ un corpus de documents :

1. **Récupération** : extraire les k documents les plus pertinents $\{d_1, \ldots, d_k\} = \text{Retrieve}(q, \mathcal{C})$
2. **Génération** : produire la réponse $y = \text{Generate}(q, d_1, \ldots, d_k)$

Formulation probabiliste du RAG

$$P(y|q) = \sum_{d \in \mathcal{C}} P(d|q)\, P(y|q,d) \approx \sum_{i=1}^{k} P(d_i|q)\, P_{\text{LLM}}(y|q,d_i)$$

où $P(d|q)$ est le score de pertinence et $P_{\text{LLM}}(y|q,d)$ est la probabilité générative conditionnée sur le document.

Proposition 9.2 (Avantages du RAG)**.**

1. **Connaissances à jour** : la base de connaissances peut être mise à jour sans ré-entraîner le modèle.
2. **Traçabilité** : les sources sont identifiables (citations).
3. **Réduction des hallucinations** : le modèle s'appuie sur des faits vérifiables.
4. **Efficacité paramétrique** : un modèle plus petit peut atteindre la qualité d'un grand modèle.

9.2 Récupération dense

Définition 9.3 (Récupération dense (DPR))**.** Dense Passage Retrieval (Karpukhin et al., 2020) encode les requêtes et les documents dans un espace vectoriel commun :

$$\text{sim}(q,d) = \mathbf{e}_q^\top \mathbf{e}_d = E_Q(q)^\top E_D(d)$$

où E_Q et E_D sont des encodeurs BERT fine-tunés pour maximiser la similarité entre requêtes et passages pertinents.

Définition 9.4 (Fonction de perte contrastive pour DPR)**.** L'entraînement utilise une perte contrastive sur des lots de positifs et négatifs :

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(q,d^+))}{\exp(\text{sim}(q,d^+)) + \sum_{j=1}^{n} \exp(\text{sim}(q,d_j^-))}$$

où d^+ est le passage pertinent et $\{d_j^-\}$ sont les négatifs.

Définition 9.5 (ColBERT)**.** ColBERT (Khattab et al., 2020) utilise une interaction *late* entre les tokens de la requête et du document :

$$\text{sim}(q,d) = \sum_{i=1}^{|q|} \max_{j=1}^{|d|} \mathbf{q}_i^\top \mathbf{d}_j$$

où $\mathbf{q}_i$ et $\mathbf{d}_j$ sont les embeddings contextuels de chaque token. Cette interaction fine permet une meilleure précision tout en restant efficace grâce au pré-calcul des embeddings de documents.

9.3 Bases de données vectorielles

Définition 9.6 (Base de données vectorielle)**.** Une **base de données vectorielle** stocke des vecteurs d'embeddings et permet la recherche de plus proches voisins (ANN — Approximate Nearest Neighbors) en temps sous-linéaire. Exemples : FAISS, Pinecone, Weaviate, Qdrant, ChromaDB.

Proposition 9.7 (Méthodes d'indexation)**.** Les principales méthodes d'indexation pour la recherche ANN sont :

- **IVF** (Inverted File) : partition de l'espace en cellules de Voronoï.
- **HNSW** (Hierarchical Navigable Small World) : graphe navigable hiérarchique, offrant un excellent compromis vitesse/précision.
- **PQ** (Product Quantization) : compression des vecteurs par quantification pour réduire l'empreinte mémoire.

9.4 Stratégies de découpage (Chunking)

Définition 9.8 (Découpage de documents)**.** Le **chunking** consiste à découper les documents en fragments de taille adaptée au contexte du modèle :

- **Taille fixe** : n tokens avec chevauchement de m tokens.
- **Par paragraphes** : respecte la structure du document.
- **Sémantique** : découpe aux changements de sujet (utilisant un modèle de segmentation).
- **Récursif** : hiérarchie de découpages (titre → section → paragraphe).

Taille des chunks

Des chunks trop petits perdent le contexte ; des chunks trop grands diluent l'information pertinente et gâchent le budget de tokens du modèle. En pratique, 256–512 tokens avec 50–100 de chevauchement offrent un bon compromis.

