**Université de Perpignan Via Domitia (UPVD)**

**UFR Sciences Exactes et Expérimentales - SEE**

**Département d'Informatique**

# Stage Master CHPS

## Thème

---

## Mise en Place d'un Cluster Kubernetes et Déploiement d'un Pipeline MLOps LSTM sur le Cluster avec Techniques d'Optimisation HPC

---

**Réalisé par**

— Bouzidi Adel

2024/2025

## Résumé

Ce projet porte sur le déploiement d'un cluster Kubernetes en local, ainsi que sur la cloudification et le développement, d'une application d'autocomplétion de texte basée sur un modèle de prédiction Long Short Term Memory (LSTM). L'application est ensuite optimisée et déployée sur le cluster Kubernetes, suivie par sa supervision (monitoring) et son évaluation à travers des opérations de benchmarking.



**Mots clés** : HPC, LSTM, Horovod, Monitoring, Optimisation, Docker, Kubernetes.

# Table des matières

# Table des figures

# Liste des abréviations

| | |
|---|---|
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| GPU | Graphics Processing Unit |
| HPC | High Performance Computing |
| LSTM | Long Short-Term Memory |
| MLOp | Machine Learning Operations |
| NAT | Network Address Translation |
| NFS | Network File System |
| NLP | Natural Language Processing |
| RNN | Recurrent Neural Network |

# Introduction

Les applications modernes exploitent de plus en plus l'intelligence artificielle, en particulier pour des fonctionnalités comme l'autocomplétion de texte. Ces systèmes reposent sur des modèles capables de prédire le mot suivant d'une phrase, et sont souvent déployés sur des clusters orchestrés par Kubernetes, parfois renforcés par des techniques issues du calcul haute performance (HPC).

Dans ce contexte, mon stage a eu pour objectif de déployer un modèle de prédiction basé sur LSTM (Long Short-Term Memory), un type de réseau de neurones récurrent performant pour le traitement de séquences.

Une partie essentielle de ce projet a été la mise en place d'un cluster Kubernetes complet en local, composé d'une machine master et de nœuds workers, pour exécuter et orchestrer l'ensemble du pipeline. Ce pipeline, déployé sur un cluster Kubernetes, automatise tout le cycle de vie du modèle. Cela inclut la préparation des données, l'entraînement du modèle (en mode simple et distribué), son évaluation, et enfin, sa mise à disposition via une interface web interactive.

Afin de mieux comprendre le système mis en place, ce rapport est structuré de manière progressive. Je commencerai par une présentation des principales technologies utilisées dans le projet, notamment LSTM, la conteneurisation avec Docker, l'orchestration avec Kubernetes, et les outils d'entraînement distribué comme Horovod. Je décrirai ensuite l'environnement de stage et l'état initial du cluster local, avant de détailler l'architecture complète du pipeline MLOps mis en œuvre. Les chapitres suivants seront consacrés aux mécanismes de déploiement, d'automatisation et d'évaluation continue.

Chapitre 1

# Vue d'Ensemble

## 1.1 Les modèles LSTM pour l'autocomplétion de texte

Les réseaux de neurones récurrents (RNN) sont une classe de réseaux de neurones adaptée au traitement de données séquentielles. Contrairement aux réseaux de neurones traditionnels, les RNN possèdent des connexions récurrentes qui leur permettent de maintenir une mémoire des informations précédentes dans la séquence. Cependant, les RNN simples souffrent du problème de disparition du gradient, limitant leur capacité à apprendre des dépendances à long terme. [7]

Les réseaux LSTM (Long Short-Term Memory) ont été introduits pour pallier cette limitation. Ils utilisent une architecture plus complexe avec des unités de mémoire (cellules) et des portes (porte d'oubli, porte d'entrée, porte de sortie) qui contrôlent le flux d'information. Ces mécanismes permettent aux LSTM de stocker et d'accéder à des informations sur de longues périodes, les rendant très efficaces pour des tâches de traitement du langage naturel (NLP) comme la modélisation du langage, la traduction automatique, et l'autocomplétion de texte. [4]

Pour l'autocomplétion, un modèle LSTM est typiquement entraîné à prédire le mot suivant dans une séquence de mots. L'architecture peut comprendre une couche d'embedding pour représenter les mots, une ou plusieurs couches LSTM pour traiter la séquence, et une couche de sortie (souvent avec une activation softmax) pour prédire la probabilité de chaque mot du vocabulaire comme étant le mot suivant. [5]

## 1.2 Conteneurisation avec Docker

Docker est une plateforme de conteneurisation qui permet d'empaqueter une application et toutes ses dépendances dans une unité appelée conteneur. Un conteneur isole l'application de l'environnement hôte, garantissant qu'elle s'exécute de manière cohérente quel que soit l'endroit où elle est déployée. [3]

Les images Docker sont des modèles en lecture seule qui servent de base pour créer des conteneurs. Elles sont construites à partir d'un fichier 'Dockerfile' qui décrit les étapes nécessaires pour assembler l'image. [2]

## 1.3 Orchestration avec Kubernetes

Kubernetes (K8s) est une plateforme open-source d'orchestration de conteneurs qui automatise le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Il offre un cadre robuste pour exécuter des systèmes distribués de manière organisée. [15]

Les principaux composants de Kubernetes utilisés dans ce projet incluent [16] [14] :

- **Pods :** La plus petite unité déployable dans Kubernetes, représentant une ou plusieurs instances de conteneurs partageant des ressources (réseau, stockage).
- **Deployments :** Gèrent le cycle de vie des Pods, permettant des mises à jour déclaratives et le scaling.
- **Services :** Fournissent une abstraction réseau stable pour accéder aux Pods (par exemple, via un LoadBalancer pour une exposition externe).
- **CronJobs :** Permettent de planifier l'exécution de tâches (Jobs) à des intervalles réguliers.
- **MPIJob (via Kubeflow) :** Une ressource personnalisée pour exécuter des applications parallèles basées sur MPI, comme celles utilisant Horovod.

## 1.4 Entraînement distribué et stockage partagé

L'entraînement des modèles de deep learning, en particulier sur de grands jeux de données, peut être très gourmand en temps et en ressources. L'entraînement distribué permet de paralléliser cette tâche sur plusieurs nœuds de calcul ou plusieurs GPUs, réduisant ainsi le temps d'entraînement. [17]

**Horovod** est un framework d'entraînement distribué open-source pour TensorFlow, Keras, PyTorch et Apache. Il simplifie la distribution de l'entraînement. Dans ce projet, Horovod est utilisé avec TensorFlow et Keras pour l'entraînement distribué du modèle LSTM. Le composant 'MPIJob' de Kubeflow est utilisé pour orchestrer le lancement des processus Horovod (un launcher et plusieurs workers) sur le cluster Kubernetes. [12]

Le **stockage partagé** est essentiel dans un environnement distribué pour que tous les composants du pipeline puissent accéder aux mêmes données, scripts, modèles et résultats. [11]

## 1.5 Entraînement d'un modèle de deep learning basé sur LSTM

L'entraînement d'un modèle de deep learning repose sur l'ajustement itératif de ses paramètres internes à partir d'un jeu de données étiqueté. Dans le cas des réseaux de neurones récurrents (RNN) tels que les LSTM (*Long Short-Term Memory*), ce processus permet d'apprendre des dépendances temporelles ou séquentielles dans les données, ce qui est particulièrement adapté aux tâches de traitement du langage naturel.

