TP: Analyse de Sentiment avec des Réseaux RNN et LSTM

Objectif :

**Implémenter un réseau RNN pour l'analyse de sentiment à l'aide d'un ensemble de données au format CSV.**

**Implémenter un réseau RNN pour effectuer la tâche du Part Of Speech.**

**Partie I : Implémenter un réseau RNN pour l'analyse de sentiment à l'aide d'un ensemble de données au format CSV.**

Étapes :

1. Préparation de l’environnement du travail

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import SimpleRNN, Dense, Embedding,

1. Chargement et Exploration du Jeu de Données

* Chargement des données
* Exploration du dataset.

1. Pre-processing

* Nettoyer et prétraiter les textes (enlever la ponctuation, convertir en minuscules, etc.).
* Encoder les données si nécessaire
* Tokeniser les textes en utilisant la fonction Tokenizer de tensorflow.keras.preprocessing.text
* Une fois le corpus de texte entier tokenisé, nous devons convertir chaque critique textuelle en une séquence numérique à l'aide du tokenizer adapté.
* Préciser une longueur de séquence nominale de 200 pour chaque token. max\_length = 200
* Les séquences numériques dont la longueur est supérieure à 200 seront tronquées à la fin, tandis que celles dont la longueur est inférieure à 200 seront complétées par des zéros à la fin.

1. Construction du Modèle RNN basique

* Créer un modèle avec une couche Embedding (La première couche est une couche d'embedding. Elle sert à représenter les mots sous forme de vecteurs dans un espace vectoriel) avec les spécificités suivantes

input\_dim=10000 : Il y a 10 000 mots dans notre vocabulaire (les 10 000 mots les plus fréquents sont utilisés).

output\_dim=32 : Chaque mot sera représenté par un vecteur de 32 dimensions.

input\_length=max\_length

* Ajouter une couche SimpleRNN avec 32 unités. Cela signifie qu'il y a 32 cellules SimpleRNN dans cette couche.
* Ajouter une couche dense avec une seule unité (car il s'agit d'une tâche de classification binaire) et une fonction d'activation sigmoid.

|  |  |
| --- | --- |
| Layer | Dimensions |
| Embedding | 10000 input dimensions, 32 output dimensions |
| SimpleRNN | 32 units |
| Dense | 1 unit, sigmoid activation function |

* Afficher le résumé du modèle avec la fonction .summary.

**HINT**

model = Sequential()

model.add(Embedding(input\_dim=10000, output\_dim=32, input\_length=max\_length))

model.add(SimpleRNN(units=32))

model.add(Dense(units=1, activation='sigmoid'))

1. Compilation et Entraînement du Modèle

* Compiler le modèle en spécifiant l'optimiseur (Adam), la fonction de perte (binary\_crossentropy car c'est une classification binaire) et les métriques à suivre (dans ce cas, l’accuracy).
* Entraîner le modèle avec les données d'entraînement (x\_train et y\_train) pendant 10 epochs, avec un batch size de 128.

1. Évaluation du Modèle

* Évaluer la performance du modèle sur l'ensemble de test.
* Calculer la précision.
* Discuter des métriques de performance.

1. Analyse des Paramètres du Modèle

Expliquer les principaux paramètres du modèle LSTM :

units : Nombre de neurones LSTM dans la couche LSTM.

embedding\_dim : Dimension de l'espace d'incorporation (embedding).

input\_length : Longueur des séquences en entrée.

activation : Fonction d'activation utilisée dans la dernière couche.

* Modifiez le nombre d'unités dans le RNN
* Essayez de modifier le nombre d'unités dans le RNN. Comment cela affecte-t-il la performance du modèle ?
* Utilisez un autre type de fonction d'activation
* Essayez d'utiliser un autre type de fonction d'activation pour la couche de sortie du modèle. Comment cela affecte-t-il la performance du modèle ?

**Partie II : Implémenter un réseau RNN pour effectuer la tâche du Part Of Speech.**

Étapes :

1. Préparation de l’environnement du travail

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, Dense, SimpleRNN

1. Création d’un simple Jeu de Données

* Créer les listes de phrases (sentences) et de parties du discours (pos).
* Générer les listes uniques de mots (word\_list) et de parties du discours (pos\_list).
* Créer des mappings entre les mots et les indices (word2idx, idx2word) et entre les parties du discours et les indices (pos2idx, idx2pos).