9.5 Architectures RAG

Définition 9.9 (RAG-Sequence et RAG-Token)**.** Lewis et al. (2020) proposent deux variantes :

- **RAG-Sequence** : un seul document est utilisé pour générer toute la réponse : $P(y|q) \approx \sum_d P(d|q) \prod_t P(y_t|q, d, y_{<t})$
- **RAG-Token** : le document peut changer à chaque token : $P(y|q) \approx \prod_t \sum_d P(d|q) P(y_t|q, d, y_{<t})$

Remarque 9.10. Les architectures RAG modernes utilisent souvent un pipeline plus complexe : reranking des documents récupérés, fusion de contexte, et génération itérative (le modèle peut décider de faire des requêtes supplémentaires).

9.5.1 RAG avancé

Définition 9.11 (Self-RAG)**.** Self-RAG (Asai et al., 2023) apprend au modèle à décider *quand* récupérer (via des tokens spéciaux de réflexion) et à évaluer la pertinence des documents récupérés.

Définition 9.12 (RAPTOR)**.** RAPTOR construit un arbre hiérarchique de résumés : les feuilles sont les chunks originaux, les noeuds internes sont des résumés de clusters. La recherche parcourt l'arbre du général au spécifique.

9.6 Évaluation des systèmes RAG

Définition 9.13 (Métriques RAG)**.** L'évaluation d'un système RAG couvre trois dimensions :

1. **Qualité de récupération** : recall@k, precision@k, MRR (Mean Reciprocal Rank).
2. **Fidélité (faithfulness)** : la réponse est-elle cohérente avec les documents récupérés ?
3. **Pertinence de la réponse** : la réponse répond-elle correctement à la question ?

Recall@k et MRR

$$\text{Recall@}k = \frac{|\{d \in \text{top-}k : d \in \mathcal{R}\}|}{|\mathcal{R}|} \qquad \text{MRR} = \frac{1}{|Q|}\sum_{q \in Q} \frac{1}{\text{rank}(d_q^+)}$$

où $\mathcal{R}$ est l'ensemble des documents pertinents et d_q^+ est le premier document pertinent.

Exemple 9.14 (RAGAS)**.** Le framework RAGAS évalue automatiquement les systèmes RAG selon quatre métriques : fidélité, pertinence de la réponse, pertinence du contexte, et rappel du contexte. Il utilise un LLM comme juge pour évaluer chaque dimension.

9.7 Implémentation Python

Pipeline RAG minimaliste

```python
from sentence_transformers import SentenceTransformer
import numpy as np

class SimpleRAG:
    def __init__(self, docs, model_name="all-MiniLM-L6-v2"):
        self.encoder = SentenceTransformer(model_name)
        self.docs = docs
        self.embeddings = self.encoder.encode(docs)

    def retrieve(self, query, k=3):
        q_emb = self.encoder.encode([query])
        scores = (self.embeddings @ q_emb.T).squeeze()
        top_k = np.argsort(scores)[-k:][::-1]
        return [(self.docs[i], scores[i]) for i in top_k]

    def generate_prompt(self, query, k=3):
        results = self.retrieve(query, k)
        context = "\n\n".join([doc for doc, _ in results])
        return f"Context:\n{context}\n\nQuestion: {query}\nAnswer:"

# Usage
docs = ["Paris est la capitale de la France.",
        "Berlin est la capitale de l'Allemagne.",
```

```
        "La tour Eiffel mesure 330 metres."]
rag = SimpleRAG(docs)
prompt = rag.generate_prompt("Quelle est la capitale de la France?")
```

9.8 Exercices

Exercice 9.1 (Pipeline RAG)**.** Implémenter un pipeline RAG complet avec ChromaDB comme base vectorielle et un modèle HuggingFace pour la génération. Tester sur un corpus de pages Wikipédia en français.

Exercice 9.2 (Stratégies de chunking)**.** Comparer trois stratégies de découpage (fixe, par paragraphe, sémantique) sur un corpus de 100 documents. Mesurer le recall@5 pour un ensemble de 50 questions.

Exercice 9.3 (DPR vs BM25)**.** Comparer DPR (récupération dense) avec BM25 (récupération creuse) sur le dataset Natural Questions. Dans quels cas la récupération dense surpasse-t-elle BM25 ? Et vice-versa ?

Exercice 9.4 (Évaluation RAG)**.** Concevoir un protocole d'évaluation pour un système RAG de question-réponse médical. Quelles métriques sont critiques ? Comment gérer les hallucinations dans un contexte médical ?