L'entraînement consiste à présenter des séquences d'entrée au réseau, à calculer les prédictions, puis à ajuster les poids internes pour minimiser une fonction de perte à l'aide d'un optimiseur. Ce processus est répété sur plusieurs itérations, appelées *époques*, afin d'améliorer progressivement la performance du modèle. [9]

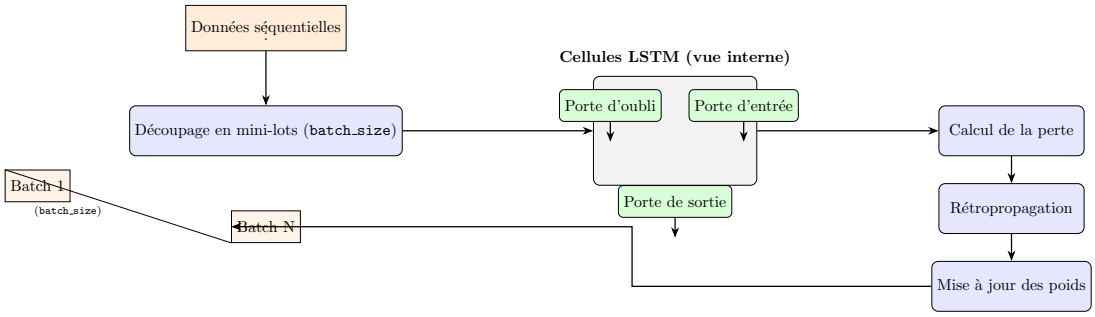


FIGURE 1.1 – Cycle d'entraînement d'un modèle LSTM avec traitement par lots, propagation interne dans la cellule, calcul de perte et mise à jour des poids.

### 1.5.1 Optimisation de la phase d'entraînement par paramétrage Fin

Dans les bibliothèques de *deep learning* comme `TensorFlow` et `Keras`, la méthode `model.fit()` permet de lancer facilement l'entraînement d'un modèle. Elle accepte plusieurs paramètres qui permettent d'adapter le déroulement de cette phase en fonction des ressources disponibles et des besoins de l'utilisateur. Par exemple, le paramètre `batch_size` détermine combien de phrases sont traités en même temps avant que le modèle ne mette à jour ses poids. Le paramètre `validation_split` permet de garder automatiquement une partie des données pour tester le modèle après chaque cycle d'apprentissage (appelé époque). Ces paramètres, s'ils sont bien choisis, peuvent améliorer l'efficacité de l'entraînement. [6]

Chapitre 2

# Réalisation et Implémentation

## 2.1 Description du cluster Kubernetes mis en place

Le premier objectif de ce projet a été la mise en place d'un cluster Kubernetes local (version 1.29.15), composé de deux nœuds : un nœud maître (Master1) et un nœud de travail (Worker1), Le Master1 et Worker1 sont des machines virtuelles crées via VirtualBox (version 7.1.8) disposant respectivement de 6.42 Go et 9.33 Go de mémoire vive, et 2 CPU pour chaque machine. L'installation de Kubernetes a été réalisée sur CentOS 9, en utilisant Docker comme moteur d'exécution des conteneurs (Container Runtime). lstm

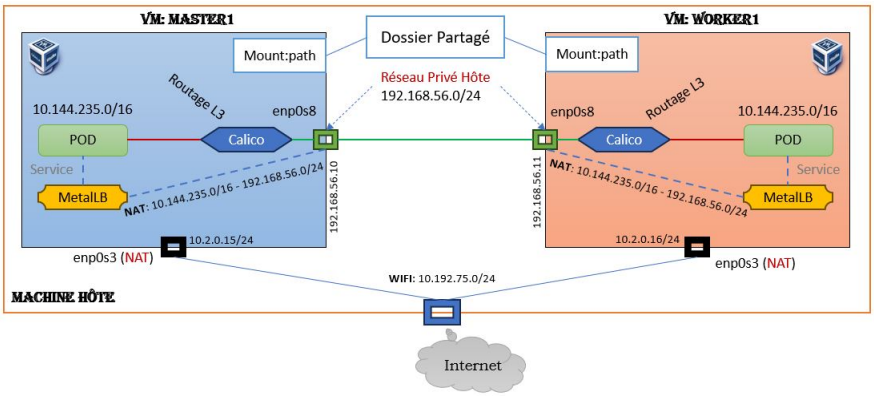


FIGURE 2.1 – Architecture globale du cluster Kubernetes

Comme illustré dans la figure 2.1, j'ai activé deux interfaces réseau pour chaque machine virtuelle. La première interface (enp0s3), configurée en mode NAT, fournit un accès à Internet afin de permettre l'installation des différents paquets nécessaires au cluster Kubernetes ainsi qu'à l'application LSTM à déployer dessus. La deuxième interface réseau (enp0s8), configurée en mode Réseau Privé Hôte, permet la communication entre les nœuds du cluster. Afin d'assurer la connectivité entre les pods situés sur différents nœuds, j'ai installé le plugin Calico. De plus, j'ai déployé le LoadBalancer MetalLB, qui permet l'accès depuis l'extérieur de la machine virtuelle aux services exposés dans les pods, en effectuant une opération de NAT entre l'adresse IP privée du pod et une adresse attribuée depuis le Réseau Privé Hôte.

