Compte rendu du projet (1er sujet dans la liste de sujets méthodologistes)Implémentation d'un modèle d'embedding vectoriel de mots

Le code principal :

Il se trouve sur mon compte GitHub sous https://github.com/javaskater/word2vecpy

1. Le fichier word2vec.py est le fichier d'origine (Python 2.7)
2. Le fichier mycbowskipgram.py est mon travail qui est la version :
    1. La version d'origine portée sous Python 3.10.6 que je détaille juste après dans le rapport
    2. et que j'ai fortement commentée !

Lancement du code principal :

Sous un répertoire files a été récupéré le fichier texte reprenant les Fleurs du Mal de Beaudelaire (utilisé au TP RNN pour la génération de texte)
Sous ce même répertoire files sera généré au format texte le fichier contenant ligne par ligne les mots suivis du vecteur correspondant h de la couche cachée (il n'y a pas de sigmoïde en sortie de la couche cachée)
Le lancement lui même :
On laisse la plupart des paramètres non obligatoires à leurs valeurs par défaut :

```
jpmena@LAPTOP-E2MJK1UO:~/CONSULTANT/word2vecpy$ ./mycbowskipgram.py -train files/fleurs_mal.txt -model files/fleurs_mal_mots.txt
[train] the input training file: files/fleurs_mal.txt
[train] the output model file: files/fleurs_mal_mots.txt
[train] I create a un Continuous Bag Of Words
[train] the number of negative examples (if 0 hierarchival softmax): 5
[train] the size of the hidden layer (word embedding): 100
[train] coeffcient for gradient descending: 0.025
[train] the window size of words: 5
[train] the minimum number of words used for learning the model: 5
[train] the number of parallel processes for learning the model: 1
[train] the output model file files/fleurs_mal_mots.txt is in text format
 Reading 38000 wordsunknown vocab size 8900
Total words in training file 38496
Total bytes in training file 183596
[train] initializing the UNIGRAMM table
[UnigramTable __init__] Filling the Unigram table
[UnigramTable __init__] Unigram table filled
[train] starting the 1 process(es)
[train] the total duration of the training took 0 minutes using 1 threads
[save]Saving output vectors to files/fleurs_mal_mots.txt in binary(False) mode
[__init_process] initialization of process 4063 (parent 4031) out of 1 processes ended
[train_process_0] Worker of number: 1 out of 1 starting to read the input file at (included) 0 position until (excluded) 183596 position
```

-train et -model sont les deux seuls paramètres obligatoires

1. -train pointe vers le fichier texte qui va nous servir à construire notre sac de mots
2. -model est le fichier en sortie les lignes étant classées dans l'ordre du mot le plus fréquent au mot le moins fréquent
    1. NB : (le codage one Hot commence du mot le plus fréquent vers le mot le mois fréquent) il en va ainsi des lignes de la matrice W = syn0 qui sont nos projections de mots dans la couche cachée

Ce sont donc les lignes de W=syn0 qui sont nos lignes (précédées du mot correspondant de la classe Vocab) chaque élément de la ligne est séparé par un espace !

Le but du programme :

Il s'agit d'entraîner un modèle de mots dans un texte découpés en contexte (la taille du contexte est de 5 par défaut, 5 mots avant le mot courant et 5 mots après le mot courant) on fait la moyenne des vecteurs de la couche cachée sur contexte (en excluant le mot courant) et le target en sortie du softmax (représenté par une sigmoïde ici) doit être la représentation one hot du mot courant !

En fait la taille du contexte est un entier tiré au hasard entre 1 et win par défaut à 5 qui est un paramètre de la ligne de commande de notre programme

La classe Vocab :

La classe Vocab_item :

Elle contient un mot du texte appris.