**HINT :**

sentences = [['I', 'feel', 'hungry'],

['tensorflow', 'is', 'very', 'difficult'],

['tensorflow', 'is', 'a', 'framework', 'for', 'deep', 'learning'],

['tensorflow', 'is', 'very', 'fast', 'changing']]

pos = [['pronoun', 'verb', 'adjective'],

['noun', 'verb', 'adverb', 'adjective'],

['noun', 'verb', 'determiner', 'noun', 'preposition', 'adjective', 'noun'],

['noun', 'verb', 'adverb', 'adjective', 'verb']]

word\_list =['<pad>'] + sorted(set(sum(sentences, [])))

word2idx = {word:idx for idx, word in enumerate(word\_list)}

idx2word = {idx:word for idx, word in enumerate(word\_list)}

1. Pre-processing

* Transformer les phrases et les parties du discours en séquences d'indices en utilisant les mappings créés précédemment.
* Utiliser la fonction pad\_sequences pour rembourrer les séquences de manière à avoir une longueur uniforme.

**HINT** :

X = list(map(lambda sentence: [word2idx.get(token) for token in sentence], sentences))

y = list(map(lambda sentence: [pos2idx.get(token) for token in sentence], pos))

X = pad\_sequences(X, maxlen=10, padding='post')

X\_mask = (X != 0).astype(np.float32)

X\_len = np.array(list((map(lambda sentence: len(sentence), sentences))), dtype=np.float32)

y = pad\_sequences(y, maxlen=10, padding='post')

1. Construction du Modèle RNN basique

* Créer un modèle avec une couche Embedding (La première couche est une couche d'embedding. Elle sert à représenter les mots sous forme de vecteurs dans un espace vectoriel) avec les spécificités suivantes

input\_dim= len(word2idx): Il y a 10 000 mots dans notre vocabulaire (les 10 000 mots les plus fréquents sont utilisés).

output\_dim= len(word2idx): Chaque mot sera représenté par un vecteur de 32 dimensions.

input\_length=max\_length

* Ajouter une couche SimpleRNN avec 10 unités. Cela signifie qu'il y a 10 cellules SimpleRNN dans cette couche.
* Ajouter une couche dense avec len(pos2idx) unité et une fonction d'activation de votre choix.

|  |  |
| --- | --- |
| Layer | Dimensions |
| Embedding | len(word2idx)input dimensions, len(word2idx) output dimensions |
| SimpleRNN | 10 units |
| Dense | len(pos2idx) unit |

* Afficher le résumé du modèle avec la fonction .summary.

**HINT**

model = Sequential([

Embedding(input\_dim=input\_dim, output\_dim=output\_dim,

mask\_zero=True, , input\_length=10),

SimpleRNN(units=10),

Dense(units= len(pos2idx))

])

1. Compilation et Entraînement du Modèle

* Compiler le modèle en spécifiant l'optimiseur (Adam)
* Entraîner le modèle avec les données d'entraînement des epochs spécifiés

1. Évaluation du Modèle

* Évaluer la performance du modèle sur l'ensemble de test.
* Calculer la précision.
* Discuter des métriques de performance.

1. Analyse des Paramètres du Modèle

Expliquer les principaux paramètres du modèle RNN :

units :

embedding\_dim :

input\_length :

activation :

* Modifiez le nombre d'unités dans le RNN
* Essayez de modifier le nombre d'unités dans le RNN. Comment cela affecte-t-il la performance du modèle ?
* Utilisez un autre type de fonction d'activation
* Essayez d'utiliser un autre type de fonction d'activation pour la couche de sortie du modèle. Comment cela affecte-t-il la performance du modèle ?

Pour cette deuxième partie, vous pouvez se baser sur ce support pour réaliser les étapes du TP : https://github.com/goodboychan/goodboychan.github.io/blob/main/\_notebooks/2020-12-09-01-RNN-Many-to-many.ipynb

**Notions importantes**

**A lire attentivement**

Le choix du batch size, du nombre d'epochs et de la fonction d'activation a un impact significatif sur l'entraînement et la performance d'un modèle LSTM.