Chapitre 10

Évaluation des Modèles de Langage

Comment savoir si un modèle de langage est “bon” ? La question est plus épineuse qu'il n'y paraît. Contrairement à la classification d'images (où la précision est sans ambiguïté), le langage est subjectif, contextuel et multidimensionnel. Une traduction peut être fidèle mais maladroite ; un résumé peut être fluide mais infidèle ; une réponse peut être factuelle mais inutile. Les métriques automatiques — BLEU (Papineni et al., 2002), ROUGE, perplexité — capturent certains aspects mais échouent à mesurer la qualité perçue par les humains. Les benchmarks (GLUE, SuperGLUE, MMLU) fournissent des comparaisons standardisées, mais saturent rapidement. L'évaluation humaine reste l'étalon-or, mais elle est coûteuse et peu reproductible. Ce chapitre explore cet écosystème de métriques, ses forces et ses limites.

Pourquoi évaluer est difficile

Évaluer un système de TAL est fondamentalement difficile car le langage est ambigu, subjectif et dépendant du contexte. Une traduction peut être fidèle mais maladroite, un résumé peut être fluide mais infidèle. Chaque tâche nécessite des métriques adaptées, et aucune métrique automatique ne capture parfaitement la qualité perçue par les humains.

10.1 Évaluation intrinsèque vs extrinsèque

Définition 10.1 (Évaluation intrinsèque)**.** L'**évaluation intrinsèque** mesure la qualité d'un composant isolé, indépendamment de son application finale. Exemples : perplexité d'un modèle de langue, précision d'un étiqueteur POS.

Définition 10.2 (Évaluation extrinsèque)**.** L'**évaluation extrinsèque** mesure l'impact d'un composant sur une tâche applicative complète. Exemple : l'impact d'un meilleur modèle de langue sur la qualité d'un système de dialogue.

Remarque 10.3. L'évaluation intrinsèque est reproductible et peu coûteuse mais ne garantit pas l'utilité pratique. L'évaluation extrinsèque est plus pertinente mais plus coûteuse et difficult à isoler.

10.2 Perplexité

Définition 10.4 (Perplexité)**.** La **perplexité** d'un modèle de langue P sur un corpus de test $w_1, \ldots, w_N$ est :

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i})\right) = \exp(H_{\text{cross}})$$

où H_{cross} est l'entropie croisée entre la distribution réelle et le modèle.

Proposition 10.5 (Interprétation de la perplexité)**.**

- $\text{PPL} = |\mathcal{V}|$: le modèle est uniforme (aucune prédiction).
- $\text{PPL} = 1$: prédiction parfaite.
- En pratique, GPT-2 atteint PPL ≈ 22 sur WikiText-103.

Limites de la perplexité

La perplexité ne mesure que la capacité prédictive du modèle sur le corpus de test. Elle ne capture pas la cohérence, la fidélité factuelle, ou la qualité stylistique. Deux modèles avec la même perplexité peuvent générer du texte de qualités très différentes.

10.3 BLEU

Définition 10.6 (BLEU)**.** BLEU (Bilingual Evaluation Understudy, Papineni et al., 2002) mesure la qualité d'une traduction en comparant les n-grammes entre la sortie candidate c et les références $\{r\}$:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N}\frac{1}{N}\log p_n\right)$$

où p_n est la précision modifiée des n-grammes et BP est la pénalité de brièveté.

Composantes de BLEU

Précision n-gramme :

$$p_n = \frac{\sum_{n\text{-gram}\in c}\min(\text{count}(c), \max_r \text{count}(r))}{\sum_{n\text{-gram}\in c}\text{count}(c)}$$

Pénalité de brièveté :

$$\text{BP} = \begin{cases} 1 & \text{si } |c| > |r| \\ \exp(1 - |r|\,/\,|c|) & \text{sinon} \end{cases}$$

Remarque 10.7. BLEU-4 (avec $N = 4$) est le standard en traduction automatique. Il corrèle raisonnablement avec le jugement humain au niveau du corpus mais pas au niveau de la phrase individuelle.

10.4 ROUGE

Définition 10.8 (ROUGE)**.** ROUGE (Recall-Oriented Understudy for Gisting Evaluation, Lin, 2004) mesure la qualité des résumés en se concentrant sur le rappel :

- **ROUGE-N** : rappel des n-grammes : ROUGE-N $= \frac{\sum_{n\text{-gram}\in r} \min(\text{count}(c),\text{count}(r))}{\sum_{n\text{-gram}\in r} \text{count}(r)}$
- **ROUGE-L** : plus longue sous-séquence commune (LCS).

