Université PSL – Paris Dauphine
Master Big Data & IA

2025–2026

Analyse de sentiments
TF-IDF, RNN et Attention

Rapport de projet NLP

Auteurs
Fatma CHAHED & Aziz DHIF

Enseignante
Mme Amel Tarifa

Plan

Chapitre 1

Présentation du problème

Contexte et motivation

Les avis en ligne (films, produits, restaurants) constituent une source d'information clé pour la prise de décision. Automatiser leur **analyse de sentiments** permet de résumer rapidement l'opinion globale et d'identifier des signaux forts (satisfaction, insatisfaction). Deux grandes familles de représentations sont couramment opposées : (i) les *représentations creuses* de type TF–IDF (sac de mots, n-grammes), (ii) les *représentations denses* apprises par des réseaux (embeddings + RNN/LSTM), éventuellement **pondérées par une attention** pour la mise en évidence des tokens discriminants.

Problématique

Nous cherchons à **prédire la polarité** d'un avis (*positif* / *négatif*) et à **comparer** des approches basées sur :

- une **ligne de base TF–IDF** couplée à des classifieurs linéaires (**Régression Logistique**, **SVM**) ;
- un **RNN (LSTM)** avec embeddings (appris pendant l'entraînement ou *pré-entraînés* type FastText) ;
- un **LSTM + attention (Bahdanau)**, pour améliorer la performance et l'interprétabilité (poids d'attention).

Approche et protocole d'évaluation

- **Prétraitement** : tokenisation, normalisation légère ; pour TF–IDF, choix des n-grammes et filtrage de fréquence ; pour RNN, *padding* à une longueur fixe.

- **Entraînement** : Split dédié (train/validation/test), *early stopping*, recherche légère d'hyperparamètres (C, n-grammes, `min_df/max_df`, taille LSTM, dropout).
- **Mesures** : **accuracy** et **F1**, avec rapports de classification et matrices de confusion ; pour les modèles probabilistes, courbes ROC/PR et AUC.
- **Interprétabilité** : inspection des poids TF–IDF (features les plus informatives) et **visualisation des poids d'attention** pour LSTM+Bahdanau.

Contributions

1. **Implémentation d'une baseline** TF–IDF + (Régression Logistique / SVM) et comparaison chiffrée.
2. **RNN (LSTM)** avec deux régimes d'embeddings : *appris pendant l'entraînement* vs *pré-entraînés* (FastText gelés / éventuellement ajustés).
3. **Ajout d'une attention (Bahdanau)** au-dessus du LSTM ; analyse qualitative des mots surpondérés.
4. **Synthèse des résultats** (tableaux, courbes, learning curves) et discussion des compromis performance/interprétabilité/coût.

Données et outils

- **Dataset** : Allociné (FR) `https://huggingface.co/datasets/tblard/allocine` ; splits fournis (`train`, `validation`, `test`).
- **Outils** : Python, scikit-learn (TF–IDF, LR, SVM, métriques), TensorFlow/Keras (LSTM, attention), Matplotlib (visualisation).
- **Reproductibilité** : *early stopping*, graines aléatoires, notebooks `.ipynb` fournis.

Organisation du rapport

- **Chapitre 1** : Problématique et protocole (présent chapitre).
- **Chapitre 2** : Présentation des données utilisées.
- **Chapitre 3** : Baseline TF–IDF + modèles linéaires (réglages, résultats, interprétations).
- **Chapitre 4** : RNN (LSTM) avec embeddings (appris vs pré-entraînés), comparaisons.
- **Chapitre 5** : LSTM + **attention de Bahdanau**, résultats et visualisations des poids.

Chapitre 2

Présentation des données utilisées

2.1 Chargement des données

Les données comportent un **train**, une **validation** et un **test**. On standardise les noms de colonnes et on ajoute une étiquette textuelle.

```
# 1) Charger le dataset Allocin  (FR)
ds = load_dataset("tblard/allocine")
# 2) Convertir en DataFrame pandas
df_train = pd.DataFrame(ds["train"])
df_valid = pd.DataFrame(ds["validation"])
df_test  = pd.DataFrame(ds["test"])
```

2.2 Traitement des données

Nous analysons dans cette partie, la taille du jeu de données, les valeurs manquantes, les doublons, et le mapping des étiquettes.

2.2.1 Quelques statistiques

- **Jeu de données** : le dataset Allociné contient **3 sous-ensembles** (train, validation, test), chacun avec **2 colonnes**.
- **Tailles** :
  1. Entraînement (train) : $160\,000 \times 2$
  2. Validation (validation) : $20\,000 \times 2$
  3. Test (test) : $20\,000 \times 2$

- **Colonnes** :
  - `text` : avis (film, produit, restaurant).
  - `label` : étiquette binaire initiale $\in \{negative, positif\}$.
- **Répartition des données d'entrainement :**

```
label
1    80334
0    79127
Name: count, dtype: int64
```

→ Les données sont considérées **équilibrées**.

2.2.2 Encodage de la colonne `label`

Nous allons convertir les labels textuels en labels numériques.

```python
# Mapper les labels entiers vers des libell s texte
label_map = {0: "negative", 1: "positive"}
for df in (df_train, df_valid, df_test):
    df["label_text"] = df["label"].map(label_map)
```

Résultat du mapping (`label_text` → `label`) :

	text	label	label_text
0	Si vous cherchez du cinéma abrutissant à tous ...	0	negative
1	Trash, re-trash et re-re-trash...! Une horreur...	0	negative
2	Et si, dans les 5 premières minutes du film, l...	0	negative
3	Mon dieu ! Quelle métaphore filée ! Je suis ab...	0	negative
4	Premier film de la saga Kozure Okami, "Le Sabr...	1	positive
5	L'amnésie est un thème en or pour susciter le ...	0	negative
6	Tout commence comme une comédie légère avant u...	1	positive
7	un excellent film qui merite ses quatre étoile...	1	positive
8	Deuxième long métrage de Pasolini, Mamma Roma ...	1	positive
9	Créateur de la célèbre série télévisée Kaamelo...	0	negative

2.2.3 Gestion des doublons

Nous allons faire la vérification dans les données d'entrainement :

```
# Le nombre de lignes duppliqu es
print(df_train.duplicated(subset=["label", "text"]).sum())
# Suppression
df_train = df_train.drop_duplicates(subset="text").reset_index(drop=
    True)
```

Résultat de suppression dans les données d'entrainement : (160 000 → 159 461)
Le nombre de lignes dupliquées est : **534**

2.2.4 Gestion des valeurs manquantes

```
# valeurs manquantes
print(df_train["label"].isna().sum())
```

Alors, il n'existe pas de valeurs manquantes.

Chapitre 3

Première partie : TF-IDF

3.1 Introduction

L'approche **TF-IDF** (Term Frequency–Inverse Document Frequency) transforme chaque avis en un vecteur qui pondère l'importance des mots (et éventuellement des n-grammes). Cette représentation *bag-of-words* ignore l'ordre exact des mots mais offre :

- une **base robuste et interprétable** : poids des termes, inspection des features ;
- un **apprentissage rapide** avec des modèles linéaires (Régression Logistique, SVM) ;
- une **forte ligne de base** pour comparer ensuite des modèles séquentiels.