1. Batch Size :

Le batch size définit le nombre d'échantillons qui seront utilisés pour mettre à jour les poids du modèle à chaque itération lors de l'entraînement. Voici quelques considérations :

* Impact sur la Mémoire : Un batch plus grand nécessite plus de mémoire. Des batchs trop grands peuvent conduire à des problèmes de mémoire sur les GPU.
* Stabilité de l'Entraînement : Un batch plus grand peut fournir une estimation plus stable du gradient, car il est calculé sur un plus grand nombre d'échantillons. Cependant, il peut également augmenter le temps d'entraînement.
* Stochasticité vs. Déterminisme : Un batch plus petit introduit de la stochasticité dans la mise à jour des poids. Cela peut aider à sortir de minima locaux, mais peut aussi rendre l'entraînement moins stable

1. Epochs :

Le nombre d'epochs définit combien de fois l'ensemble de données complet est passé à travers le réseau de neurones. Voici quelques points à considérer :

* Underfitting vs. Overfitting : Trop peu d'epochs peuvent conduire à un sous-ajustement (underfitting), tandis que trop d'epochs peuvent conduire à un surajustement (overfitting).
* Définition de l'Arrêt Prématuré : Il est souvent utile de surveiller les performances sur un ensemble de validation et d'arrêter l'entraînement lorsque les performances sur l'ensemble de validation commencent à stagner.
* Temps d'Entraînement : Un grand nombre d'epochs peut augmenter considérablement le temps d'entraînement, surtout pour des modèles complexes.

1. Fonction d'Activation :

La fonction d'activation est utilisée pour introduire de la non-linéarité dans le modèle. Voici quelques fonctions d'activation couramment utilisées et leurs impacts :

* Sigmoid : Souvent utilisée dans la dernière couche d'un modèle de classification binaire pour obtenir des probabilités entre 0 et 1. Elle peut avoir des problèmes de disparition du gradient.
* ReLU (Rectified Linear Unit) : Une fonction d'activation très couramment utilisée qui permet un apprentissage plus rapide et résout les problèmes de disparition du gradient.
* Tanh : Une variante du sigmoïde qui génère des valeurs entre -1 et 1. Elle est parfois utilisée, bien qu'un peu moins courante que ReLU.
* Softmax : Utilisée dans la dernière couche pour la classification multiclasse, car elle génère des probabilités pour chaque classe.

Le choix de la fonction d'activation dépend de la nature du problème (classification binaire ou multiclasse) et de l'architecture du réseau.

En pratique, il est souvent nécessaire d'expérimenter avec différents hyperparamètres pour trouver la combinaison qui fonctionne le mieux pour un problème donné.

1. Optimizer

Les optimiseurs sont des algorithmes ou des méthodes utilisés pour minimiser une fonction d'erreur ("loss function") ou pour maximiser l'efficacité de la production. Les optimiseurs sont des fonctions mathématiques qui dépendent des paramètres d'apprentissage du modèle, à savoir les poids et les biais. Les optimiseurs permettent de savoir comment modifier les poids et le taux d'apprentissage du réseau neuronal afin de réduire les pertes.

* Stochastic Gradient Descent (SGD): L'optimiseur de descente de gradient stochastique est l'un des plus simples. Il met à jour les poids du modèle en proportion du gradient moyen sur un mini-lot de données.
* Adam : L'optimiseur Adam est largement utilisé dans les tâches de NLP en raison de sa capacité à adapter automatiquement le taux d'apprentissage et à gérer des ensembles de données volumineux. Il est souvent choisi comme optimiseur par défaut pour de nombreuses applications de NLP.
* RMSprop (Root Mean Square Propagation): RMSprop peut également être efficace dans certaines tâches de NLP. Il adapte le taux d'apprentissage en fonction des gradients récents, ce qui peut être bénéfique pour des tâches où les gradients varient considérablement.
* Nadam: Nadam combine les avantages de Nesterov Accelerated Gradient (NAG) et de l'optimiseur Adam. Il a montré de bonnes performances dans diverses tâches, y compris certaines tâches de NLP.
* SGD avec Momentum: La descente de gradient stochastique (SGD) avec momentum peut également être efficace, en particulier lorsque vous avez des jeux de données plus petits. L'ajout de momentum peut aider à accélérer la convergence et à éviter de rester piégé dans des minima locaux.

Il est recommandé d'expérimenter avec différents optimiseurs pour voir celui qui fonctionne le mieux pour votre tâche spécifique. Les performances peuvent varier en fonction de la complexité de la tâche, de la taille du jeu de données et de l'architecture du modèle. En règle générale, Adam est un bon choix de départ, mais il peut être utile d'essayer d'autres options pour voir si elles conduisent à une convergence plus rapide ou à de meilleures performances sur votre tâche de NLP particulière.