

Nom et Prénom 1 : KABA SEKOU

Nom et Prénom 2 : KHALIL SALAH EDDINE



NI : 537026208

Matière : IFT-7022 Traitement  
automatique de la langue naturelle

Professeur : M. Luc Lamontagne

Période : A 2022

## **Rapport du TP2**

NB : Le code avec word2vec est dans le fichier python nommé word2vec.py, celui de fasttext est dans le fichier nommé fasttext.py et celui de la deuxième est dans le fichier nommé lstm.py .

Pour la tâche 2, nous avons voulu faire l'entraînement avec poutyne, nous avons rencontré des problèmes, ça ne marchait pas, donc nous avons adapté nos propres fonctions qui nous ont permis d'entraîner et d'évaluer notre modèle.

Le fichier d'entraînement de la tâche 2 en utilisant poutyne nommé Tache2LSTM.py est joint à ce rapport mais il ne fonctionne pas.

Veuillez ne pas le prendre en considération, il faut prendre en considération celui nommé juste « lstm.py »

Nos analyses sur la tâche 2 sont faites sur les résultats fournis par le fichier nommé « lstm.py »

## **Introduction :**

Ce rapport a pour but de répondre aux questions du travail 2, il est construit afin de guider la lecture des fonctions implémentées.

L'objectif de ce travail a été l'utilisation des plongements de mots préentraînés, l'entraînement d'un modèle feedforward(MLP) en

premier temps et l'entraînement d'un modèle récurrent (LSTM) en second temps tout en utilisant Pytorch.

## **Tâche 1 : Classification de questions-Réseau feedforward(MLP) et plongements de mots**

### **Tâche 1.a : Fasttext vs word2vec**

Dans cette partie, nous avons utilisé un fichier d'entraînement nommé « questions-train.txt » qui contient différents types de questions pour entraîner notre modèle à partir du réseau de neurones de type feedforward (MLP) pour pouvoir classer la classe d'une question.

Ainsi, dans ce travail, nous allons comparer l'efficacité du modèle construit en utilisant les plongements de mots pré entraînés de word2vec avec celle du modèle construit en utilisant les plongements de mots de fasttext.

**Présentez les résultats clairement à l'aide de figures. Analysez les résultats. Expliquez les fonctions particulières qui ont été implémentées pour pouvoir utiliser les plongements de FastText et de Word2Vec.**

Après avoir créé en **première étape notre jeu de données** en le chargeant, en convertissant nos classes en identifiants numériques, nous avons divisé le jeu d'entraînement en deux sous-ensembles, un pour l'entraînement et le second pour la validation (nous avons choisi 10% pour les données de validation), nous avons ensuite fait **la gestion des plongements de mots pré entraînés de fasttext et de word2vec** par l'entremise de gensim, nous avons utilisé les plongements de mots de 300, nous avons donc créé deux instances, une pour

avoir accès aux embedding de mots de fasttext et l'autre pour ceux de word2vec. Nous **avons créé notre réseau de neurones multicouches à deux couches**, une première qui va convertir notre vecteur de document en une représentation intermédiaire dans la couche cachée et une autre qui va produire les valeurs de sortie pour nous, nous avons appliqué la fonction d'activation Relu sur la première couche.

Par la suite, Pour pouvoir créer un plongement de notre document, nous avons mis en place **deux types de fonction d'aggrégation** qui sont : average\_embedding(qui va prendre la moyenne des embedding des mots de notre document) et maxpool(qui va prendre le maximum sur chacune des dimensions des embeddings de mots de notre document) ; ainsi pour ce faire, dans chacune de ces deux fonctions, nous avons **tokénisé notre document en mots individuels grâce à spacy**, nous avons cherché les **embeddings de chacun des mots de notre document en utilisant fasttext et word2vec** et selon **le type de fonction d'aggrégation**, nous avons retourné le plongement de mots de notre document le correspondant.