3.2 Importation des bibliothèques

- Chargement des données (Hugging Face Datasets)

```
from datasets import load_dataset
```

- Prétraitement texte / Pipeline

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
```

- Modèles (baseline)

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier
```

- Évaluation

```
from sklearn.metrics import (
  accuracy_score, f1_score, precision_score, recall_score,
  classification_report, confusion_matrix, precision_recall_curve,
     roc_curve, auc, average_precision_score )
```

- Analyse / Visualisation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

3.3 Prédéfinition des fonctions

Dans cette partie, nous présentons les fonctions utilitaires retenues pour : l'évaluation des modèles,la visualisation des résultats et des courbes, la comparaison systématique de plusieurs architectures.

Remarque

Nous présentons ici uniquement un extrait du code. Le code complet est disponible dans le fichier `.ipynb`.

- **Fonction d'évaluation :** Calculer l'accuracy, f1-score, la précision et le recall

```
def evaluate_and_report(y_true, y_pred, title=""):
    acc = accuracy_score(y_true, y_pred)
    f1  = f1_score(y_true, y_pred)
    p   = precision_score(y_true, y_pred)
    r   = recall_score(y_true, y_pred)
    ...
```

- **Fonction** de visualisation de **la matrice de confusion** :

```
def plot_confusion(y_true, y_pred, labels=("neg","pos"), title="
   Confusion Matrix"):
    cm = confusion_matrix(y_true, y_pred, labels=[0,1])
    fig, ax = plt.subplots(figsize=(4.5,4))
    im = ax.imshow(cm, interpolation="nearest")
    ax.figure.colorbar(im, ax=ax)
    ...
```

- **Fonction** de visualisation de la **courbe ROC**

```
def plot_pr_roc(pipe, X_test, y_test, title_prefix=""):
...
```

- **Fonction** de visualisation de la **courbe d'apprentissage**

```
def plot_learning_curve(pipe, X_train, y_train, cv_splits=5, title=
   "Learning curve"):
    cv = StratifiedKFold(n_splits=cv_splits, shuffle=True,
       random_state=42)
    fig, ax = plt.subplots(figsize=(5.8,4.2))
    ...
```

3.4 Construire et tester les modèles

Dans cette section, nous allons nous limiter à la construction de trois modèles avec des paramètres différents :

1. **Régression logistique (LR)**
   - Sans régularisation
   - Avec régularisation **L2 (Ridge)** et **L1 (Lasso)**
2. **Machine à vecteurs de support (SVM)**
3. ***k* plus proches voisins (k-NN)**

3.4.1 Régression Logistique (sans régularisation)

1. Construction du pipline

```
pipe_lr = Pipeline([
 ("tfidf", TfidfVectorizer(
  lowercase=True,
  strip_accents="unicode",
  ngram_range=(1,2),  # unigram+bigram
  min_df=2,
  max_df=0.98,
  sublinear_tf=True)),
 ("clf", LogisticRegression(max_iter=200, C=2.0, n_jobs=None))
])
```

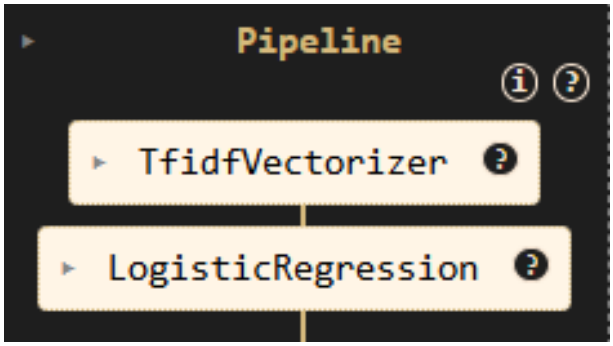


Figure 3.1 – Pipeline du modèle

2. Entrainer le modèle

```
pipe_lr.fit(df_train["text"], df_train["label"])
```

3. Prédiction des données test

```
pred_lr = pipe_lr.predict(df_test["text"])
```

4. Évaluer le modèle

```
res_lr = evaluate_and_report(df_test["label"], pred_lr, title="
    TF-IDF + LogisticRegression")
plot_confusion(df_test["label"], pred_lr, title="CM - TFIDF+LR")
```

5. Résulats :

```
=== TF-IDF + LogisticRegression ===
Accuracy: 0.9426 | F1: 0.9405 | Precision: 0.9358 | Recall: 0.9453

Classification report:
              precision    recall  f1-score   support

           0     0.9491    0.9402    0.9446     10408
           1     0.9358    0.9453    0.9405      9592

    accuracy                         0.9426     20000
   macro avg     0.9424    0.9428    0.9426     20000
weighted avg     0.9427    0.9426    0.9427     20000
```

⇒ Le modèle Logistic Regression donne de **bonnes performnaces** en termes d'accuracy, de recall et de precision.

3.4.2 Ridge

Comparons à une régression linéaire simple, nous allons simplement modifier l'étape "clf" (classifieur) afin de prendre en compte la régularisation Ridge.

```
pipe_lr = Pipeline([
    ("tfidf", TfidfVectorizer(...)),
    ("clf", LogisticRegression(
        solver="liblinear",   # solide pour texte binaire
        penalty="l2",          # Ridge
        C=2.0,                # regularisation un peu plus souple
        class_weight="balanced",  # retire si classes parfaitement
           equilibrees
        max_iter=5000,
        n_jobs=-1      # utilise par liblinear si multiclasses
    ))
])
```

```
=== TF-IDF + LogisticRegression (tuned defaults) ===
Accuracy: 0.9433 | F1: 0.9411 | Precision: 0.9379 | Recall: 0.9443

Classification report:
              precision    recall  f1-score   support

           0     0.9484    0.9424    0.9453     10408
           1     0.9379    0.9443    0.9411      9592

    accuracy                         0.9433     20000
   macro avg     0.9431    0.9433    0.9432     20000
weighted avg     0.9433    0.9433    0.9433     20000
```

⇒ Comparée à la régression logistique simple (accuracy ≈ 0,942), la régularisation Ridge donne des résultats légèrement meilleurs (accuracy ≈ 0,943), mais l'écart est négligeable.

3.4.3 Lasso

Comparons à une régression linéaire simple, nous allons simplement modifier l'étape clf (classifieur) afin de prendre en compte la régularisation Lasso.

```
pipe_lr = Pipeline([
    ("tfidf", TfidfVectorizer(...)),
    ("clf", LogisticRegression(
        solver="liblinear",   # requis pour L1 (ou 'saga')
        penalty="l1",         # LASSO
        C=2.0,
        class_weight="balanced",  # enleve si classes equilibrees
        max_iter=5000,
        n_jobs=-1))  ])
```

```
pipe_lr.fit(df_train["text"], df_train["label"])
pred_lr = pipe_lr.predict(df_test["text"])
res_lr = evaluate_and_report(df_test["label"], pred_lr, title="TF-IDF
    + LogisticRegression (L1 - Lasso)")
```

```
=== TF-IDF + LogisticRegression (L1 - Lasso) ===
Accuracy: 0.9413 | F1: 0.9391 | Precision: 0.9351 | Recall: 0.9431

Classification report:
              precision    recall  f1-score   support

           0     0.9471    0.9397    0.9434     10408
           1     0.9351    0.9431    0.9391      9592

    accuracy                         0.9413     20000
   macro avg     0.9411    0.9414    0.9412     20000
weighted avg     0.9413    0.9413    0.9413     20000
```

⇒ La régularisation Lasso (accuracy ≈ 0,941) n'améliorent pas les résultats de la régression logistique (accuracy ≈ 0,942).