## 2.2 Architecture détaillée du pipeline MLOps

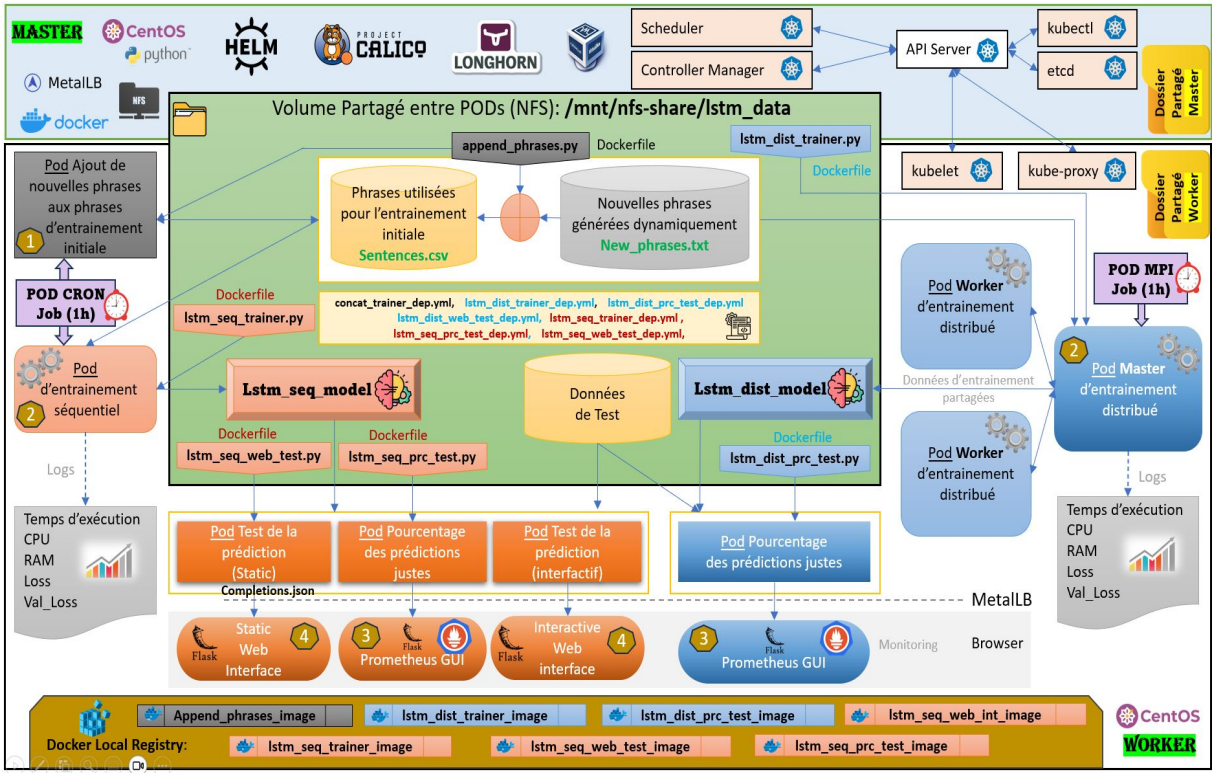


FIGURE 2.2 – Architecture détaillée du pipeline

Le système MLOps mis en place dans le cadre de ce stage permet d'automatiser et d'orchestrer l'ensemble du cycle de vie d'un modèle LSTM pour l'autocomplétion de texte, avec une mise à jour périodique des données d'entraînement et une regénération continue des modèles entrainés.

Comme illustré dans la figure 2.2, le pipeline comprend les étapes suivantes : 1) la mise à jour périodique des données d'entraînement en ajoutant régulièrement de nouvelles phrases aux données d'entraînement, ces phrases sont générées via un programme, 2) l'entraînement du modèle LSTM séquentiel et distribué, 3) l'évaluation continue des modèles LSTM entraînés (séquentiel et distribué), à travers l'exposition de métriques mesurant le pourcentage de prédictions correctes via Prometheus, permettant une visualisation comparative dans le navigateur. 4) démonstration de la prédiction via : a) une interface web statique affichant des résultats pré-calculés, b) une interface web interactive permettant à l'utilisateur de tester directement la prédiction.

Ce qui suit décrit en détail le rôle et le fonctionnement des principaux composants et modules de l'architecture illustrée.

### 2.2.1 Le volume partagé

Le système de prédiction basé sur LSTM se compose de plusieurs programmes ou modules s'exécutant séparément dans des conteneurs déployés sur différents pods. Ces modules doivent pouvoir communiquer entre eux. Par exemple, le module de prédiction a besoin d'accéder au modèle entraîné généré par le module d'entraînement. Pour établir cette communication, j'ai créé un volume de stockage partagé (/mnt/nfs-share/lstm_data) sur le nœud Master (192.168.56.10). Ce volume a ensuite été monté sur l'ensemble des pods du système de prédiction, de test et de monitoring à l'aide de NFS (Network File System). Ce volume partagé contient tous les scripts de prédiction, de test et de monitoring, ainsi que les fichiers Dockerfile utilisés pour construire leurs images respectives, et les fichiers Deployment.yaml nécessaires au déploiement de ces modules sur le cluster Kubernetes. Le volume partagé joue un rôle essentiel dans l'architecture. Il permet à l'ensemble des composants du pipeline d'accéder de manière cohérente aux mêmes fichiers. Il offre aussi une grande souplesse : on peut modifier les scripts ou les fichiers sans devoir reconstruire à chaque fois les images Docker.

### 2.2.2 Module de mise à jours de données d'entrainement (`append_phrases`)

Afin de garantir un entraînement sur des données constamment à jour, j'ai mis en place un mécanisme d'enrichissement automatique du jeu de données d'entraînement. Ce processus repose sur un script Python, `append_phrases.py`, qui utilise la bibliothèque `pandas` pour insérer régulièrement un lot de nouvelles phrases, extraites du fichier source `new_phrases.txt`, au début du fichier `sentences.csv`.

Le script est placé dans le volume partagé et conteneurisé à l'aide d'un `Dockerfile` basé sur `python:3.10-slim`, intégrant l'ensemble des dépendances nécessaires. Son exécution est planifiée de manière périodique, toutes les heures, via un `CronJob` Kubernetes défini dans le fichier `insert-cronjob.yaml`. Ce dernier déclenche régulièrement le pod `append_phrases`, lequel accède au volume partagé (NFS) et assure ainsi une mise à jour autonome et continue des données d'entraînement.

### 2.2.3 Modules d'entraînement

Le cœur du pipeline est les modules d'entraînement, conçu pour fonctionner selon deux modes distincts, chacun déployé sur un pod différent : un mode d'entraînement séquentiel classique, et un mode d'entraînement distribué. Dans ce qui suit, je présenterai d'abord le modèle LSTM choisi pour la prédiction, puis les deux modules d'entraînement, séquentiel et distribué. Il convient aussi de noter que les scripts Python d'entrainement distribué et séquentiel ont été développés en s'appuyant sur un exemple issu de la plate-forme Pluralsight

