Génération de code python avec LSTM

Réalisé par :

BOUFALA Yacine
TAKOUTSING Jerry
BENMANSOUR Amine
KILIC Firat

M1 Data Engineer/Data Scientist

2021-2022

Table des matières

1. Présentation du jeu de données

Notre jeu de données se compose de 3000 fichiers qui sont des scripts écrits en python.

Dans ces scripts, on retrouve certains codes source (keras par exemple), et d'autres scripts plus « classiques » de création de fonctions, classes...

Il est nécessaire de faire un travail en amont sur ces scripts écrits en python avant de les faire ingérer au modèle, c'est ce qu'on appelle le pré-processing.

2. pré-processing

Le pré-processing est une phase cruciale de la création d'un modèle.

Il consiste à apporter des modifications et transformations sur les données afin de les nettoyer pour qu'elles soient plus digestes pour le modèle en question.

Dans notre cas, nous avons utilisé du regex pour le pre-processing, comme suit :

Dans un premier temps, nous nous sommes occupés des espaces qui seraient présents après une virgule et avant un mot / un nombre (par exemple 'quelque chose,<espace>1' -> 'quelque chose,1').

Ensuite, les commentaires ont été éliminés, qu'ils soient sur une ligne ou plusieurs, nous les remplaçons par des \n.

Ces \n sont remplacés par des espaces, et les espaces sont finalement supprimés en passant par un strip. Ainsi, tous les espaces vides inutiles dans le code sont effacés.

Enfin, nous avons tokenisé les données. Cela correspond à transformer un texte en une série de tokens individuels. Par exemple, « J'ai froid » donnerait « J » « ' » « ai » « froid », chaque bout de texte est un token séparé.

2.1. Pré-construction du modèle

Maintenant que les données sont nettoyées et digestes pour notre futur modèle, il faut poser les premières pierres de sa construction.

Notre modèle générant des séquences de mots, il faut définir leur longueur et le pas d'avancement.

Dans notre cas, nous avons fixé une séquence de 2 mots et un pas de 1.

Par exemple, pour la phrase 'Ceci est un test', nous aurons les séquences suivantes : Ceci est, est un, un test.

Cela nous donne 1 356 983 séquences de 2 mots, avec 400 355 tokens uniques.

Pour vérifier que tout s'est bien passé, nous regardons les 10 premiers retours, ce qui nous donne ['import', 'os', 'import', 'unittest', 'from', 'tensorflow.tools.tensorflow_builder.compat_checker', 'import', 'compat_checker', 'PATH_TO_DIR', '='].

On peut se dire aux premiers abords que cela a l'air de s'être bien passé.

Étant donné qu'on cherche à distribuer les calculs, il est nécessaire de définir une taille de batch, un certain nombre de séquences envoyés dans le modèle afin qu'il apprenne d'eux. Nous avons choisi une taille de batch de 4096 séquences, nous avons testé plus ou moins de séquences dans le batch, et ce nombre de séquences était un bon compromis pour la performance et le temps d'exécution.

2.2. LSTM – Présentation et entraînement

Maintenant que les données ont été soumises au pre-processing et à la tokenisation, on peut passer à la construction du modèle et son entraînement par rapport aux paramètres définis.

Le LSTM (Long Short-Term Memory) est un reseau neuronal qui diffère des réseaux classiques puisqu'il est capable de se souvenir d'un problème plus longtemps et répète les opérations pour le résoudre au mieux.

Lorsque l'état de la cellule est porteur de l'information, ces portes aident le nouveau flux d'information. Les portes indiquent les données qu'il est utile de conserver et celles qui ne le sont pas, ce qui permet de les jeter en passant par la porte d'oubli. Ainsi, seules les données pertinentes passent par la chaîne de séquence pour faciliter la prédiction.
Chaque entrée aura par la suite un poids dans la mémoire du réseau.

Maintenant, nous allons entraîner notre modèle. Nous avons choisi un certain nombre de neurones qui composeront le réseau (512 neurones en l'occurrence), et l'optimiseur Adamax. Finalement, nous avons défini une fonction coût qu'on va chercher à minimiser. Cette dernière est l'entropie croisée binaire.