Nous avons **créé une classe dataloader** qui permet à la fois de conserver les documents (*dataset*) et de construire un plongement pour chacun de ces documents avec la fonction `_getitem_`. La fonction d'aggrégation d'*embeddings*, utilisée par `_getitem_`, est passée en argument du constructeur de la classe. On a **créé par la suite les dataloaders pour gérer les documents des datasets**. Nous devons sélectionner le type d'aggrégation qui sera utilisée par la classe *FasttextDataset* et *Word2vecDataset*.

Après, nous avons **créé une boucle d'entraînement** pour entraîner nos modèles en commençant par déterminer la taille des *embeddings* et le nombre de classes pour notre jeu de données. Ces valeurs correspondent donc respectivement à la taille des vecteurs en entrée du réseau et au nombre de neurones de sortie du réseau. Pour l'entraînement de nos modèles, on a **utilisé une optimisation de type descente de gradient stochastique (SGD)**, on a utilisé 150 fois notre jeu de données (**epoch=150**) pour entraîner le réseau, on

a utilisé à chaque itération **10 exemples (minibatch)** avant de mettre à jour le poids et on utilise les *data\_loaders* créés à l'étape précédente pour gérer les exemples durant l'apprentissage (*train\_loader*) et pour l'évaluation du modèle (*valid\_loader*) après chaque époque. Pour cet entraînement, on a utilisé la classe **Experiment de poutyne**.

Nous avons ensuite après l'entraînement des modèles tester nos modèles avec de nouveaux exemples du fichier test en créant une fonction utilitaire pour obtenir notre prédiction et enfin on a testé nos modèles sur le fichier test grâce la classe **experiment de poutyne** pour évaluer nos modèles.

Ci-dessous une comparaison détaillée entre les deux types de modèles dans un tableau, la taille de la couche cachée est de 100, hidden\_size=100.

Modèles	Efficacité (test_acc)
MLP_Word2vec (average_embedding)	86%
MLP_Fasttext (average_embedding)	82%
MLP_MLP_Word2vec(maxpool_embedding)	77,6%
Fasttext (maxpool_embedding)	73,8%

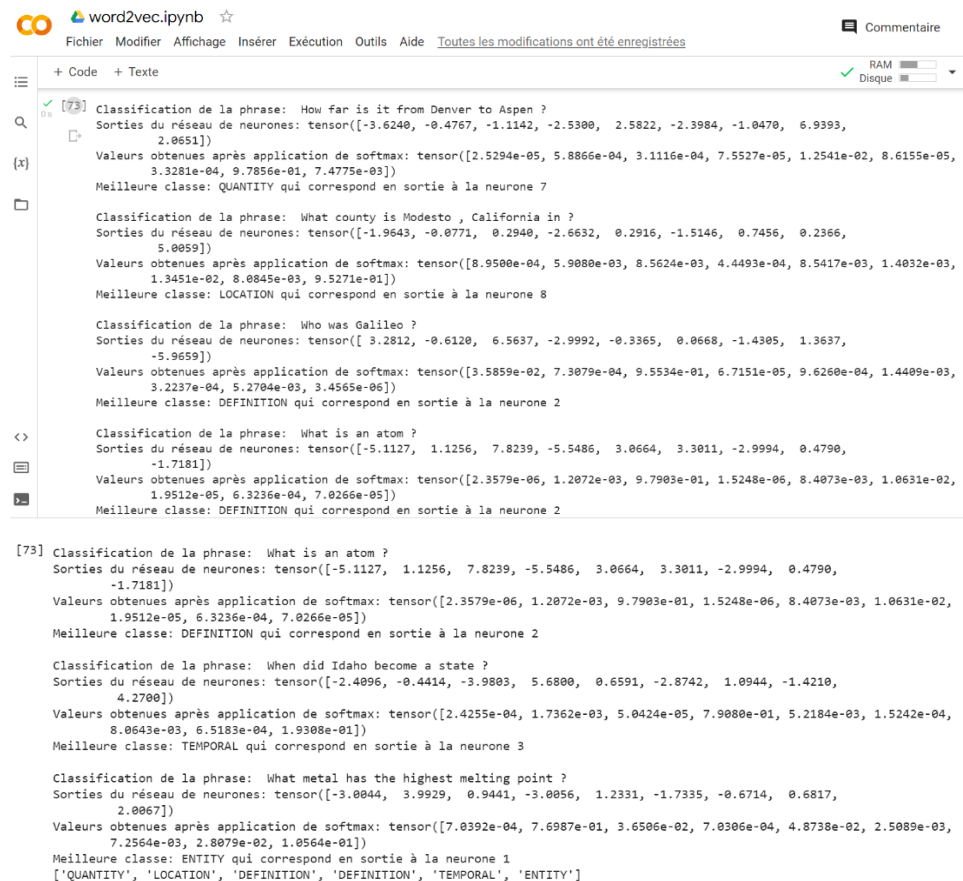
A la lumière de ce qui précède ce tableau, on constate que : les modèles établies en utilisant word2vec traitant chaque mot du corpus comme une entité atomique et générant un vecteur pour chaque produit **ont une efficacité élevée** plus que ceux de fasttext (qui traite chaque mot comme composé de ngrammes de caractère, le vecteur d'un mot est donc fait de la somme de ce caractère n grammes).