Proposition 10.9 (BLEU vs ROUGE)**.**

	BLEU	**ROUGE**
Orientation	Précision	Rappel
Usage principal	Traduction	Résumé
Granularité	Corpus	Document

10.5 BERTScore

Définition 10.10 (BERTScore)**.** BERTScore (Zhang et al., 2020) utilise les embeddings contextuels de BERT pour calculer la similarité entre tokens de la référence et du candidat :

$$P_{\text{BERT}} = \frac{1}{|c|} \sum_{c_i \in c} \max_{r_j \in r} \mathbf{c}_i^\top \mathbf{r}_j, \qquad R_{\text{BERT}} = \frac{1}{|r|} \sum_{r_j \in r} \max_{c_i \in c} \mathbf{r}_j^\top \mathbf{c}_i$$

$$F_{\text{BERT}} = 2 \cdot \frac{P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}$$

Remarque 10.11. BERTScore capture la similarité sémantique plutôt que la correspondance exacte des mots, ce qui le rend plus robuste aux paraphrases. Il corrèle mieux avec le jugement humain que BLEU et ROUGE sur de nombreuses tâches.

10.6 Évaluation humaine

Définition 10.12 (Protocoles d'évaluation humaine)**.** L'évaluation humaine reste le gold standard. Les protocoles courants :

1. **Échelle de Likert** : les annotateurs notent de 1 à 5 selon des critères (fluence, cohérence, pertinence).
2. **Comparaison par paires** : "Quelle réponse est meilleure ? A ou B ?" (plus fiable que l'échelle absolue).
3. **Classement** : ordonner n sorties de la meilleure à la pire.
4. **Elo rating** : classement dynamique par comparaisons itératives (utilisé par Chatbot Arena).

Définition 10.13 (Accord inter-annotateurs)**.** Le **kappa de Cohen** mesure l'accord entre deux annotateurs corrigé du hasard :

$$\kappa = \frac{P_o - P_e}{1 - P_e}$$

où P_o est l'accord observé et P_e l'accord attendu par hasard. $\kappa > 0.8$ indique un excellent accord.

Biais de l'évaluation humaine

L'évaluation humaine souffre de biais : préférence pour les réponses longues, biais de position (la première option est souvent préférée), fatigue des annotateurs, et variabilité culturelle. Des protocoles rigoureux (randomisation, critères explicites, calibration) sont essentiels.

10.7 Benchmarks : GLUE et SuperGLUE

Définition 10.14 (GLUE)**.** Le benchmark **GLUE** (General Language Understanding Evaluation, Wang et al., 2018) regroupe 9 tâches de compréhension du langage :

- Inférence textuelle : MNLI, RTE, WNLI
- Similarité : STS-B, MRPC, QQP
- Sentiment : SST-2
- Acceptabilité : CoLA
- Question-réponse : QNLI

Le score GLUE est la moyenne des performances sur toutes les tâches.

Définition 10.15 (SuperGLUE)**. SuperGLUE** (Wang et al., 2019) succède à GLUE avec des tâches plus difficiles : BoolQ, CB, COPA, MultiRC, ReCoRD, RTE, WiC, WSC. Les modèles actuels dépassent la performance humaine sur SuperGLUE.

Remarque 10.16. Depuis que les LLM ont saturé GLUE et SuperGLUE, de nouveaux benchmarks plus exigeants sont apparus : MMLU (connaissances multi-domaines), BIG-Bench (tâches émergentes), HELM (évaluation holistique), et LMSYS Chatbot Arena (évaluation par les utilisateurs).

10.8 LLM comme juge

Définition 10.17 (LLM-as-a-Judge)**.** L'approche **LLM-as-a-Judge** utilise un modèle de langue puissant (GPT-4, Claude) pour évaluer les sorties d'autres modèles. Le juge reçoit un prompt structuré avec les critères d'évaluation et fournit un score ou un classement.

Pièges des classements

Les classements (leaderboards) peuvent être trompeurs :

1. **Overfitting au benchmark** : les modèles sont optimisés spécifiquement pour les tests.
2. **Contamination des données** : les données de test peuvent apparaître dans les corpus d'entraînement.