**Le modèle LSTM utilisé pour la prédiction du mot suivant**

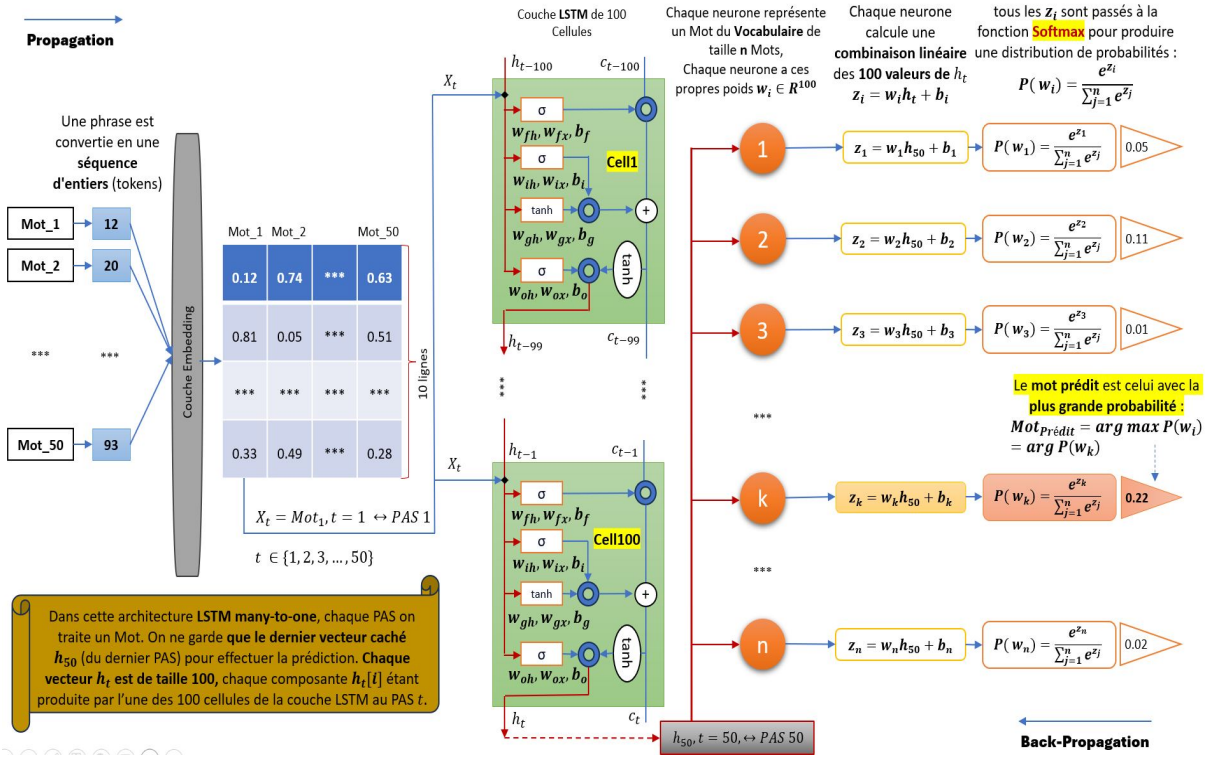


FIGURE 2.3 – Le modèle LSTM utilisé pour la prédiction du mot suivant, Basé sur [10]

Le modèle LSTM utilisé s'appuie sur une architecture classique des systèmes de génération de texte. Il s'agit d'un réseau de type `many-to-one`, dans lequel une séquence de mots en entrée (sous forme de tokens) est d'abord projetée dans un espace vectoriel au moyen d'une couche `Embedding` transformant chaque mot (représenté par un entier) en vecteur dense de dimension 10. Elle permet d'apprendre des représentations sémantiques distribuées. Cette représentation est ensuite traitée par une couche mémoire `LSTM` à long terme avec 100 unités. Elle est conçue pour capter les dépendances contextuelles dans la séquence de texte. Enfin, une couche de sortie `Dense (Softmax)` avec une activation `softmax` de taille égale au vocabulaire. Elle prédit la probabilité de chaque mot comme prochaine sortie.

Cette architecture est parfaitement adaptée à la tâche d'autocomplétion, car elle permet au modèle de tenir compte du contexte passé pour générer des prédictions pertinentes.

### Le module d'entrainement séquentiel

L'entraînement séquentiel est implémenté dans le script Python `lstm_seq_train.py` (Basé sur : [10]). Celui-ci effectue successivement le prétraitement du texte (tokenisation et *padding*), la construction du modèle LSTM à l'aide de Keras, puis son entraînement sur un seul processus. À partir d'un fichier `Dockerfile`, basé sur l'image `python:3.10-slim`, une image Docker nommée `lstm_seq_train_image` a été construite pour exécuter un pod dédié, qui lance à son tour le script `lstm_seq_train.py` depuis le volume partagé. Ce script prend en entrée les données d'entraînement disponibles sur le volume partagé, constituées à la fois des phrases initiales présentes dans le fichier `sentences.csv`, et des phrases générées dynamiquement, ajoutées périodiquement au début du même fichier à partir de `new_phrases.txt`. En sortie, `lstm_seq_train.py` génère le modèle LSTM entraîné (`lstm_seq_model`) ainsi que le tokenizer associé, et les enregistre tous deux dans le volume partagé sous forme de fichiers.

Pour automatiser l'exécution de cette tâche, un `CronJob` Kubernetes, défini dans le fichier `train-cronjob.yaml`, est configuré afin de lancer périodiquement l'entraînement. Par ailleurs, un script shell, `start_seq.sh`, permet de déclencher manuellement cette opération si nécessaire.

### Le module d'entraînement distribué (avec Horovod)

L'entraînement distribué a été implémenté dans le script Python `lstm_dist_train.py` ( Basé sur : [13, 8, 1] ) à l'aide du framework Horovod. L'entraînement distribué avec Horovod repose sur le principe du parallélisme de données, dans lequel les trois pods ou processus, comme montré dans la figure 2.2 (pod master, deux pod worker) entraînent simultanément une copie identique du modèle sur des sous-ensembles différents du jeu de données. Chaque pod exécute localement le même script d'entraînement et calcule les gradients sur sa portion de données. Horovod utilise ensuite une opération collective appelée AllReduce pour agréger et synchroniser les gradients entre tous les pods, garantissant que les poids du modèle restent identiques sur chaque instance. Ainsi, les trois pods participent activement à l'entraînement. Toutefois, pour éviter les conflits d'écriture sur le volume partagé, seul le pod master est autorisé à sauvegarder le modèle final entraîné `lstm_dist_model` ainsi que le tokenizer associé.

À partir d'un fichier `Dockerfile` basé sur l'image Horovod `lstm-horovod:latest`, une image Docker nommée `lstm_dist_train_image` a été construite pour exécuter les trois pods (un master et deux workers), qui lancent chacun le script `lstm_dist_train.py` de manière distribuée, depuis le volume partagé. Ce script prend en entrée les données d'entraînement accessibles sur le volume partagé. En sortie, `lstm_dist_train.py` produit le modèle LSTM entraîné (`lstm_dist_model`) ainsi que le tokenizer associé, qui sont tous deux enregistrés dans le volume partagé.