Comme conclusion de notre analyse, on peut dire que : word2vec et fasttext ayant le même objectif, apprendre les représentations vectorielles des mots seulement

contrairement à Word2vec, fasttext qui fonctionne à un niveau plus granulaire avec des n-grammes de caractères. Nous concluons que pour les textes en anglais comme notre cas, Word2vec est le meilleur.

Voici quelques prédictions de nos modèles avec (average\_embedding)

### Pour celui établi en utilisant Word2vect



```
word2vec.ipynb
Fichier Modifier Affichage Insérer Exécution Outils Aide Toutes les modifications ont été enregistrées
+ Code + Texte
RAM
Disque

[73] Classification de la phrase: How far is it from Denver to Aspen ?
Sorties du réseau de neurones: tensor([[-3.6240, -0.4767, -1.1142, -2.5300, 2.5822, -2.3984, -1.0470, 6.9393,
2.0651]])
Valeurs obtenues après application de softmax: tensor([[2.5294e-05, 5.8866e-04, 3.1116e-04, 7.5527e-05, 1.2541e-02, 8.6155e-05,
3.3281e-04, 9.7856e-01, 7.4775e-03]])
Meilleure classe: QUANTITY qui correspond en sortie à la neurone 7

Classification de la phrase: What county is Modesto , California in ?
Sorties du réseau de neurones: tensor([[-1.9643, -0.0771, 0.2940, -2.6632, 0.2916, -1.5146, 0.7456, 0.2366,
5.0859]])
Valeurs obtenues après application de softmax: tensor([[8.9500e-04, 5.9080e-03, 8.5624e-03, 4.4493e-04, 8.5417e-03, 1.4032e-03,
1.3451e-02, 8.0845e-03, 9.5271e-01]])
Meilleure classe: LOCATION qui correspond en sortie à la neurone 8

Classification de la phrase: Who was Galileo ?
Sorties du réseau de neurones: tensor([[-3.2812, -0.6120, 6.5637, -2.9992, -0.3365, 0.0668, -1.4305, 1.3637,
-5.9659]])
Valeurs obtenues après application de softmax: tensor([[3.5859e-02, 7.3079e-04, 9.5534e-01, 6.7151e-05, 9.6260e-04, 1.4409e-03,
3.2237e-04, 5.2704e-03, 3.4565e-06]])
Meilleure classe: DEFINITION qui correspond en sortie à la neurone 2

Classification de la phrase: What is an atom ?
Sorties du réseau de neurones: tensor([[-5.1127, 1.1256, 7.8239, -5.5486, 3.0664, 3.3011, -2.9994, 0.4790,
-1.7181]])
Valeurs obtenues après application de softmax: tensor([[2.3579e-06, 1.2072e-03, 9.7903e-01, 1.5248e-06, 8.4073e-03, 1.0631e-02,
1.9512e-05, 6.3236e-04, 7.0266e-05]])
Meilleure classe: DEFINITION qui correspond en sortie à la neurone 2

[73] Classification de la phrase: What is an atom ?
Sorties du réseau de neurones: tensor([[-5.1127, 1.1256, 7.8239, -5.5486, 3.0664, 3.3011, -2.9994, 0.4790,
-1.7181]])
Valeurs obtenues après application de softmax: tensor([[2.3579e-06, 1.2072e-03, 9.7903e-01, 1.5248e-06, 8.4073e-03, 1.0631e-02,
1.9512e-05, 6.3236e-04, 7.0266e-05]])
Meilleure classe: DEFINITION qui correspond en sortie à la neurone 2

Classification de la phrase: When did Idaho become a state ?
Sorties du réseau de neurones: tensor([[-2.4096, -0.4414, -3.9803, 5.6800, 0.6591, -2.8742, 1.0944, -1.4210,
4.2700]])
Valeurs obtenues après application de softmax: tensor([[2.4255e-04, 1.7362e-03, 5.0424e-05, 7.9080e-01, 5.2184e-03, 1.5242e-04,
8.0643e-03, 6.5183e-04, 1.9308e-01]])
Meilleure classe: TEMPORAL qui correspond en sortie à la neurone 3

Classification de la phrase: What metal has the highest melting point ?
Sorties du réseau de neurones: tensor([[-3.0044, 3.9929, 0.9441, -3.0056, 1.2331, -1.7335, -0.6714, 0.6817,
2.0067]])
Valeurs obtenues après application de softmax: tensor([[7.0392e-04, 7.6987e-01, 3.6506e-02, 7.0306e-04, 4.8738e-02, 2.5089e-03,
7.2564e-03, 2.8079e-02, 1.0564e-01]])
Meilleure classe: ENTITY qui correspond en sortie à la neurone 1
['QUANTITY', 'LOCATION', 'DEFINITION', 'DEFINITION', 'TEMPORAL', 'ENTITY']
```