3.4.4 SVM

Nous utilisions la fonction LinearSVC pour ce modèle.

```
pipe_svm = Pipeline([
    ("tfidf", TfidfVectorizer(...)),
    ("clf", LinearSVC(C=1.0))
])
pipe_svm.fit(df_train["text"], df_train["label"])
pred_svm = pipe_svm.predict(df_test["text"])
res_svm = evaluate_and_report(df_test["label"], pred_svm, title="TF-
   IDF + LinearSVM")
```

```
=== TF-IDF + LinearSVM ===
Accuracy: 0.9455 | F1: 0.9434 | Precision: 0.9388 | Recall: 0.9481

Classification report:
              precision    recall  f1-score   support

           0     0.9517    0.9430    0.9473     10408
           1     0.9388    0.9481    0.9434      9592

    accuracy                         0.9455     20000
   macro avg     0.9452    0.9456    0.9454     20000
weighted avg     0.9455    0.9455    0.9455     20000
```

⇒ Le modèle SVM est le meilleur en termes de performance (accuracy ≈ 0,955), recall et précision en comparaison avec la régression logistique ((accuracy ≈ 0,945)

3.4.5 KNN

Nous utilisions la fonction KNeighborsClassifier pour ce modèle avec les hyper-paramètres :

- la distance **cosinus** : recommandé en texte
- le nombre des **k** plus proches voisins : k=5

```
pipe_knn = Pipeline([
    ("tfidf", TfidfVectorizer(...)),
    ("clf", KNeighborsClassifier(
        n_neighbors=5,        # k de d part
        metric="cosine",      # distance cosinus
        algorithm="brute",    # n cessaire pour 'cosine'
        weights="distance",
        n_jobs=-1
    ))
])
pipe_knn.fit(df_train["text"], df_train["label"])
pred_knn = pipe_knn.predict(df_test["text"])
res_knn = evaluate_and_report(df_test["label"], pred_knn, title="TF-
   IDF + KNN (cosine)")
```

```
=== TF-IDF + KNN (cosine) ===
Accuracy: 0.8316 | F1: 0.8289 | Precision: 0.8082 | Recall: 0.8508

Classification report:
              precision    recall  f1-score   support

           0     0.8555    0.8139    0.8342     10408
           1     0.8082    0.8508    0.8289      9592

    accuracy                         0.8316     20000
   macro avg     0.8318    0.8324    0.8316     20000
weighted avg     0.8328    0.8316    0.8317     20000
```

⇒ KNN (accuracy ≈ 0,83) est nettement moins bon que tes modèles linéaires (accuracy ≈ 0,95).

En texte TF-IDF (très haute dimension, très clairsemé), KNN souffre de :

- temps d'exécution très important
- malédiction de la dimension → toutes les distances se ressemblent ;
- mauvaise pertinence des distances (si pas cosinus) ;
- coût d'inférence élevé (comparaison à tous les docs).

3.5 Améliorations possibles

- **Augmenter la taille des données :**
  Nous allons entrainer le meilleure modèle (SVM) avec les données d'entrainement et de validation. Nous pouvons par la suite l'entrainer sur la totalité des données (avec les données test) :

  1. Concaténer les données train + valid

```
df_tv = pd.concat([df_train, df_valid], ignore_index=True)
```

  2. Redéfinir le pipeline SVM

```
pipe_svm = Pipeline([
    ("tfidf", TfidfVectorizer(
        lowercase=True, strip_accents="unicode",
        ngram_range=(1,2), min_df=2, max_df=0.98, sublinear_tf=
           True
    )),
    ("clf", LinearSVC(C=1.0))
])
```

  3. Entraîner sur train + valid

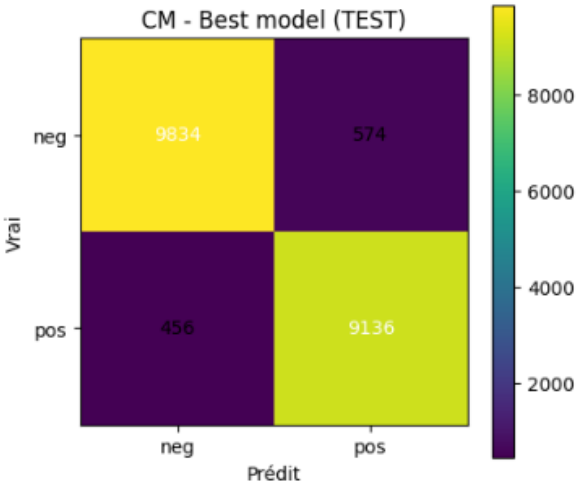
```
pipe_svm.fit(df_tv["text"], df_tv["label"])
```

```
0.9485
```

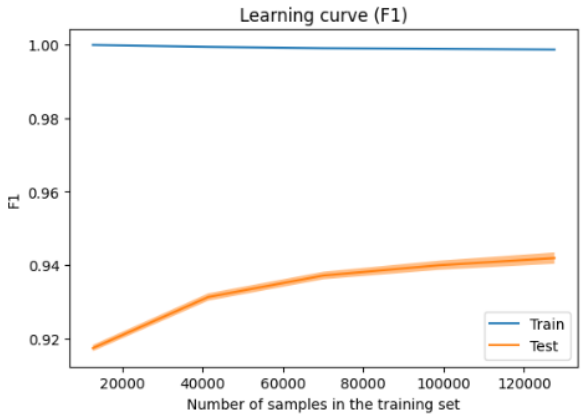
⇒ En ajoutant 20 000 exemples, l'accuracy passe de 0,9455 à 0,9485, soit un gain absolu de +0,0030 (= +0,30 point de pourcentage) et un gain relatif d'environ $\frac{0{,}9485-0{,}9455}{0{,}9455} \approx 0{,}317\%$. L'amélioration est donc **faible** au regard du coût/complexité d'augmenter les données.