Ensuite, nous avons un dropout de 0.3, ce qui signifie que chaque neurone a une probabilté de 30% d'être temporairement désactivée, ce qui aidera à supprimer un certain nombre de données et d'apprentissage afin d'éviter l'overfitting.

Pour finir, un nombre d'epoch a été fixé. Une epoch est un passage de l'ensemble des données dans le réseau neuronal. Plus le nombre est grand, plus le modèle est précis, mais plus il prendra de temps à apprendre. Nous avons fixé ce paramètre à 20.

Notre modèle va se construire en se basant sur les paramètres précédemment présentés, et nous avons des statistiques (précision et fonction de perte) à chaque epoch. Nous pouvons voir de façon assez claire que la précision augmente à chaque passage, et la fonction de perte diminue.

2.3. Résultats du modèle

Puisque le modèle s'est entraîné sur du script python, la cible est la génération de code python.

On peut voir que les imports se font en début de code, et que le modèle réussit à créer des structures de données telles que des array, et des fonctions de façon assez pertinentes.

Néanmoins, on peut se poser la question d'augmenter le volume de script ingérés par le modèle afin d'affiner la génération.

3. Encodage et décodage

Notre prochain modèle s'appuie sur un transformer, qui contient un certain nombre d'encodeurs / décodeurs et qui s'enchaînent.

L'entrée d'un encodeur est la sortie du précédent. Le dernier décodeur a pour but d'identifier à quels mots du vocabulaire correspondent les sorties du dernier encodeur.

Un encodeur lit le texte qui lui est attribué et le décodeur produit une prédiction pour ce texte en utilisant les masques d'attention que nous verrons ci-dessous.

3.1. Tokenisation

Après avoir spécifié la taille du vocabulaire (nb de tokens dans le tokeniseur), nous avons implémenté les tokens spéciaux suivants :

<s> : Début de séquence (BOS) ou token de classification (CLS)

</s> : Fin de séquejce (EOS) ou token de séparation (SEP) de séquence
<pad> : Token de padding

<mask> : Token de masque

Un MLM (masque d'attention) permet un apprentissage bidirectionnel (ne lit pas que de gauche à droite par exemple).

Pour chaque séquence, 15% des mots sont masqués par le jeton [MASK], et le modèle va tenter de prédire la valeur originale de ce mot masqué.

Nous avons deux fichiers en sortie après avoir entraîné le tokeniseur :

Un merges.txt qui transforme le fichier initial en tokens.

Un Vocab.Json qui transforme les tokens en identifiants de tokens.

Grâce à ces fichiers, nous pouvons passer à l'entraînement de notre modèle.

3.2. Elephas (Apprentissage distribué)

Elephas est une extension de Keras, qui vous permet d'exécuter des modèles d'apprentissage en profondeur distribués à grande échelle avec Spark. Elephas implémente une classe d'algorithmes parallèles aux données au-dessus de Keras, en utilisant les RDD et les trames de données de Spark. Les modèles Keras sont initialisés sur le Master, puis sérialisés et expédiés aux Workers, accompagnés de données et de paramètres de modèle diffusés. Les Worker Sparks désérialisent le modèle, entraînent leur bloc de données et renvoient leurs gradients au Master. Le modèle "Master" sur le pilote est mis à jour par un optimiseur, qui prend les gradients de manière synchrone ou asynchrone. Le modèle ayant besoin des paramètres pour s'exécuter, on peut donc l'entrainer a l'aide des paramètre suivants :

- train_generator_emb : les données en entrée.
- batch_size : lot de donnée en entrée.
- steps_per_epoch : nombre total de donnée
- Epoch : Nombre de fois que le modèle aura accès aux jeux de données.
- Verbose : Pour l'affichage de l'entrainement.

3.3. Configuration et création du modèle

Tout comme le LSTM vu précédemment, le modèle doit s'appuyer sur des paramètres pour s'exécuter.

Cette configuration s'enregistre dans un fichier config.json qui permettra d'utiliser et d'exporter le modèle si besoin.