On voit que ce modèle prédit correctement les classes de nos exemples.

### Pour celui établi en utilisant Fasttext

```

Classification de la phrase: How far is it from Denver to Aspen ?
Sorties du réseau de neurones: tensor([-3.0846, -0.1555, -3.0369, 2.3072, 4.9800, -0.0839, 8.3343, -1.8157,
-7.3627])
Valeurs obtenues après application de softmax: tensor([1.0585e-05, 1.9805e-04, 1.1102e-05, 2.3245e-03, 3.3661e-02, 2.1275e-04,
9.6354e-01, 3.7652e-05, 1.4680e-07])
Meilleure classe: QUANTITY qui correspond en sortie à la neurone 6

Classification de la phrase: What county is Modesto , California in ?
Sorties du réseau de neurones: tensor([-1.0218, -0.0462, 1.3585, 3.2619, 0.2006, -0.9693, -0.4925, -0.9803,
-1.5037])
Valeurs obtenues après application de softmax: tensor([0.0106, 0.0280, 0.1140, 0.7650, 0.0358, 0.0111, 0.0179, 0.0110, 0.0065])
Meilleure classe: LOCATION qui correspond en sortie à la neurone 3

Classification de la phrase: Who was Galileo ?
Sorties du réseau de neurones: tensor([ 0.8880, 1.1368, 5.3845, -2.0010, -1.2872, -0.6439, -9.2949, 0.9072,
4.4932])
Valeurs obtenues après application de softmax: tensor([7.6818e-03, 9.8511e-03, 6.8908e-01, 4.2734e-04, 8.7250e-04, 1.6602e-03,
2.9043e-07, 7.8303e-03, 2.8260e-01])
Meilleure classe: DEFINITION qui correspond en sortie à la neurone 2

Classification de la phrase: What is an atom ?
Sorties du réseau de neurones: tensor([-3.2270, 2.0781, 5.8038, -1.3266, 2.9117, -3.7372, -0.5675, 2.8823,
-4.9189])
Valeurs obtenues après application de softmax: tensor([1.0533e-04, 2.1210e-02, 8.8018e-01, 7.0453e-04, 4.8817e-02, 6.3237e-05,
1.5051e-03, 4.7399e-02, 1.9400e-05])
Meilleure classe: DEFINITION qui correspond en sortie à la neurone 2

Meilleure classe: DEFINITION qui correspond en sortie à la neurone 2

Classification de la phrase: When did Idaho become a state ?
Sorties du réseau de neurones: tensor([-0.1720, -1.3858, -1.6111, 3.2652, 0.7061, 3.1586, -0.7431, -1.5903,
-1.9640])
Valeurs obtenues après application de softmax: tensor([0.0156, 0.0046, 0.0037, 0.4862, 0.0376, 0.4370, 0.0088, 0.0038, 0.0026])
Meilleure classe: LOCATION qui correspond en sortie à la neurone 3

Classification de la phrase: What metal has the highest melting point ?
Sorties du réseau de neurones: tensor([-0.0722, 2.3355, 1.0200, 1.8991, 0.2627, -1.7447, -1.5869, -0.6956,
-1.5616])
Valeurs obtenues après application de softmax: tensor([0.0403, 0.4473, 0.1200, 0.2891, 0.0563, 0.0076, 0.0089, 0.0216, 0.0091])
Meilleure classe: ENTITY qui correspond en sortie à la neurone 1
['QUANTITY' 'LOCATION' 'DEFINITION' 'DEFINITION' 'LOCATION' 'ENTITY']