3.6 Visualiser les résultats d'entraînement

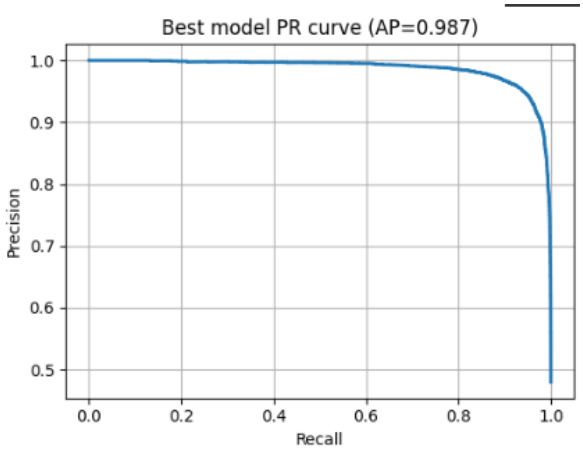
Courbe PR (AP ≈ 0,986). Sur un jeu *quasi équilibré*, une baseline aléatoire tournerait autour de 0,5 d'AP. Ici, la courbe reste très proche de la précision = 1 pour une large plage de rappels, et ne chute qu'à l'approche de Rec → 1, où l'on accepte davantage de *faux positifs* pour capter les derniers vrais positifs. Cela indique :



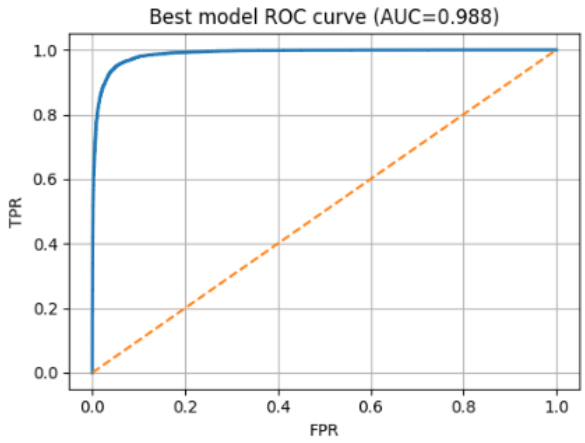
Matrice de confusion



Courbes d'entraînement



Courbe ROC



Courbe PR / AUC

- un modèle très **précis** dès que le seuil n'est pas trop bas ;
- une **dégradation contrôlée** de la précision uniquement en zone de rappel extrême.

Courbe ROC (AUC $\approx$ 0,987). Un ROC–AUC de $\sim$ 0,987 signifie qu'en tirant au hasard un exemple positif et un négatif, le classifieur (ici SVM) classe le positif au-dessus du négatif environ **98,7%** du temps. C'est un **pouvoir de séparation** très élevé et cohérent avec l'AP.

Courbes d'apprentissage. La courbe d'apprentissage montre des performances **proches entre train et validation/test** avec des niveaux élevés : pas de surapprentissage notable. *Bias/variance* raisonnable : ajouter plus de données devrait encore lisser la variance ; l'AP/ROC déjà élevés indiquent un **faible biais**.

3.7 Conclusion

Avec une représentation TF–IDF (uni+bi-grammes) et des modèles linéaires, nous obtenons une ligne de base solide. Parmi les classifieurs testés, **le SVM linéaire** ressort comme le plus performant en accuracy, devant la régression logistique et ses variantes L2/L1 ; **KNN** reste moins compétitif dans cet espace haute dimension.

Table 3.1 – Accuracy (jeu de test) des modèles TF–IDF

Modèle	**Accuracy**
Régression logistique (LR)	0.9426
Ridge (Logistique + L2)	0.9433
Lasso (Logistique + L1)	0.9413
SVM linéaire	**0.9455**
KNN	0.8316

L'écart entre **Ridge** et **LR** est minime (gain $\approx 0{,}1$ point), tandis que **SVM** apporte le meilleur compromis biais/variance sur TF–IDF.
Les gains restants proviennent surtout du **tuning** (C, n-grammes, min_df/max_df) et d'une sélection d'hyperparamètres plus fine, avant de passer aux modèles séquentiels (RNN/LSTM).

Chapitre 4

Deuxième partie : RNN

4.1 Introduction

Contrairement à TF–IDF, qui représente les avis comme des sacs de mots indépendants, un **RNN** à base de **LSTM** exploite l'*ordre* des tokens et les *dépendances longues* (négations, portées syntaxiques, co-références).

Après **tokenisation** et **padding**, chaque mot est projeté dans un espace continu via des **embeddings** (pré-entraînés ou appris).

Le **LSTM** agrège séquentiellement l'information et produit une représentation riche du message, mieux adaptée aux structures linguistiques que TF–IDF. Nous évaluerons ce modèle sur les mêmes jeux de données.

4.2 Importation des bibliothèques

- Prétraitement du texte

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

- Entraînement

```
from tensorflow.keras.callbacks import EarlyStopping
```

- Visualisation

```
import matplotlib.pyplot as plt
```

- Modélisation

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (         # couches
    Embedding,
    LSTM,                                     # RNN LSTM
    Dense,                                    # sortie
    Dropout  )                                 # regularisation
```

Prédéfinition des fonctions

```
def plot_training_history(history, model_name="Mod le"):
    ...
    plt.subplot(1,2,1)
    plt.plot(history_dict['loss'], label='Entra nement')
    plt.plot(history_dict['val_loss'], label='Validation')
```

4.3 Tokenisation

Le **tokenizer** apprend le vocabulaire et transforme chaque avis en séquence d'indices entiers. *Ici, nous avons appris le vocabulaire sur* `df_train['text']`*, converti les avis en séquences d'indices, puis normalisé leur longueur par* ***padding/troncature*** *à 150 tokens (même traitement pour validation et test).*

```
vocab_size = 20000
max_len = 150
embedding_dim = 128

tokenizer = Tokenizer(num_words=vocab_size, oov_token="<OOV>")
tokenizer.fit_on_texts(df_train['text'])

X_train = tokenizer.texts_to_sequences(df_train['text'])
X_train = pad_sequences(X_train, maxlen=max_len, padding="post",
   truncating="post")
#... faire de meme pour X_test et X_valid
```

Récupérer les targets ('label') correspondants des 3 dataframes :

```
y_val= np.array(df_valid['label'])
y_train = np.array(df_train['label'])
y_test = np.array(df_test['label'])
```

4.4 Construire et tester les modèles

Nous allons évaluer deux types d'*embeddings* :

- **Embeddings appris pendant l'entraînement** (learned from scratch) ;
- **Embeddings pré-entraînés** (p. ex. FastText).

4.4.1 Embedding pendant l'entrainement

1. Construire le modèle

```
model_1 = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim,
              input_length=max_len),
    LSTM(128, return_sequences=False),
    Dropout(0.3),
    Dense(1, activation="sigmoid")
])
```

2. Compiler le modèle