- Ce mot appelé token est contenu dans l'attribut self.word
- Le nombre d'occurrences de ce mot dans le texte d'apprentissage est dans l'attribut self.count
- Et un codage Huffman de ce mot est dans l'attribut self.code (on verra plus bas ce qu'apporte l'encodage Huffman quand il est généré) et comment il est utilisé dans Hierarchical SoftMax du CBOW

Le constructeur, La méthode __init__

C'est ce constructeur qui fait tout le boulot de stocker les mots et leurs représentations (classe VocabItem).
Ce constructeur prend en paramètres

- fi (paramètre en entrée de la ligne de commande) : le fichier (texte qui va nous servir pour générer le vocabulaire et son encodage Huffman
- min_count (également paramètre en entrée de la ligne de commande) : n'est utilisé que dans la méthode sort de la classe Vocab, c'est le nombre minimal d'occurrences nécessaires pour que le terme soit pris en compte si ce mot apparaît dans le texte moins de min_count il n'est pas créé comme Vocab_Item, à sa place son nombre d'occurrences est ajouté au Vocab_item unk (pour unknown inconnu)

Ce constructeur crée pour chaque mot un Vocab_item l'ajoute dans un tableau (self.vocab_items)
Noter qu'il aura ajouté 2 mots :

- <BOL> pour begin of line (début de ligne)
- <EOL> pour end of line (fin de ligne)

C'est la méthode __sort (appelée en fin de constructeur __init__) qui met à jour et réorganise le tableau (list en Python) des Vocab_items (en mettant en première) et génère le dictionnaire vocab_hash qui associe à chaque mot (token) trouvé dans le texte son indice (sa position) dans le tableau des vocab_items. On classe les token du plus fréquent au moins fréquent (sachant qu'au texte initial on a ajouté les tokens <unk> <bol> <eol>)

La méthode __sort :

On crée une list python tmp qui va remplacer le premier self.vocab_items
on l'initialise avec le vocab_item <unk> et on la complète avec tous les vocab_items précédemment trouvés (si un vocab_item a été trouvé moins de min_count fois sont nombre d'occurrences est ajouté au vocab_item <unk> mais lui même n'est pas ajouté à la liste temp)

on trie cette liste tmp de vocab_items par nombre décroissant de nombre d'occurrences du vocab_item c'est ce que permet la fonction lambda lambda vi : vi.count (vi étant une instance de Vocable_item)

```
tmp.sort(key=lambda vi: vi.count, reverse=True)
```

(les auteurs on appelé un vocab_item token par abus de langage)
tmp devient alors le nouveau self.vocab_items
On itère ensuite sur tmp en prenant l'indice du tableau pour créer le nouveau self.vocab_hash qui prend en clé vi.word (vi pour vocab_item) et en valeur l'indice qui est la position du vi dans le tmp maintenant trié

tests du sort en interactifs ;

```
>>> a = ['Mena','Jean-Pierre']
>>> a.sort(key=lambda str: str.lower())
>>> a
['Jean-Pierre', 'Mena']
>>> a.sort(key=lambda str: str.lower(), reverse=True)
>>> a
['Mena', 'Jean-Pierre']
```

Pour récupérer les bonnes valeurs triées :

```
>>> for i,n in enumerate(a):
...    print(f"indice {i} valeur {n}")
...
indice 0 valeur Mena
indice 1 valeur Jean-Pierre
```

Plus sur la One Hot représentation d'un mot dans ce programme :

Dans cette même classe Vocab fait appel à la méthode sa méthode list pour trouver les lignes de la matrice d'entrée syn0 (Wh : matrice des poids entre la matrice d'entrée de dimension vocab_size) et la couche cachée. H le vecteur en sortie de syn0 est la projection du mot attendue, Cette donnée h en entrée de la couche caché de dimension dim (paramètre en entrée du programme si pas spécifié sur la ligne de commande sa valeur par défaut est de 100). Ce vecteur est multiplié par syn1 (transposée de W) pour retrouver une représentation onHot à comparer avec la représentation attendue de l'entrée.

Cette représentation on Hot d'un token (mot) est donné par l'indice retourné par la méthode list (que nous expliciterons plus bas) qui est pour ce token la valeur dans le dictionnaire self.vocab_hash correspondant à l'indice de ce mot dans le self.vocab_items.