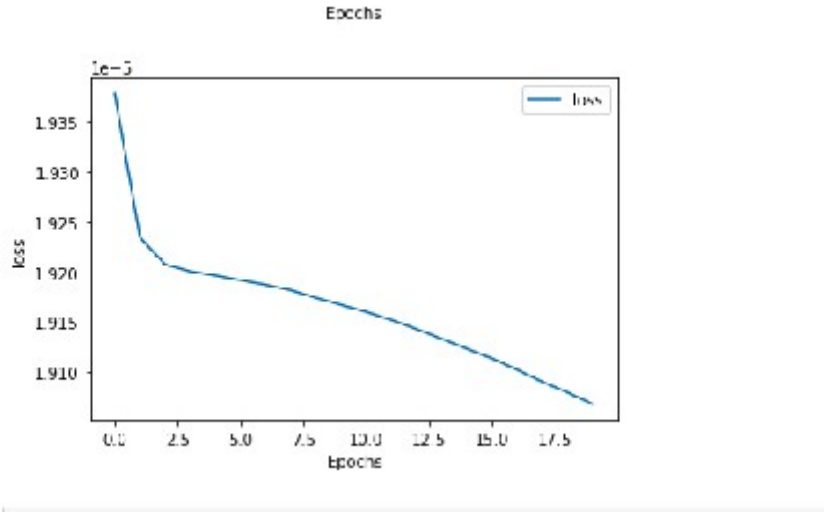
Nous avons spécifié une taille de vocabulaire comme précédemment pour le LSTM, ainsi qu'un nombre de neurones, de têtes d'attention, de couches cachées, une taille de batch...

L'ensemble de ces paramètres nous permettent de créer le modèle et de le tester dans un premier temps en remplissant les masques d'attention.

```
Entrée [44]: # lancement du train avec les couches LSTM et dropout
             model.fit(train_generator, batch_size=batch_size, steps_per_epoch=len(train_generator),
                       epochs=nb_epoch, verbose=1)

             Epoch 1/10
             108/108 [==============================] - 843s 8s/step - loss: 1.1294e-05 - accuracy: 0.0532
             Epoch 2/10
             108/108 [==============================] - 844s 8s/step - loss: 1.1291e-05 - accuracy: 0.0537
             Epoch 3/10
             108/108 [==============================] - 1031s 10s/step - loss: 1.1291e-05 - accuracy: 0.0537
             Epoch 4/10
             108/108 [==============================] - 846s 8s/step - loss: 1.1291e-05 - accuracy: 0.0537
             Epoch 5/10
             108/108 [==============================] - 765s 7s/step - loss: 1.1291e-05 - accuracy: 0.0537
             Epoch 6/10
             108/108 [==============================] - 758s 7s/step - loss: 1.1290e-05 - accuracy: 0.0536
             Epoch 7/10
             108/108 [==============================] - 881s 8s/step - loss: 1.1290e-05 - accuracy: 0.0537
             Epoch 8/10
             108/108 [==============================] - 934s 9s/step - loss: 1.1289e-05 - accuracy: 0.0537
             Epoch 9/10
             108/108 [==============================] - 815s 8s/step - loss: 1.1286e-05 - accuracy: 0.0536
             Epoch 10/10
             108/108 [==============================] - 778s 7s/step - loss: 1.1282e-05 - accuracy: 0.0537

Out[44]: <keras.callbacks.History at 0x7f59435c0e20>
```



3.4. Finalité et axes d’amélioration

En conclusion, le modèle LSTM est totalement fonctionnel, il génère du code qui est très satisfaisant.

Nous pourrions par exemple le ré-entrainer sur un certain nombre de scripts afin d’affiner la génération de code, mais cela prendrait énormément de temps et de ressources, nous ne l’avons donc pas fait dans le cadre de ce projet.

Concernant le modèle BERT, nous avons eu pas mal de complications en rapport avec son implémentation au niveau du code, certaines problématiques ont été résolues, mais les majeures sont toujours d’actualité.

Nous avons donc présenté les pistes et débuts de construction de ce modèle, mais nous n’avons pas la finalité.

Nous pourrions trouver la solution à l’implémentation de ce modèle pour finalement comparer les performances et pertinence du code généré des deux modèles.