L'orchestration sur Kubernetes est assurée par une ressource personnalisée de Kubeflow, un `MPIJob` défini dans le fichier `lstm-horovod-mpijob.yaml`, qui déploie le pod *master* ainsi que les deux pods *worker*. Le cycle de vie de ce job distribué est en outre géré par le script shell `start_horovod_loop.sh`, chargé de déclencher périodiquement l'entraînement distribué tout en s'assurant, au préalable, qu'aucune session précédente n'est encore en cours d'exécution.

### Optimisation de la phase d'entraînement distribué via la méthode `model.fit()`

Dans le module d'entraînement distribué, une optimisation supplémentaire a été intégrée lors de la phase d'apprentissage du modèle. Elle consiste en une configuration fine de l'appel à la méthode `model.fit()` dans le script Python d'entraînement distribué. Cette configuration a pour objectif de maximiser l'utilisation des ressources disponibles — même en l'absence d'accélération GPU — en ajustant plusieurs paramètres clés : la taille des lots de données (`batch_size`), la fraction dédiée à la validation (`validation_split`), les fonctions de rappel (`callbacks`) ainsi que le niveau de verbosité (`verbose`). Le code suivant illustre l'appel correspondant tel qu'implémenté dans le script :

```
model.fit(
    X, y,
    epochs=15,
    batch_size=128,
    validation_split=0.1,
    callbacks=callbacks,
    verbose=1 if hvd.rank() == 0 else 0 )
```

Chaque paramètre joue un rôle important dans ce contexte : 1) `batch_size=128` : permet de traiter efficacement les données tout en limitant les synchronisations entre processus. 2) `validation_split=0.1` : réserve 10 % des données pour évaluer la qualité du modèle à chaque époque. 3) `callbacks` : inclut un mécanisme de sauvegarde automatique (`ModelCheckpoint`), activé uniquement pour le processus principal (rang 0 ou master), afin d'éviter tout conflit d'écriture sur le volume partagé. 4) `verbose` : limite l'affichage à un seul processus pour garder les journaux propres et lisibles.

Grâce à ce paramétrage, l'entraînement est non seulement distribué, mais également contrôlé et optimisé pour exploiter efficacement les ressources CPU dans l'environnement Kubernetes.

### Accélération GPU

L'environnement logiciel utilisé permet l'accélération matérielle sur des cartes graphiques compatibles (NVIDIA CUDA). Cette optimisation est donc activée dans le pipeline. Lors du démarrage, TensorFlow tente automatiquement de détecter les GPU disponibles pour y déléguer les calculs.

Dans le cadre de ce stage, les tests ont été effectués sur un cluster ne disposant que des ressources CPU. En conséquence, bien que le support GPU soit actif, il n'a pas été utilisé lors des exécutions. Le code reste néanmoins entièrement compatible avec une exécution accélérée sur GPU, ce qui représente une optimisation exploitable dans des contextes matériels plus puissants.

### Fonctionnement de l'dntraînement LSTM distribué via Horovod et MPIJob

Comme illustré dans la figure 2.4, l'entraînement distribué d'un modèle LSTM avec `Horovod` repose sur une stratégie de parallélisme de données, où plusieurs *pods workers et master* partagent le travail en traitant chacun un sous-ensemble différent du jeu de données. Chaque *pod* possède une copie locale du modèle et exécute les étapes classiques de l'apprentissage : propagation avant, calcul de la perte, puis

rétropropagation pour déterminer les gradients des poids.

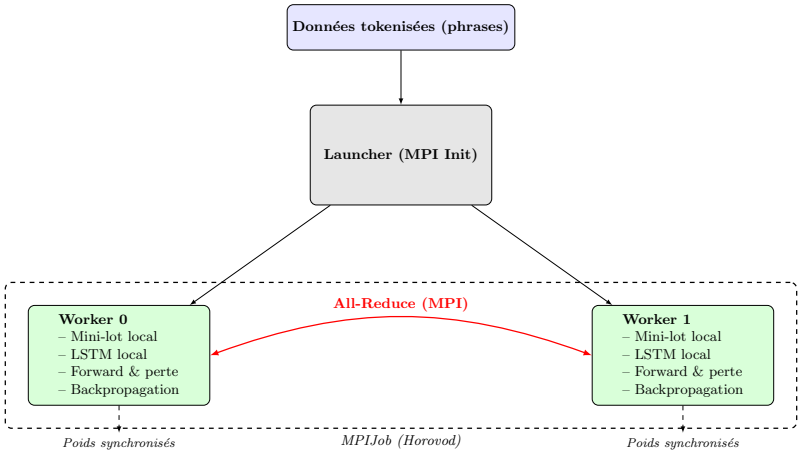


FIGURE 2.4 – Fonctionnement de l'entraînement LSTM distribué avec Horovod et MPIJob.

Une opération collective `All-Reduce`, coordonnée par Horovod via MPI, est ensuite utilisée pour agréger les gradients entre tous les *pods*. Cela permet à chacun de mettre à jour les poids du modèle de manière identique, garantissant ainsi une convergence cohérente de l'antrainement.

Ce cycle se répète pour chaque lot de données, sur plusieurs époques. L'ensemble du processus est orchestré par la ressource `MPIJob` de Kubeflow, qui déploie un *pod master* pour initier la session MPI et plusieurs *pods workers* pour exécuter le code d'entraînement en parallèle dans un cluster Kubernetes.

### 2.2.4 Module d'evaluation et de monitoring

Un élément clé du pipeline MLOps réside dans l'évaluation régulière des modèles afin de suivre leur performance au fil du temps. À cette fin, j'ai développé un composant dédié, constitué de deux services distincts, chacun chargé d'évaluer l'un des deux types de modèles générés par le système : le modèle entraîné de manière séquentielle et celui issu de l'entraînement distribué.

#### Le module d'evaluation du modèle d'entrainement séquentiel

L'évaluation du modèle entraîné de manière séquentielle est implémentée dans le script Python `lstm_seq_prc_test.py`. À partir d'un `Dockerfile` basé sur une image Flask, une image Docker nommée `lstm_seq_prc_test_image` a été construite afin de déployer un pod dédié, qui exécute le script d'évaluation depuis le volume partagé. Ce script, correspondant à une application Flask, prend en entrée les données de test ainsi que la dernière version du modèle `lstm_seq_model`. Il évalue le pourcentage de prédictions correctes par rapport à l'ensemble des données de test, et expose ce score sous forme de métriques lisibles par Prometheus.

#### Le module d'evaluation du modèle d'entrainement distribué

L'évaluation du modèle entraîné de manière distribuée est implémentée dans le script Python textttlstm_dist_prc_test.py. À partir d'un `Dockerfile` basé sur une image Flask, une image Docker

nommée `lstm_dist_prc_test_image` a été construite afin de déployer un pod dédié, qui exécute le script d'évaluation depuis le volume partagé. Ce script, correspondant à une application Flask, prend en entrée les données de test ainsi que la dernière version du modèle `lstm_dist_model`. Il évalue le pourcentage de prédictions correctes par rapport à l'ensemble des données de test, et expose ce score sous forme de métriques lisibles par Prometheus.