3. **Métriques inappropriées** : un score unique masque les forces et faiblesses.
4. **Manque de diversité** : les benchmarks sous-représentent certaines langues et cultures.

10.9 Implémentation Python

Calcul de BLEU et ROUGE

```python
from nltk.translate.bleu_score import sentence_bleu
from rouge_score import rouge_scorer

# BLEU
reference = [["le", "chat", "est", "sur", "le", "tapis"]]
candidate = ["le", "chat", "est", "assis", "sur", "le", "tapis"]
bleu = sentence_bleu(reference, candidate)
print(f"BLEU: {bleu:.4f}")

# ROUGE
scorer = rouge_scorer.RougeScorer(
    ['rouge1', 'rouge2', 'rougeL'], use_stemmer=True)
ref = "le chat est sur le tapis"
hyp = "le chat est assis sur le tapis"
scores = scorer.score(ref, hyp)
for key, val in scores.items():
    print(f"{key}: P={val.precision:.3f} R={val.recall:.3f} "
          f"F={val.fmeasure:.3f}")
```

BERTScore

```python
from bert_score import score as bert_score

refs = ["Le chat est sur le tapis."]
hyps = ["Le chat est assis sur le tapis."]
P, R, F1 = bert_score(hyps, refs, lang="fr")
print(f"BERTScore F1: {F1.mean():.4f}")
```

10.10 Exercices

Exercice 10.1 (BLEU manuel)**.** Calculer manuellement le score BLEU-2 (bigrammes) pour la référence « le petit chat noir dort »et le candidat « le chat noir dort bien ». Détailler le calcul de p_1, p_2 et BP.

Exercice 10.2 (BERTScore vs BLEU)**.** Sur un corpus de 100 paires (référence, candidat) contenant des paraphrases, comparer les scores BLEU et BERTScore. Identifier les cas où les deux métriques divergent et expliquer pourquoi.

Exercice 10.3 (Protocole d'évaluation humaine)**.** Concevoir un protocole d'évaluation humaine pour un chatbot de service client. Définir les critères, l'échelle, le nombre d'annotateurs nécessaire, et comment mesurer l'accord inter-annotateurs.

Exercice 10.4 (Contamination des benchmarks)**.** Expliquer comment la contamination des données de test dans le corpus d'entraînement d'un LLM peut fausser les résultats sur MMLU. Proposer des méthodes pour détecter et atténuer ce problème.

Chapitre 11

Éthique, Biais et Sécurité

Responsabilité en TAL

Les modèles de langage sont déployés à grande échelle et influencent des décisions conséquentes : recrutement, justice, santé, éducation. Ils reproduisent et amplifient les biais présents dans leurs données d'entraînement. Comprendre, mesurer et atténuer ces biais est une responsabilité fondamentale du praticien en TAL.

11.1 Biais dans les plongements de mots

Définition 11.1 (Biais dans les embeddings)**.** Les plongements de mots capturent les régularités statistiques du corpus, y compris les stéréotypes sociaux. Par exemple, dans Word2Vec entraîné sur Google News :

$$\vec{\text{homme}} - \vec{\text{femme}} \approx \vec{\text{informaticien}} - \vec{\text{ménagère}}$$

Théorème 11.2 (WEAT — Word Embedding Association Test)**.** *Le test WEAT (Caliskan et al., 2017) mesure l'association entre deux ensembles de mots cibles X, Y et deux ensembles d'attributs A, B :*

$$d(X, Y, A, B) = \sum_{x \in X} s(x, A, B) - \sum_{y \in Y} s(y, A, B)$$

où $s(w, A, B) = \frac{1}{|A|} \sum_{a \in A} \cos(w, a) - \frac{1}{|B|} \sum_{b \in B} \cos(w, b)$. La significativité statistique est évaluée par un test de permutation.

Exemple 11.3 (Biais de genre)**.** En appliquant WEAT avec $X = \{\text{math, science, ingénierie}\}$, $Y = \{\text{arts, littérature, humanités}\}$, $A =$ noms masculins, $B =$ noms féminins, on observe une association significative entre sciences et masculin, reproduisant un stéréotype de genre.

11.2 Biais dans les modèles de langue

Définition 11.4 (Biais des LLM)**.** Les LLM exhibent des biais à plusieurs niveaux :

1. **Représentationnel** : certains groupes sont sous-représentés ou stéréotypés.