```
model_1.compile(loss="binary_crossentropy", optimizer="adam",
   metrics=["accuracy"])
model_1.summary()
```

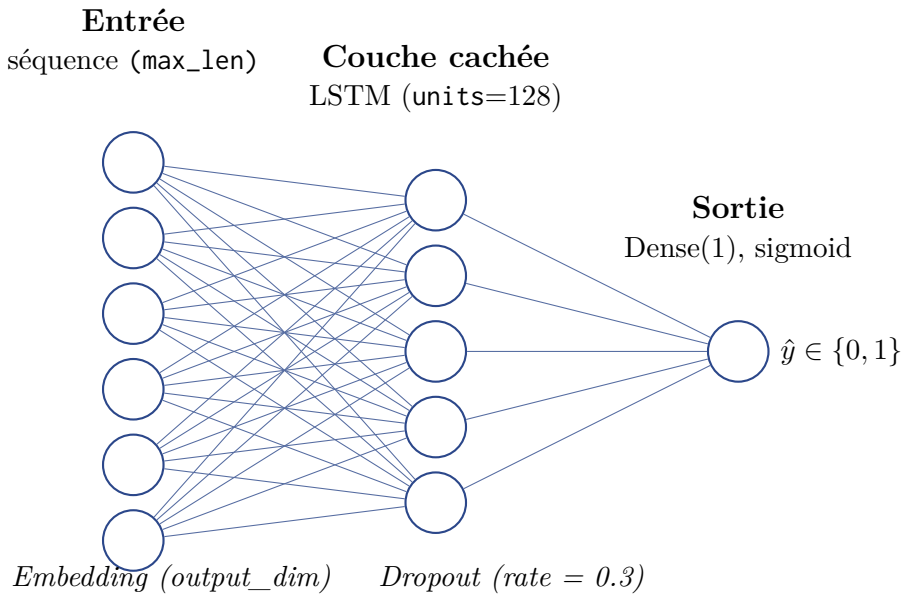


Figure 4.1 – Visualisation du réseau : Entrée → LSTM(128) → Dense(sigmoid).

3. Configurer l'arrêt anticipé (*Early Stopping*)

```
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,   # arrete si pas amelioration pendant 3 epochs
    restore_best_weights=True
)
```

4. Entraîner du modèle avec 50 epochs

```
history_1 = model_1.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
    callbacks=[early_stop],
    batch_size=128,
    verbose=1
)
```

Nous avons choisi **128** comme taille du batch car les données sont **massives**.

5. Évaluer le modèle

```
loss, acc = model_1.evaluate(X_test, y_test, verbose=0)
print(f"Accuracy sur test : {acc:.2f}")
```

```
Accuracy sur test : 0.93
```

6. Exemple d'inférence

```
#exemple
new_text = ["Ce film est g nial , un chef d'oeuvre incroyable!"]
# tokenisation
seq = tokenizer.texts_to_sequences(new_text)
padded_seq = pad_sequences(seq, maxlen=max_len, padding="post")
# prediction
prediction = model_1.predict(padded_seq)
print("Probabilit  positif :", prediction[0][0])
print("Classe pr dite :", int(prediction[0][0] > 0.5))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 248ms/step
Probabilité positif : 0.980217
Classe prédite : 1
```

4.4.2 Embedding préentrainer avec "fastText facebook 2016"

1. Charger des embeddings préentrainés

```python
import gensim.downloader as api
# Charger FastText fran ais
ft_model = api.load("fasttext-wiki-news-subwords-300")  # vecteurs 300d
```

2. **Créer la matrice d'embedding.** Nous construisons `embedding_matrix_FastText` de taille **num_words** × **embedding_dim** (par ex. 50 000 × 300).

   Chaque ligne i contient le vecteur de l'*embedding* du mot d'indice i (selon le `Tokenizer`).

```python
# Hypers
embedding_dim = 300  # dimension des vecteurs FastText
num_words = min(vocab_size, len(tokenizer.word_index) + 1)
# Creer la matrice d'embedding
embedding_matrix_FastText = np.zeros((num_words, embedding_dim))
# Remplissage depuis FastText
for word, i in tokenizer.word_index.items():
    if i >= num_words:
        continue
    w = word.lower()
    if w in ft_kv.key_to_index:  # present dans vocab FastText
        embedding_matrix_FastText[i] = ft_kv.get_vector(w)
    # sinon: on garde la valeur al atoire initialis e (OOV)
```

3. Configurer l'arrêt anticipé (*Early Stopping*)

```python
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3, # arrete si pas amelioration pendant 3 epochs
    restore_best_weights=True
)
```

4. Construire le modèle

```
model_fasText = Sequential([
    Embedding(input_dim=num_words,
              output_dim=embedding_dim,
              weights=[embedding_matrix_FastText],
              input_length=max_len,
              trainable=False),
    LSTM(128),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
    ])
```

5. Compiler le modèle

```
model_fasText.compile(loss='binary_crossentropy', optimizer='
   adam', metrics=['accuracy'])
model_fasText.summary()
```

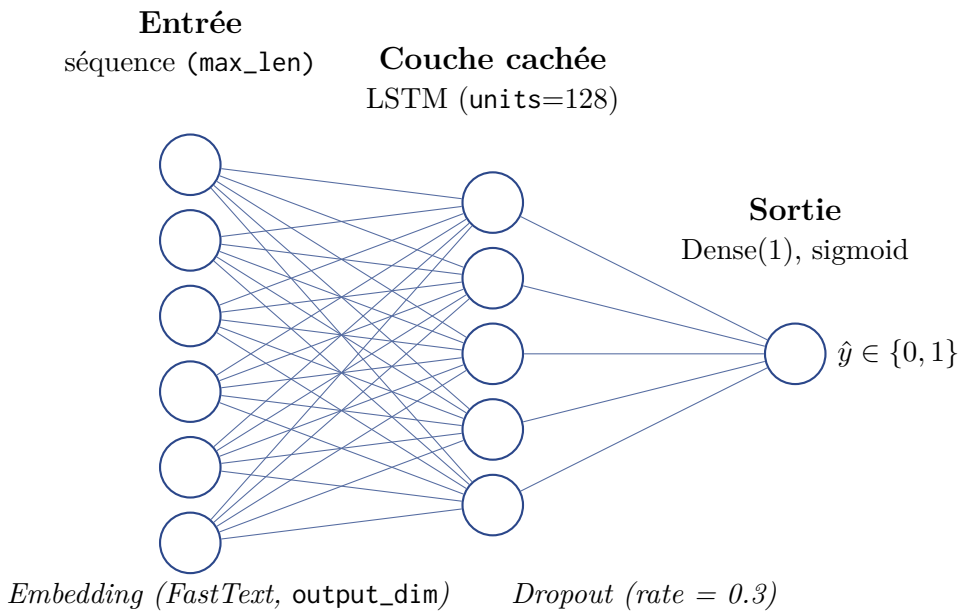


Figure 4.2 – Réseau : Entrée → **Embedding FastText (gelé)** → LSTM(128) → Dropout(0.3) → Dense(sigmoid).

6. Entraîner le modèle avec 50 epochs

```
# Entra nement
history_2=model_fasText.fit(X_train, y_train, validation_data=(
   X_val, y_val),callbacks=[early_stop], epochs=50, batch_size
   =128
)
```

7. Évaluer le modèle

```
loss, acc = model_fasText.evaluate(X_test, y_test, verbose=0)
print(f"Accuracy sur test : {acc:.2f}")
```

Nous avons choisi **128** comme taille du batch car les données sont **massives**.