#### Intégration avec Prometheus

L'intégration de ces deux évaluateurs avec le système de monitoring est un aspect clé. Chaque service utilise la bibliothèque `prometheus_client` pour exposer la métrique de précision calculée, nommée `lstm_model_accuracy`, sur un endpoint HTTP `/metrics`. Pour différencier les modèles, j'ai utilisé un label Prometheus `mode` (avec les valeurs `"sequential"` ou `"distributed"`). Enfin, le serveur Prometheus est configuré, via le fichier `prometheus.yml`, pour collecter (*scraper*) périodiquement ces métriques depuis les adresses IP des services d'évaluation, permettant ainsi de visualiser et d'analyser l'évolution de la performance des modèles de prédiction.

### 2.2.5 Interfaces web statique et intéractive

Pour rendre les modèles et leurs résultats accessibles, j'ai développé deux interfaces web distinctes basées sur le framework Flask.

Le script Python `lstm_seq_web_test.py` permet d'effectuer des prédictions sur un ensemble de données de test. À partir d'un `Dockerfile` basé sur une image Flask, une image Docker nommée `lstm_seq_web_test_image` a été construite afin de déployer un pod dédié, qui exécute le script de prédiction depuis le volume partagé. Ce script prend en entrée les données de test ainsi que la dernière version du modèle `lstm_seq_model`. En sortie, il génère les résultats de prédiction dans un fichier `completions.json`, également stocké dans le volume partagé.

La première interface web, est exécutée sur un pod, cette interface lit le fichier de résultats précalculés, `completions.json`, généré après chaque entraînement séquentiel et stocké sur le volume partagé. L'interface affiche ensuite des paires de phrases d'entrée et de complétions générées, offrant un aperçu statique des capacités du modèle.

La seconde interface web offre une expérience bien plus dynamique et interactive. Plutôt que d'afficher des résultats statiques, elle charge directement le modèle LSTM entraîné séquentiellement `lstm_seq_model` ainsi que son tokenizer depuis le volume partagé. Elle expose un endpoint API qui, lorsqu'il est appelé par l'interface utilisateur, effectue une prédiction en temps réel pour suggérer le mot suivant. L'interface, développée en HTML et JavaScript, permet à l'utilisateur de saisir une phrase et de visualiser instantanément la suggestion générée par le modèle.

Les deux interfaces ont été conteneurisées avec Docker et déployées sur Kubernetes via des manifestes `Deployment` et `Service` de type `LoadBalancer`, les rendant accessibles depuis l'extérieur du cluster.

### 2.2.6 Les scripts d'automatisation et de gestion

Pour gérer la complexité du déploiement et de l'exécution du pipeline, j'ai développé une suite de `scripts bash` qui automatisent les interactions avec le cluster Kubernetes. Le script `start_evaluation.sh` s'occupe de déployer les deux services d'évaluation des modèles. Pour l'entraînement, deux scripts distincts ont été créés : `start_horovod_loop.sh` gère le cycle de vie des entraînements distribués en lançant un job `MPIJob` périodiquement, mais seulement si aucun entraînement précédent n'est encore actif ; tandis que `start_seq.sh` permet de déclencher manuellement un entraînement séquentiel, en évitant les lancements multiples. Un script principal, `start_distribue_et_sequentiel.sh` , orchestre le tout en lançant ces processus d'entraînement en parallèle, ainsi qu'un script de surveillance, `monitor_pods.sh` , qui enregistre l'état des pods à intervalles réguliers. Enfin, pour faciliter la réinitialisation de l'environnement, un script de nettoyage, `del__.sh` , a été créé pour supprimer rapidement toutes les ressources Kubernetes associées au projet.

# Chapitre 3

# Analyses et Scénario d'Exécution

## 3.1 Interfaces web - démonstration

Comme expliqué plus tôt, j'ai créé deux interfaces web différentes avec le framework `Flask`. La première interface permet d'afficher des exemples de résultats du modèle. Elle lit un fichier appelé `completions.json`, qui contient des phrases et leurs complétions générées lors d'un précédent entraînement. Ces résultats sont stockés sur le volume partagé, puis affichés simplement dans une page web, comme montré dans la figure 3.1.

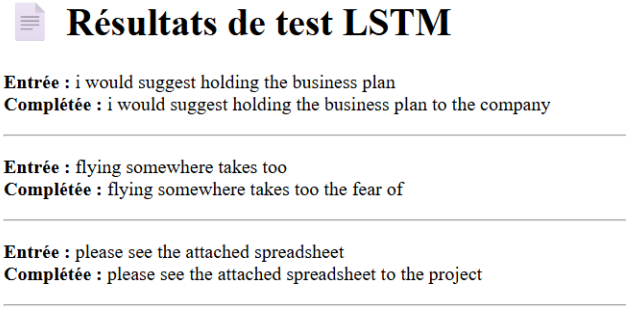


FIGURE 3.1 – Interface web statique

La deuxième interface web est plus interactive, elle charge directement le modèle LSTM entraîné (`Lstm_seq_model`) ainsi que son *tokenizer* depuis le volume partagé. Grâce à une API, elle peut générer une prédiction en temps réel dès qu'un utilisateur entre une phrase de 3 mots, comme montré dans la figure 3.2.


FIGURE 3.2 – Interface web intéractive

## 3.2 Monitoring de la précision de prédiction via Prometheus

Comme indiqué précédemment, l'évaluation des modèles LSTM est effectuée en continu par les composants `lstm_seq_prc_test` et `lstm_dist_prc_test`. Ces évaluateurs calculent la précision des modèles sur un ensemble de phrases de test, où une prédiction est considérée comme correcte si le mot suivant généré par le modèle figure parmi une liste de mots attendus. Ainsi, les valeurs produites par ces deux modules représentent le pourcentage de prédictions correctes parmi l'ensemble des cas testés. La figure 3.3 illustre le suivi de cette métrique, assuré par Prometheus, pour les modèles distribués et séquentiels.


FIGURE 3.3 – Monitoring de la précision de prédiction via Prometheus

## 3.3 Analyse comparative : entraînement séquentiel vesus distribué

Rappelons que dans ce projet, deux approches pour l'entraînement des modèles LSTM ont été implémentée : 1) **Entraînement Séquentiel :** exécuté sur un seul pod, orchestré par un `CronJob` ou lancé manuellement, 2) **Entraînement Distribué :** utilisant Horovod et un `MPIJob` avec un launcher et deux workers, orchestré par un script de boucle.