2. **Allocatif** : le modèle performe mieux pour certains groupes (ex. : résumés de CV favorisant certains prénoms).
3. **Génératif** : production de contenu toxique, discriminatoire ou inexact concernant certains groupes.

Proposition 11.5 (Sources de biais).

- **Données d'entraînement** : le web contient des contenus biaisés, sexistes, racistes.
- **Annotation** : les annotateurs apportent leurs propres biais culturels et linguistiques.
- **Objectif d'entraînement** : maximiser la vraisemblance reproduit les patterns dominants.
- **Déploiement** : les biais de feedback (les utilisateurs interagissent plus avec certains contenus) amplifient les biais existants.

11.3 Métriques d'équité

Définition 11.6 (Égalité des chances (Equalized Odds)). Un classifieur satisfait l'**égalité des chances** si sa performance est identique pour tous les groupes démographiques g :

$$P(\hat{Y} = 1|Y = y, G = g) = P(\hat{Y} = 1|Y = y) \quad \forall g, \forall y \in \{0, 1\}$$

Définition 11.7 (Parité démographique). La **parité démographique** exige que la décision soit indépendante du groupe :

$$P(\hat{Y} = 1|G = g) = P(\hat{Y} = 1) \quad \forall g$$

Incompatibilité des critères

Il est généralement impossible de satisfaire simultanément la parité démographique, l'égalité des chances et la calibration (théorème d'impossibilité de Chouldechova, 2017, et Kleinberg et al., 2016). Le choix du critère dépend du contexte applicatif.

Définition 11.8 (Biais de toxicité). Le **biais de toxicité** mesure la propension d'un modèle à générer du contenu toxique (insultes, menaces, discours haineux). Le score RealToxicityPrompts (Gehman et al., 2020) mesure la probabilité de générer du contenu toxique à partir d'amorces neutres.

11.4 Techniques de débiaisage

Définition 11.9 (Débiaisage des embeddings). La méthode de Bolukbasi et al. (2016) identifie la direction de biais $\mathbf{b}$ (ex. : direction homme-femme) et projette les embeddings pour éliminer cette composante :

$$\mathbf{w}_{\text{debias}} = \mathbf{w} - (\mathbf{w} \cdot \mathbf{b})\mathbf{b}$$

pour les mots neutres (non définitionnellement genreés).

Remarque 11.10. Le débiaisage géométrique des embeddings a été critiqué : Gonen et Gold (2019) montrent que les biais persistent dans la structure des voisinages même après projection. Des approches plus récentes interviennent au niveau des données ou de l'entraînement.

Définition 11.11 (Stratégies de débiaisage pour les LLM)**.**

1. **Curation des données** : filtrage des contenus toxiques, équilibrage des représentations.
2. **Instruction tuning** : fine-tuning avec des instructions explicites d'équité.
3. **RLHF/DPO** : entraîner le modèle à préférer les réponses non biaisées via feedback humain.
4. **Prompting** : instructions système spécifiant un comportement équitable.
5. **Décodage contrôlé** : modifier les probabilités de génération pour réduire la toxicité.

11.5 Détection de toxicité

Définition 11.12 (Classifieur de toxicité)**.** Un **classifieur de toxicité** est un modèle supervisé entraîné à détecter le contenu offensant, haineux ou dangereux. Exemples : Perspective API (Google), OpenAI Moderation API.

Proposition 11.13 (Défis de la détection de toxicité)**.**

1. **Définition subjective** : la toxicité dépend du contexte, de la culture et de l'intention.
2. **Biais du classifieur** : les modèles de toxicité eux-mêmes peuvent être biaisés (ex. : classifier le dialecte afro-américain comme plus toxique).
3. **Évasion** : les utilisateurs contournent les filtres par des reformulations créatives.
4. **Surtoxicité** : un filtrage trop agressif censure du contenu légitime.

11.6 Impact environnemental

Définition 11.14 (Coût carbone de l'entraînement)**.** L'entraînement d'un grand modèle de langue consomme une énergie considérable. Strubell et al. (2019) estiment l'empreinte carbone de l'entraînement d'un Transformer à ∼284 tonnes de CO_2 (comparable à 5 fois la vie d'une voiture américaine).

Estimation du coût carbone

$$CO_2 \approx \text{PUE} \times \text{kWh} \times \text{intensité carbone (g } CO_2\text{/kWh)}$$

où PUE (Power Usage Effectiveness) ≈ 1.1–1.4 pour les datacenters modernes.