```

On voit que pour nos exemples utilisés avec fasttext, ce modèle n’arrive pas à bien prédire la classe de l’exemple suivant : When did Idaho become a state ? qui, sa vraie classe est « TEMPORAL » mais ce modèle prédit « LOCATION ».

## Tâche 1.b : Impact de la taille de la couche cachée du reseau feedforward

Ici, on utilisera « average\_embedding »

Modèles	Taille de la couche cachée	Performances
MLP_Word2vec	100	86%
MLP_Fasttext	100	82%
MLP_Word2vec	200	87,8%
MLP_Fasttext	200	82,8%

MLP_Word2vec	400	87%
MLP_Fasttext	250	81,6%

On constate que la taille de la couche cachée du réseau feedforward a un impact mais pas gros, en l'augmentant à 200 avec plongement word2vec, on voit que la performance est passée à 87,5% puis en l'augmentant à 400, on voit que la performance reste à 87%. Avec plongement fasttext, on constate qu'en augmentant la taille à 200, la performance passe à 82,8% puis en l'augmentant à 400, elle vient à 81,6%.

NB : il faut retenir que plongement avec average\_embedding donne de bon résultat par rapport à celui avec maxpool\_embedding.

## **Tâche 2 : Complétez le proverbe avec un réseau LSTM et des plongements de Spacy**

**a) Décrivons clairement comment vous avez procédé pour entraîner le modèle LSTM et expliquez précisément comment ce modèle est utilisé pour choisir le bon mot qui complète un proverbe.**

Pour entraîner notre modèle, nous avons commencé **par charger les données d'entraînement et test**, nous avons créé quelques fonctions, une première **create\_vocabulary** qui consiste à créer et sauvegarder le vocabulaire d'un corpus, une deuxième **initialize\_vocabulary** qui consiste à initialiser le vocabulaire, **read\_corpus** qui consiste à lire et convertir un corpus en une liste de tokens, **corpus\_to\_token\_ids** qui consiste à convertir un corpus en token-ids, **batch\_data** qui consiste à structurer nos données en batch\_size sequences continues, **detach hidden** qui consiste à transformer les

données de nos états cachés de notre LSTM en nouveaux tensors avec `require_grad=False`.

Nous avons ensuite **séparé nos données d'entraînement en deux sous-ensembles**, un pour l'entraînement et l'autre pour la validation (nous avons choisi 10% du données d'entraînement initial pour la validation).

Après nous avons **implémenté notre modèle de langue neuronal à base de LSTM**, nous avons ensuite construit notre modèle. À chaque pas temps les variables d'entrées dans notre LSTM sont une minibatch de séquences de token-ids (c.à.d. des séquences d'indices où chaque indice représente la position d'un token dans un vocabulaire) et des tuples  $(h_0, c_0)$  des états récurrents et des états mémoires équivalent aux  $(h_T, c_T)$  de la minibatch précédente (sauf pour la première minibatch où nous initialisons les valeurs de ces variables à 0.0 avec la fonction `init_hidden()`).