```
Accuracy sur test : 0.88
```

8. Exemple d'inférence

```
#exemple
new_text = ["Ce film est g nial , un chef d'oeuvre incroyable!"]
#tokenisation
seq = tokenizer.texts_to_sequences(new_text)
padded_seq = pad_sequences(seq, maxlen=max_len, padding="post")
prediction = model_fasText.predict(padded_seq)
#pr diction
print("Probabilit  positif :", prediction[0][0])
print("Classe pr dite :", int(prediction[0][0] > 0.5))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 460ms/step
Probabilité positif : 0.9960435
Classe prédite : 1
```

4.5 Visualiser les résultats d'entraînement

```
    plot_training_history(history_1, model_name="Mod le avec
       embedding pendant l'entrainement")
    plot_training_history(history_2, model_name="Mod le avec
       pr entrainement")
```

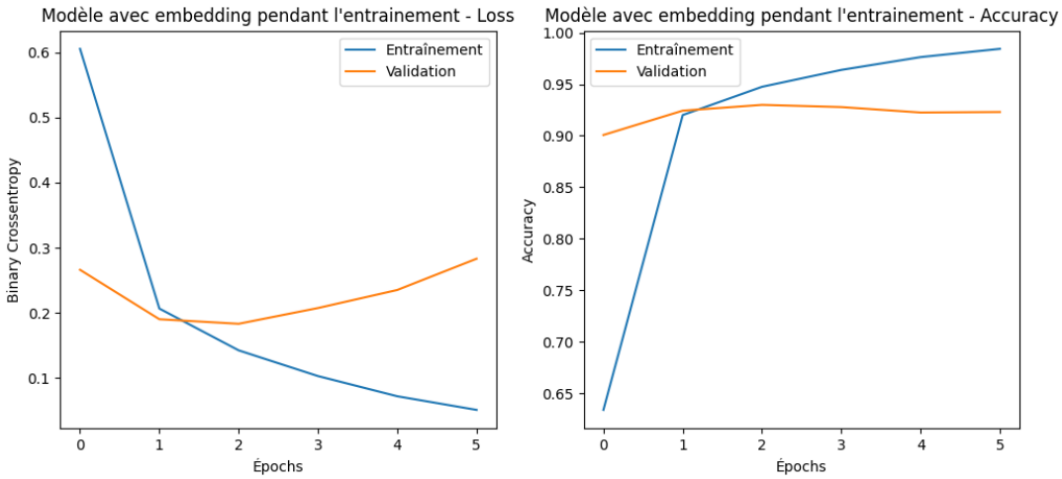


Figure 4.3 – Historique d'entraînement — *embeddings appris sur nos données.*

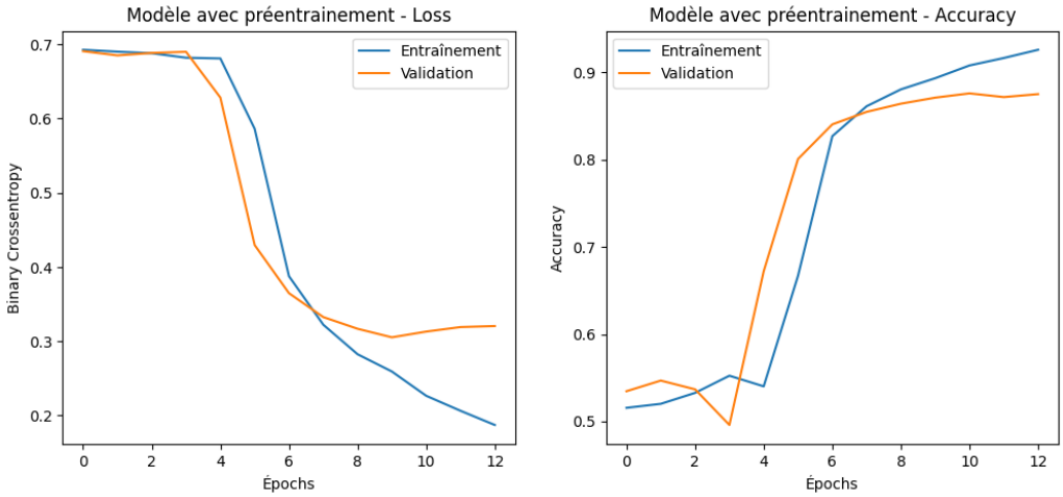


Figure 4.4 – Historique d'entraînement — *embeddings pré-entraînés.*

Figure 4.5 – Comparaison des historiques d'entraînement : embeddings appris vs pré-entraînés.

Interprétation

- Les deux modèles sont **performants** : la loss de validation baisse et l'accuracy reste élevée.
- **Embeddings appris** : convergence **plus rapide** et **plus stable**.
- **Embeddings pré-entraînés** : démarrage plus lent, mais **rattrapent** un niveau proche après quelques époques.
- Choix pratique : privilégier les **embeddings appris** si l'on a assez de données ; envisager le **pré-entraîné + fine-tuning** si les données sont limitées.

4.6 Conclusion

Dans nos expériences, le modèle **avec embeddings appris pendant l'entraînement** obtient de meilleurs scores que le modèle **initialisé par des embeddings pré-entraînés**.

Table 4.1 – Accuracy (jeu de test) des modèles LSTM

Modèle	Accuracy
Embedding pré-entrainer	0.88
Embedding pendant l'entrainement	**0.93**

- **Embeddings appris pendant l'entraînement** : meilleurs lorsque l'on dispose d'assez de données supervisées et que le domaine est spécifique. Ils s'adaptent finement au jeu de données, au prix d'un apprentissage potentiellement plus long.
- **Embeddings pré-entraînés** : utiles en *données faibles*, pour initialiser avec des connaissances lexicales générales. Leur plein potentiel apparaît souvent avec un **fine-tuning contrôlé**.

Dans notre cadre expérimental, **les embeddings appris pendant l'entraînement** sont **légèrement supérieurs** aux **embeddings pré-entraînés** *non finement ajustés*.

Chapitre 5

Troisième partie : Ajout de la couche d'attention (Bahdanau)

5.1 Introduction

Les RNN (LSTM) agrègent séquentiellement l'information, mais ne permettent pas toujours d'identifier explicitement les *mots/segments* qui portent le signal de polarité. L'**attention additive** (dite *Bahdanau*) pondère les sorties $\{h_t\}$ du LSTM par des poids α_t appris, de sorte que la représentation de phrase

$$\mathbf{c} = \sum_t \alpha_t \mathbf{h}_t \quad \text{avec} \quad \alpha_t = \text{softmax}\Big(\mathbf{v}^\top \tanh(\mathbf{W}\mathbf{h}_t + \mathbf{b})\Big)$$

mette en avant les tokens les plus discriminants pour la prédiction.

5.2 Importation des bibliothèques

- Modélisation

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Embedding, LSTM, Dropout
    , Dense, Multiply, Lambda
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping
```

- Évaluation

```
import numpy as np
from sklearn.metrics import accuracy_score, f1_score,
    classification_report, confusion_matrix
```

- Backend / utilitaires Keras (bas niveau)

```
from tensorflow.keras import backend as K  # ops bas niveau: tf.
    reduce_sum, softmax, int_shape, etc.
```

5.3 Prédéfinition de classe

Définir la classe BahdanauAttention

```
class BahdanauAttention(Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)  # pour hidden states
        self.W2 = tf.keras.layers.Dense(units)  # pour query
        self.V = tf.keras.layers.Dense(1)       # pour score
    def call(self, values, query):
         ...
        return context_vector, attention_weights
    return context
```

C'est une couche d'attention additive (Bahdanau) qui, à partir de l'état final du LSTM (query) et de toutes ses sorties temporelles (values), calcule des poids d'attention sur les pas de temps via un score appris, puis retourne la somme pondérée des sorties (vecteur contexte) ainsi que les poids eux-mêmes.