Lors de l'entraînement de modèles de deep learning, deux métriques essentielles sont surveillées : 1) `loss` : mesure l'erreur du modèle sur l'ensemble **d'entraînement** ; c'est cette valeur que l'optimiseur cherche à minimiser. 2) `val_loss` : mesure la même erreur, mais sur un ensemble **de validation** non vu pendant l'entraînement. Elle reflète ainsi la capacité du modèle à **généraliser**. En pratique, la `val_loss` est plus pertinente pour évaluer la qualité réelle du modèle.

### 3.3.1 Comparaison des performances système entre les deux méthodes d'entraînement

Les deux approches ont été évaluées sur un sous-ensemble identique de 6 000 phrases, avec un nombre fixe de 3 époques d'entraînement. Il convient de souligner que les mesures présentées ont été extraites automatiquement à partir des journaux (*logs*) Kubernetes des pods correspondants.

| Approche | Durée totale (s) | Temps CPU (s) | Mémoire max (Mo) |
|---|---|---|---|
| Séquentiel | 701.68 | 520.11 | 6671.97 |
| Distribué (Horovod) | 407.48 | 353.04 | 3632.61 |

TABLE 3.1 – Comparaison des performances système entre les deux méthodes d'entraînement

Comme le montrent le tableau 3.1 et la figure 3.4, le mode d'entraînement distribué présente de meilleures performances comparé au mode séquentiel. Cette supériorité se manifeste notamment en termes de **durée totale d'exécution**, de **temps CPU utilisé** et de **consommation mémoire maximale**, sur l'ensemble des 6 000 phrases de test. De ce fait, l'entraînement distribué se révèle nettement plus efficace que l'entraînement séquentiel pour des volumes de données importants. Il permet de réduire significativement le temps d'exécution tout en optimisant l'utilisation des ressources système.

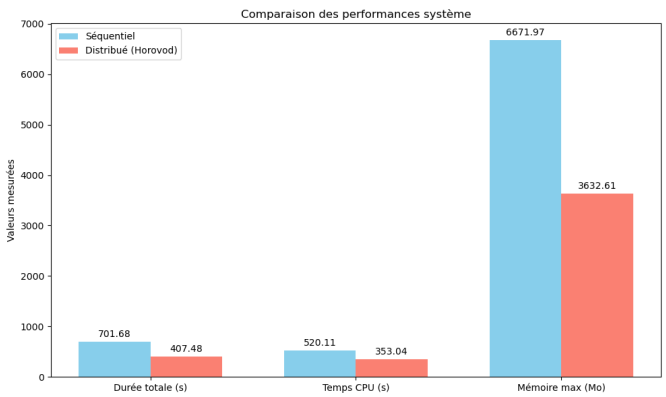


FIGURE 3.4 – Comparaison des performances système entre les deux méthodes d'entraînement

### 3.3.2 Comparaison de l'évolution des paramètres de perte (`loss` et `val_loss`) pour les deux modes d'entraînement : séquentiel et distribué

| Époque | Loss (Seq) | Val_Loss (Seq) | Loss (Dist) | Val_Loss (Dist) |
|---|---|---|---|---|
| 1 | 6.9247 | 6.7291 | 6.5237 | 6.4130 |
| 2 | 6.2609 | 6.7268 | 5.4389 | 6.2886 |
| 3 | 5.8991 | 6.7197 | 4.8241 | 6.3112 |

TABLE 3.2 – Comparaison de l'évolution des paramètres de perte (`loss` et `val_loss`) pour les deux modes d'entraînement : séquentiel et distribué

Le tableau 3.2 ainsi que la figure 3.5 comparent l'évolution des paramètres de perte (`loss` et `val_loss`) pour les deux modes d'entraînement : séquentiel et distribué. En comparant les valeurs de perte, on observe que la méthode distribuée présente de meilleures performances en termes de capacité d'optimisation par rapport à la méthode séquentielle.

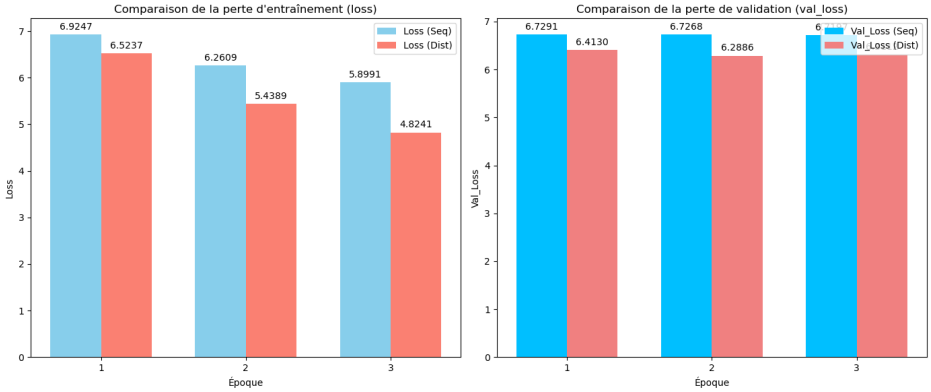


FIGURE 3.5 – Comparaison de l'évolution des paramètres de perte (`loss` et `val_loss`) pour les deux modes d'entraînement : séquentiel et distribué

### 3.3.3 Analyse

- **Temps d'exécution** : le modèle entraîné en mode distribué est **environ 42 % plus rapide** que le modèle séquentiel, à nombre d'époques et données identiques. Cette accélération est directement liée à la parallélisation des calculs.
- **Mémoire** : le mode distribué consomme **près de deux fois moins de mémoire** que le mode séquentiel, ce qui s'explique par la répartition de la charge computationnelle entre les différents processus.
- **Perte d'entraînement (`loss`)** : la courbe de `loss` diminue plus rapidement avec l'entraînement distribué, ce qui traduit une **meilleure efficacité d'optimisation** durant l'apprentissage.
- **Perte de validation (`val_loss`)** : elle est **légèrement plus faible** en mode distribué, ce qui suggère une **capacité de généralisation supérieure** par rapport au modèle entraîné de manière séquentielle.

L'approche distribuée fondée sur Horovod s'avère non seulement plus rapide, mais également plus économe en ressources mémoire. Elle offre en outre une meilleure dynamique d'apprentissage et une

capacité accrue à généraliser sur des données non vues. Cette méthode constitue donc une solution robuste et performante pour l'entraînement de modèles à grande échelle dans des environnements distribués.

# Conclusion Générale

Ce stage m'a permis de mettre en œuvre un cluster Kubernetes avec un pipeline MLOps relativement complet pour des modèles LSTM d'autocomplétion de texte. Malgré un environnement Kubernetes présentant des défis de ressources, les objectifs principaux ont été abordés.

Les compétences acquises durant ce stage sont multiples :

— Maîtrise des concepts de la conteneurisation avec Docker et de l'orchestration avec Kubernetes (Deployments, Services, CronJobs, MPIJob).
— Compréhension et mise en œuvre de l'entraînement distribué avec Horovod.
— Utilisation de NFS pour le stockage partagé dans un contexte distribué.
— Intégration de la surveillance avec Prometheus.
— Développement d'interfaces web avec Flask pour l'interaction avec les modèles.