Chaque séquence de token-ids est transformée en séquence de *word embeddings* en indexant les représentations *word embeddings* créés par la classe `torch.nn.Embedding()` qui est une matrice de paramètres de dimension  $|V| \times d$ , où  $|V|$  est la taille du vocabulaire (`vocab_size`) et  $d$  est la dimension d'un *word embedding* (`embedding_size`).

Nous appliquons du dropout sur les *word embeddings* en entrée et sur la couche de sortie pour régulariser le modèle.

Et enfin, **pour entraîner notre modèle :**

- Nous avons appliqué du dropout sur les *word embeddings* en entrée et la couche de sortie du LSTM pour éviter le surapprentissage.
- Nous avons démarrée l'entraînement avec une valeur de `learning_rate` élevée et nous l'avons diminué par un facteur de 10 en fonction de la perte sur l'ensemble de validation évaluée à la fin de chaque *epoch* d'entraînement en utilisant la classe `torch.optim.lr_scheduler.ReduceLROnPlateau()`.



- Pour éviter le *exploding gradient problem*, nous avons appliqué la technique de *gradient clipping* en normalisant la norme du gradient avec la fonction `torch.nn.utils.clip_grad_norm_()`.
- Nous avons initialisé avec des valeurs de 0.0 le tuple  $(h_0, c_0)$  de l'état récurrent et l'état mémoire hidden avec la fonction `init_hidden()` uniquement une fois au début de chaque *epoch* d'entraînement et nous avons propagé les nouvelles valeurs de hidden à travers chaque minibatch d'entraînement. En d'autres mots, les données ont été structurées à l'aide de la fonction `batch_data()` de manière à ce que nous pouvons initialiser hidden  $(h_0, c_0)$  de chaque séquence d'une nouvelle minibatch par  $(h_T, c_T)$  de la minibatch précédente. Un désavantage de cette méthode est que nous ne pouvons pas mélanger l'ordre des séquences à chaque *epoch* d'entraînement (c.à.d. `train_loader = DataLoader(train_set, batch_size=seq_len, shuffle=False)`).
- Notre fonction de coût est l'entropie croisée `torch.nn.CrossEntropyLoss()`.
- Nous avons présenté la mesure de **perplexity** qui est une mesure d'évaluation de la qualité de notre modèle de langue et c'est cette mesure qui nous permet de choisir le bon mot qui complète notre proverbe.

## **b) Présentations des résultats obtenus**

```

0.15393992500943862
Epoch 1 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
0.30787985001887724
Epoch 2 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
0.46181977502831584
Epoch 3 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
0.6157597000377545
Epoch 4 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
0.7696996250471931
Epoch 5 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
0.9236395500566318
Epoch 6 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
1.0775794750660703
Epoch 7 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
1.231519400075509
Epoch 8 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
1.3854593250849476
Epoch 9 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
1.5393992500943863
Epoch 10 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
1.693339175103825
Epoch 11 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
1.8472791001132636
Epoch 12 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.001219025122702
Epoch 13 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.1551589501321406
Epoch 14 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.3090988751415793
Epoch 15 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.463038800151018
Epoch 16 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.6169787251604566
Epoch 17 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.7709186501698952
Epoch 18 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
2.924858575179334
Epoch 19 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
3.0787985001887725
Epoch 20 | Test loss = 0.15394 | score=84.60601 | Perplexity = 1.17
la performance est de: 84.60600749905613 %

```

Pour **epoch=20**, pour chaque utilisation de notre jeu de données jusqu'à 20, on a calculé le score, après on a calculé le score moyen et nous avons trouvé comme performance de notre modèle 84,606%.

**Est-ce que le modèle capture bien le langage utilisé dans les proverbes ?**

Ainsi, le modèle arrive à capté bien le langage utilisé dans le proverbe.

**Comparaison des résultats avec ceux obtenus dans le travail #1 avec des modèles de langue N-grammes.**

<b>Modèle</b>	<b>Performance</b>
<b>LSTM</b>	<b>84,606%</b>
<b>N_gramme (trigrammes)</b>	<b>81,3%</b>

**Nous constatons que, LSTM capture mieux le langage que le modèle fait à partir du trigramme.**