Cet indice donne la position du 1 dans la one hot représentation de ce mot (token) cette one_hot representation a bien sur pour longueur vocab_size et plus le 1 est près du début/haut du vecteur plus ce mot est fréquent dans le texte d'apprentissage

La méthode __get_item__ :

Connaissant l'indice d'un mot dans le vocabulaire (ou la position du 1 dans la one hot representation) cette méthode me permet de retrouver le vi (VocabItem) correspondant en appelant vocab_instance[indice] (comme si une instance de la classe Vocab était une liste Python).
Se souvenir que l'on accède à l'indice du token/mot dans la one hot representation par self.vocab_hash[token]

La méthode __len__ :

cette méthode me permet d'obtenir le nombre de vocab_items recensés en appelant directement len(vocab_instance) (comme si une instance de la classe Vocab était une liste Python). C'est elle

qui nous donne le vocab_size utilisé pour initialiser les matrices syn0 (Wh) et syn1 (transposée de W)

La méthode __iter__ :

Permet d'itérer sur les vocab_items par une boucle for directement sur une instance de Vocab

La méthode __contains__

Elle prend en paramètre un token/mot que l'on note ici key

elle teste si ce mot existe en tant que clé dans self.vocab_hash (y a t'il une one hot representation assocée). Elle permettra de savoir si token in vocab_instance ou self (si on est une méthode de la classe comme c'est la cas de list)
Le in dans vocab_has compare la clé key (paramètre de la méthode) aux clés de self.vocab_hash comme le montre le code ci dessous testé en interactif

```
PS C:\Users\jeanp> python
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> hash = {'hello':34, 'world':45}
>>> 34 in hash
False
>>> 'hello' in hash
True
>>> a = {'a':1, 'b':2}
>>> c = 'c' in a
>>> c
False
>>> c = 'a' in a
>>> c
True
```

La méthode list :

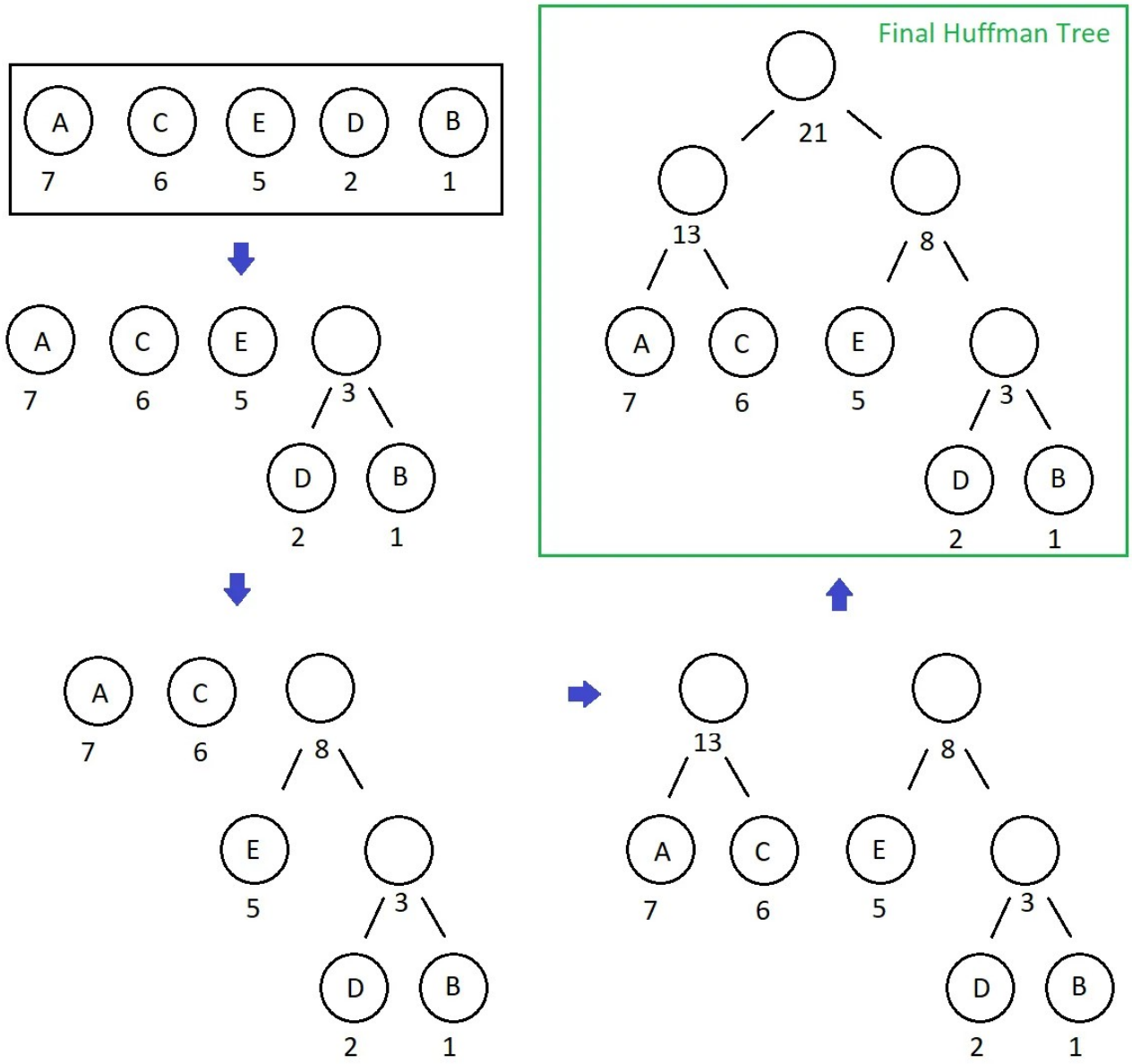
Cette méthode permet à partir d'une liste de token/mot de retourner une liste des indices des vocabItems dans la classe Vocab donc des positions des 1 dans les OneHot representations des mots correspondants
Comme expliqué dans ce lien https://stackoverflow.com/questions/2217001/override-pythons-in-operator :
token in self fait appel à la méthode *self.__contains(token)* qui retourne true si le mot fait partie du vocabulaire et si tel n'est pas cela correspond aux mots du texte avec moins de min_count occurrences.
Dans le premier cas on renvoie self.vocab_hash[token].
Dans ce dernier cas on renvoie dans la liste de retour l'indice du mot créé à cet effet self.vocab_hash['<unk>']