## Perspectives d'Amélioration et Travaux Futurs

Ce projet offre de nombreuses pistes d'amélioration et d'extension :

1. **Amélioration du Pipeline MLOps :**
   — **Gestion des Ressources Kubernetes :** définir des requêtes et des limites de ressources précises pour tous les pods.
   — **Utilisation de GPUs :** sur des serveurs supportant le GPU.
   — **Monitoring Avancé :** exploiter Prometheus et Grafana pour assurer un suivi détaillé des performances des modèles ainsi que le bon fonctionnement du pipeline.
2. **Amélioration des Modèles LSTM et de l'Application :**
   — Explorer différentes architectures de modèles LSTM, optimiser les hyperparamètres.
   — Augmenter la taille et la diversité du jeu de données d'entraînement.

En conclusion, ce stage m'a offert une expérience pratique enrichissante dans le domaine du MLOps appliqué à un environnement Kubernetes.

# Annexes

Le travail réalisé dans le cadre de ce stage est entièrement documenté et versionné dans le dépôt GitHub suivant :

github.com/AdelBouzidi/Stage_Master1_CHPS_UPVD

Ce dépôt contient l'ensemble des scripts, fichiers YAML, Dockerfile, scénarios d'exécution d'un pipeline MLOps complet basé sur un modèle LSTM pour l'autocomplétion de texte et déployé sur un cluster Kubernetes local.

# Bibliographie


[1] BytePlus. *Horovod Paper : A Comprehensive Guide to Distributed Learning*. https://www.byteplus.com/en/topic/499141?title=horovod-paper-a-comprehensive-guide-to-distributed-learning. [Online] Présentation approfondie du framework Horovod pour l'apprentissage profond distribué, avec exemples d'intégration dans TensorFlow/Keras et bonnes pratiques d'entraînement multi-nœuds. 2023 ( Retour à la page ⇐ 8).

[2] Docker, Inc. *Dockerfile reference*. https://docs.docker.com/engine/reference/builder/. Documentation de référence officielle pour la syntaxe et les bonnes pratiques des Dockerfiles. 2024 ( Retour à la page ⇐ 2).

[3] Docker, Inc. *What is a Container ?* https://www.docker.com/resources/what-container/. Documentation officielle définissant les conteneurs et la technologie Docker. 2024 ( Retour à la page ⇐ 2).

[4] Felix A. Gers, Jürgen Schmidhuber et Fred Cummins. “Learning to Forget : Continual Prediction with LSTM”. In : *Neural Computation* 12.10 (2000). Cet article a introduit la ”porte de l’oubli”, un composant clé des LSTM modernes., p. 2451-2471. URL : https://pubmed.ncbi.nlm.nih.gov/11032042/ ( Retour à la page ⇐ 2).

[5] Google. *tf.keras.layers - TensorFlow Core v2.16.1*. https://www.tensorflow.org/api_docs/python/tf/keras/layers. Documentation officielle décrivant les couches Keras, y compris Embedding, LSTM et Dense, utilisées pour construire de tels modèles. 2024 ( Retour à la page ⇐ 2).

[6] Google. *tf.keras.Model fit method*. https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit. Documentation officielle de l’API pour la méthode ‘fit‘, détaillant les paramètres ‘batch_size‘et‘validation_split‘.. 2024 ( Retour à la page ⇐ 4).

[7] Sepp Hochreiter et Jürgen Schmidhuber. “Long Short-Term Memory”. In : *Neural Computation* 9.8 (1997), p. 1735-1780. DOI : 10.1162/neco.1997.9.8.1735. URL : https://doi.org/10.1162/neco.1997.9.8.1735 ( Retour à la page ⇐ 2).

[8] Yichuan Jiang. *Distributed Deep Learning Training with Horovod on Kubernetes*. https://medium.com/data-science/distributed-deep-learning-training-with-horovod-on-kubernetes-6b28ac1d6b5d. [Online] Tutoriel décrivant comment exécuter un entraînement distribué avec Horovod et TensorFlow dans un cluster Kubernetes, y compris la configuration, l’exécution multi-pods, et l’utilisation de NFS. 2020 ( Retour à la page ⇐ 8).

[9] Keras Team. *Keras Documentation*. https://keras.io/. Documentation générale sur les principes de Keras, y compris le processus d’entraînement. 2024 ( Retour à la page ⇐ 3).

[10] Janani Ravi. *Implement Text Auto Completion using LSTM Networks*. https://app.pluralsight.com/library/courses/implement-text-auto-completion-lstm/table-of-contents. Cours Pluralsight, accès requis. 2021 ( Retour à la page ⇐ 7, 8).

[11] B. Sun. *NFS : Network File System Protocol Specification*. RFC 1094. IETF, mars 1989. DOI : 10.17487/RFC1094. URL : https://www.rfc-editor.org/info/rfc1094 ( Retour à la page ⇐ 3).

[12] The Horovod Authors. *Horovod : Distributed Deep Learning Training Framework*. https://horovod.ai/. Site officiel du framework Horovod. 2024 ( Retour à la page ⇐ 3).

[13] The Horovod Authors. *Keras and TensorFlow — Horovod documentation*. https://horovod.readthedocs.io/en/stable/keras.html. Documentation officielle montrant l’intégration de Horovod avec TensorFlow/Keras, y compris l’utilisation de ‘DistributedOptimizer‘ et les callbacks conditionnés au rank 0. 2024 ( Retour à la page ⇐ 8).

[14] The Kubeflow Authors. *MPI Training (MPIJob)*. https://www.kubeflow.org/docs/components/trainer/legacy-v1/user-guides/mpi/. Documentation officielle de Kubeflow pour la ressource personnalisée MPIJob. 2024 ( Retour à la page ⇐ 3).

[15] The Kubernetes Authors. *Concepts - Kubernetes*. https://kubernetes.io/docs/concepts/. Documentation officielle présentant les concepts fondamentaux de l’architecture Kubernetes. 2024 ( Retour à la page ⇐ 3).

[16] The Kubernetes Authors. *Workloads - Kubernetes*. https://kubernetes.io/docs/concepts/workloads/. Documentation officielle décrivant les objets de charge de travail, y compris les Pods, Deployments et CronJobs. 2024 ( Retour à la page ⇐ 3).

[17] XenonStack. *Distributed Deep Learning Benefits and Use Cases*. https://www.xenonstack.com/blog/distributed-deep-learning. Analyse des avantages de l'entraînement distribué, notamment la réduction du temps d'entraînement et l'amélioration de la scalabilité. 2024 ( Retour à la page ⇐ 3).