Proposition 11.15 (Stratégies de réduction)**.**

- **Modèles plus petits** : distillation, pruning, quantification.
- **Entraînement efficient** : mixed precision, gradient checkpointing, architecture search.

- **Localisation** : entraîner dans des régions à électricité décarbonée.
- **Réutilisation** : partager les modèles pré-entraînés (HuggingFace Hub).

11.7 Pratiques responsables en IA

Définition 11.16 (Model Cards)**.** Une **model card** (Mitchell et al., 2019) est une fiche documentant un modèle : usage prévu, limitations, biais connus, métriques d'évaluation, données d'entraînement, et considérations éthiques.

Définition 11.17 (Datasheets for Datasets)**.** Les **datasheets** (Gebru et al., 2021) documentent les jeux de données : motivation, composition, processus de collecte, prétraitement, utilisations recommandées, et distributions démographiques.

Remarque 11.18. Les réglementations évoluent rapidement :

- **AI Act** (Union Européenne, 2024) : classification des systèmes d'IA par niveau de risque, obligations de transparence.
- **Executive Order on AI** (États-Unis, 2023) : exigences de sécurité et de tests pour les modèles de fondation.
- Les entreprises adoptent des *Responsible AI frameworks* intégrant évaluation des risques, audits réguliers et comités d'éthique.

11.8 Implémentation Python

Détection de biais dans les embeddings

```python
import numpy as np
from gensim.models import KeyedVectors

def weat_score(model, X, Y, A, B):
    """Compute WEAT effect size."""
    def s(w, A_set, B_set):
        a_sims = [model.similarity(w, a) for a in A_set
                  if a in model]
        b_sims = [model.similarity(w, b) for b in B_set
                  if b in model]
        return np.mean(a_sims) - np.mean(b_sims)

    x_scores = [s(x, A, B) for x in X if x in model]
    y_scores = [s(y, A, B) for y in Y if y in model]
    d = np.mean(x_scores) - np.mean(y_scores)
    std = np.std(x_scores + y_scores)
    return d / std if std > 0 else 0.0

# Example: gender bias in professions
X = ["programmer", "engineer", "scientist"]
Y = ["nurse", "teacher", "librarian"]
```

```
A = ["man", "male", "boy", "he"]
B = ["woman", "female", "girl", "she"]
# score = weat_score(model, X, Y, A, B)
```

Débiaisage géométrique

```
def debias_embedding(embedding, bias_direction):
    """Project out the bias direction from an embedding."""
    projection = np.dot(embedding, bias_direction) * bias_direction
    return embedding - projection

# Compute bias direction (e.g., he - she)
bias_dir = model["he"] - model["she"]
bias_dir = bias_dir / np.linalg.norm(bias_dir)

# Debias a neutral word
debiased = debias_embedding(model["programmer"], bias_dir)
```

11.9 Exercices

Exercice 11.1 (Analyse de biais)**.** Télécharger des embeddings Word2Vec ou FastText pré-entraînés. Calculer le score WEAT pour le biais de genre dans les professions. Comparer les résultats entre les embeddings entraînés sur des corpus différents (Wikipédia vs Common Crawl).

Exercice 11.2 (Débiaisage)**.** Implémenter la méthode de débiaisage de Bolukbasi et al. Vérifier que les analogies biaisées sont réduites. Tester si le débiaisage affecte la performance sur une tâche de similarité sémantique (ex. : STS-B).

Exercice 11.3 (Toxicité des LLM)**.** Utiliser l'API OpenAI Moderation (ou un classifieur Hugging Face) pour évaluer la toxicité de 100 générations d'un modèle de langue à partir d'amorces neutres. Quels types de toxicité sont les plus fréquents ?

Exercice 11.4 (Model Card)**.** Rédiger une model card complète pour un classifieur de sentiment entraîné sur des critiques de films en français. Inclure les limitations, les biais potentiels et les recommandations d'utilisation.

Bibliographie

[1] Jurafsky, D. and Martin, J.H., *Speech and Language Processing*, 3rd ed. draft, 2024.

[2] Vaswani, A. et al., Attention is All You Need, *NeurIPS*, 2017.

[3] Devlin, J. et al., BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding, *NAACL*, 2019.

[4] Brown, T.B. et al., Language Models are Few-Shot Learners, *NeurIPS*, 2020.