de là le paramètre d'entrée de la méthode spéciale __contains__ précédente.
le self.vocab_hash est juste là pour nous permettre de retrouver la posiition du mot dans self.vocab_items sans avoir à le reparcourir ! Il prend le mot brut en paramètre !

Un grosse méthode la méthode huffmann encoding (TODO) :

C'est une bonne méthode pour compresser des données

Une bonne explication (et une implémentation simple en Python) de cet algorithme est donné par le lien
https://towardsdatascience.com/huffman-encoding-python-implementation-8448c3654328
Binary tree on met à gauche la valeur la moins grande et à droite la valeur la plus grande !!!
Les plus fréquents ont le plus de zéros et la séquence binaire la plus courte …
et il n'y a pas de code long qui comprenne un sous code précédent (parcours unique de chaque branche de l'arbre jusqu'à sa feuille).



Pour TotalGain j'apprends l'usage de data.count('o')

```
>>> data = "Bonjour tout le monde"
>>> data.count('o')
4
>>> data.count('b')
0
>>> data.count('B')
1
```

La méthode principale :

Noter que le neoud qui n'est pas une feuille emporte 2 nœuds (feuille ou pas) a pour fréquence la somme des fréquence de ses enfants cf site :

From the collection, we ***pick out the two nodes with the smallest sum*** *of probabilities and combine them into a new tree whose root has the probability equal to that sum.*

1. *We add the new tree back into the collection.*
2. *We* ***repeat*** *this process until one tree encompassing all the input probabilities has been constructed.*

To combine we put the node on the left if it the biggest of the combined nodes (the comming edge will be 0) right otherwise (the coming edge will be one)
Ainsi Pour A (figure) précédente j'aurai le code 00, pour C le code 01, ... et pour B 111

Le code de ce Huffmann Encoding proposé par cet article de Blog a été mis dans le répertoire tests :