5.4 Construire et tester les modèles

5.4.1 Ajout de la couche aux modèle avec embedding pendant l'entrainement

1. Construire le modèle : LSTM + Attention de Bahdanau

```
# entr e
inputs = Input(shape=(max_len,))
embedding = Embedding(input_dim=vocab_size, output_dim=
    embedding_dim, input_length=max_len)(inputs)

# LSTM avec return_sequences=True pour l'attention et
    return_state=True pour r cup rer l' tat  final
```

```
lstm_out, state_h, state_c = LSTM(128, return_sequences=True,
    return_state=True)(embedding)

# Couche Bahdanau attention (query =  tat  final du LSTM)
attention_layer = BahdanauAttention(units=128)
att_out, att_weights = attention_layer(lstm_out, state_h)

# Dropout et Dense final
drop = Dropout(0.3)(att_out)
outputs = Dense(1, activation='sigmoid')(drop)

# Cr ation du mod le
model_bahdanau_1 = Model(inputs, outputs)
```

2. Compiler

```
model_bahdanau_1.compile(loss='binary_crossentropy', optimizer='
    adam', metrics=['accuracy'])
model_bahdanau_1.summary()
```

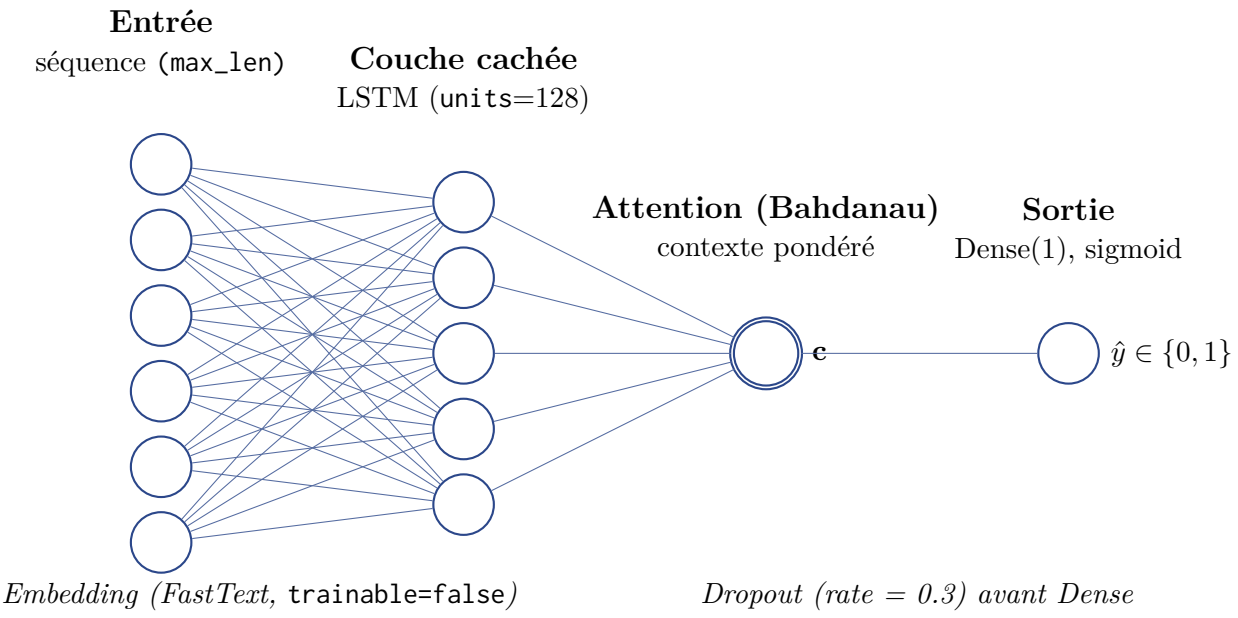


Figure 5.1 – Réseau : Entrée → **Embedding FastText (gelé)** → LSTM(128) → **Attention (Bahdanau)** → Dropout → Dense(sigmoid).

3. Configurer l'arrêt anticipé (*Early Stopping*)

```
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True )
```

4. Entraîner

```
history_bahdanau_1 = model_bahdanau_1.fit(
    X_train, y_train,
    callbacks=[early_stop],
    validation_data=(X_val, y_val),
    epochs=20,
    batch_size=128)
```

Nous avons choisi **128** comme taille du batch car les données sont **massives**.

5. Évaluer

```
loss, acc = model_bahdanau_1.evaluate(X_test, y_test, verbose=0)
print(f"Accuracy sur test : {acc:.2f}")
```

```
Accuracy sur test : 0.93
```

5.4.2 Ajout de la couche aux modèle pré-entrainer

1. Construire le modèle
   On va dans la construction du modèle, utiliser les embeddings pré-entrainés.

```
lstm_out, state_h, state_c = LSTM(128, return_sequences=True,
   return_state=True)(embedding)
attention_layer = BahdanauAttention(units=128)
att_out, att_weights = attention_layer(lstm_out, state_h)
drop = Dropout(0.3)(att_out)
outputs = Dense(1, activation='sigmoid')(drop)
model_bahdanau_2 = Model(inputs, outputs)
```

2. Compiler

```
model_bahdanau_2.compile(loss='binary_crossentropy', optimizer='
   adam', metrics=['accuracy'])
model_bahdanau_2.summary()
```

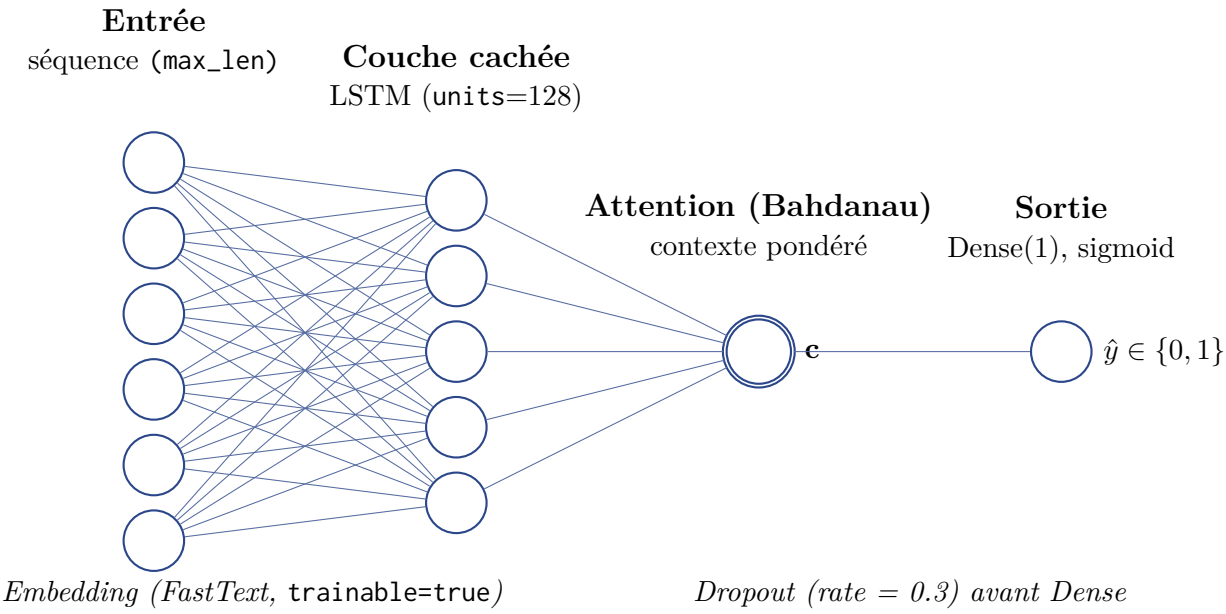


Figure 5.2 – Réseau : Entrée → **Embedding FastText (gelé)** → LSTM(128) → **Attention (Bahdanau)** → Dropout → Dense(sigmoid).

3. Entraîner

```
history_bahdanau_2 = model_bahdanau_2.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    callbacks=[early_stop],
    epochs=20,
    batch_size=128
)
```

4. Évaluer

```
loss, acc = model_bahdanau_2.evaluate(X_test, y_test, verbose=0)
print(f"Accuracy sur test : {acc:.2f}")
```

```
Accuracy sur test : 0.93
```

5.5 Visualiser les résultats d'entraînement

On trace l'évolution de la *loss* et de l'*accuracy* pour diagnostiquer le biais/variance et contrôler l'effet de l'attention.

```
plot_training_history(history_bahdanau_1, model_name="Modèle_1 et attention Bahdanau")
plot_training_history(history_bahdanau_2, model_name="Modèle_2 et attention Bahdanau")
```

1. Ajout de la couche au modèle avec *embedding* pendant l'entraînement

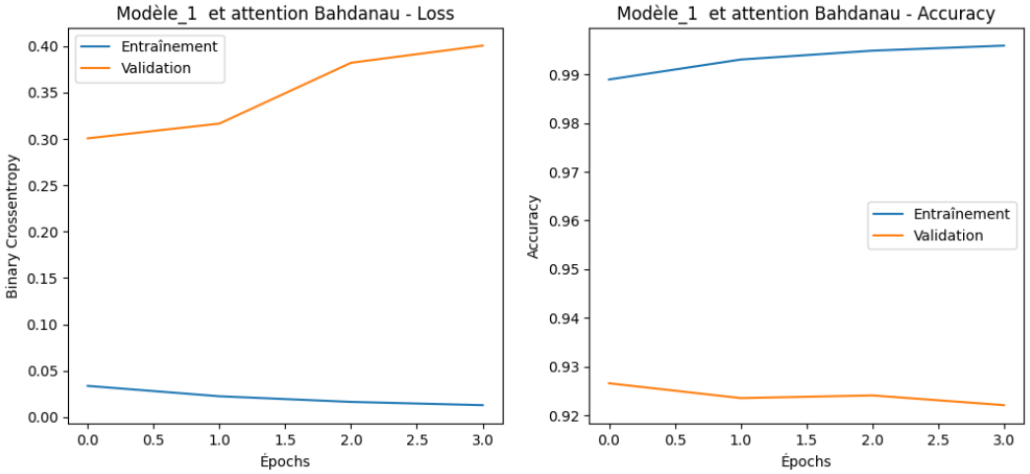


Figure 5.3 – Historique d'entraînement avec *embedding* appris.

2. Ajout de la couche au modèle pré-entraîné

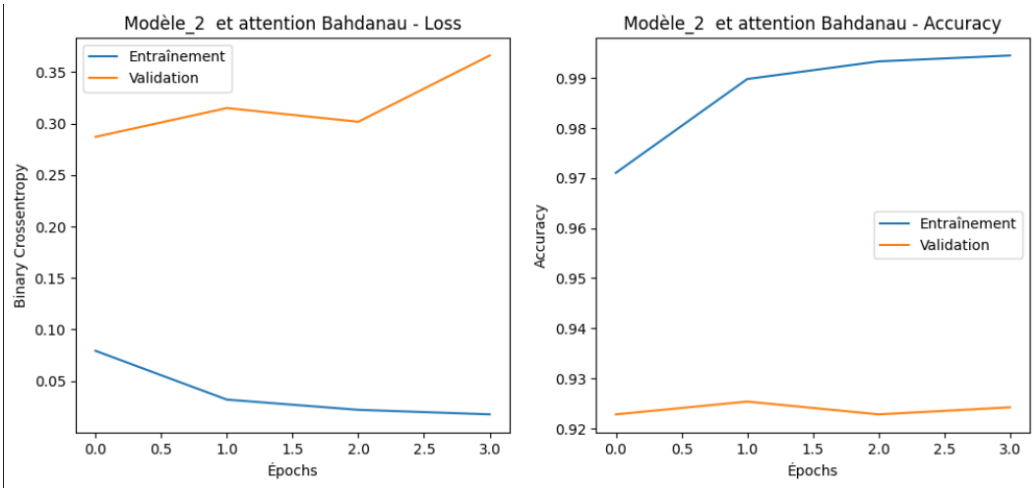


Figure 5.4 – Historique d'entraînement avec *embedding* pré-entraîné.

Interprétation

- Dans les deux cas, l'ajout de l'attention conduit à une **loss de validation faible** et une **accuracy élevée** (stabilité des courbes).
- **Embeddings appris pendant l'entraînement** : convergence **rapide** et trajectoires plus **lisses** ; l'attention capte utilement les segments pertinents.
- **Embeddings pré-entraînés** : démarrage un peu plus **lent/irrégulier**, mais la performance **rejoint un niveau proche** après quelques époques.
- Globalement, l'attention améliore l'**interprétabilité** (poids sur mots clés) et maintient une **bonne généralisation** ; le choix entre embeddings dépend surtout des données (taille/domaine) et d'un éventuel *fine-tuning*.

5.6 Conclusion

On remarque que la couche bahdannou a amélioré le modèle préentrainer et a gardé la même performance pour le modèle avec embedding pendant entrainement.

Table 5.1 – Accuracy (jeu de test) des modèles LSTM

Modèle	Accuracy
Embedding pendant l'entrainement + couche Bahdanau	**0.93**
Embeddings pré-entrainés + couche Bahdanau	**0.93**

L'ajout d'une attention additive au-dessus du LSTM :

- **améliore l'interprétabilité** grâce aux poids α_t (mise en évidence des mots clés) ;
- **peut stabiliser** l'entraînement en concentrant la représentation sur les segments pertinents ;
- apporte souvent un **gain modéré mais réel** en accuracy/F1 vis-à-vis d'un LSTM seul, surtout sur des avis longs ou des structures avec *négation / intensification*.

En pratique, l'impact dépend du **prétraitement** (tokenisation, normalisation), du **régime d'embeddings** (gelés vs *fine-tuning*), et des **hyperparamètres** (`units`, `dropout`, `learning rate`). Une analyse d'attention (extraction/visualisation des poids α_t) complète utilement la comparaison chiffrée.