Le Code est sous ***tests/huffmann.py*** (je l'ai largement commenté)
Il s'agit ici d'encoder les caractères d'une chaîne de caractère en fonction de leur fréquence d'apparition dans un texte.
Chaque caractère est représenté par une instance de la classe Node qui n'a que des attributs :

- symbol: la lettre représentée (vide si ce n'est pas un nœud feuille)
- prob : le fréquence de présence de cette lettre ou la somme des prob des (2) nœuds enfants si ce n'est pas un nœud feuille
- left : le nœud enfant gauche (None par défaut car on commence par des nœuds feuilles)
- right: le nœud enfant droit (None par défaut car on commence par des nœuds feuilles)
- code : 1 si je suis le nœud enfant droit du parent (le moins fréquent des 2 nœuds enfants), 0 si je suis le nœud enfant gauche du parent (le plus fréquent des 2 nœuds enfants),

Dans cette version on a une méthode récursive *def CalculateNodes(node, val=''):* node est le nœud en cours de l'arbre et val le code huffman du parent. Cette méthode appelée depuis le nœud racine (nodes[0] : the only remaining node in the List after the algorithm completes et val vide). Permet de remplir un dictionnaire python à partir des seuls nœuds feuilles avec en clé le caractère correspondant et en valeur le code Huffmann correspondant
Dans le programme principal on prend une chaîne de caractères et on calcule son encodage Huffmann (en concaténant les valeurs récupérées du dictionnaire précédent).

La méthode Huffmann encoding par word2vec :

On initialise 3 tableaux

1. **count** : tableaux de vi.count (vi pour vocabItem object) pour les entrées de 0 à vocab_size 1 on a les fréquences des termes actuels du plus fréquent au moins fréquent. On étend ce tableau de vocab_size-1 valeurs positionées par défaut 1e15 une valeur de fréquence que l'on ne peut pas dépasser. Ces positions de vocab_size à 2*vocab_size seront les positions des parents (nœuds non feuilles) du premier calculé par l'algorithme (fréquence la moins importante) au dernier calculé par l'algorithme (le nœud racine celui qui a la fréquence la plus importante)

2. **parent** : un tableau destiné à contenir les indices des parents. Il a pour taille vocab_size (feuilles) + vocab_size – 1 (les nœuds non feuilles) – 1 (on ne stocke pas le nœud racine qui n'a pas de parent par définition)

3. **binary** : un tableau destiné à contenir les valeur des arêtes arrivant sur le nœud (1 si je suis le plus fréquent des 2 enfants, 0 si je suis le moins fréquent des 2 parents, finalement l'inverse du codage Huffmann proposé précédemment). Il a pour taille vocab_size (feuilles)

+ vocab_size – 1 (les nœuds non feuilles) – 1 (on ne stocke pas le nœud racine qui n'a pas de arête entrante par définition)

Test, en python interactif : on initialise en double la taille du tableau initial avec des valeurs maximales pour la partie doublée

```
>>> a = [1,2,3,4]
>>> b = [x for x in a] + [1e15]
>>> b
[1, 2, 3, 4, 1000000000000000.0]
>>> b = [x for x in a] + [1e15]*len(a)
>>> b
[1, 2, 3, 4, 1000000000000000.0, 1000000000000000.0, 1000000000000000.0, 1000000000000000.0]
```

Dans la boucle principale de cette version de l'algorithme de Huffmann :

NB Python 3 range est l'ancien Python 2 xrange (lien)

```
>>> for i in range(len(a) -1):
...     print(i)
...
0
1
2
```

Plusieurs remarque sur cette boucle

- pos1 initialisé à vocab_size-1 ne peut que descendre est est donc l'indice des nœuds feuilles restant à traiter, pos2
- pos2 initialisé à vocab_size ne peut qu'augmenter et est le prochain nœud non feuille à créer
- A la fin de l'itération, La liste des nœuds à traiter (cf. le schéma de l'algorithme de Huffmann plus haut et pour rappel trouvé sur ce lien) au sens de l'algorithme de Huffmann expliqué précédemment vas de 0 à pos1 (nœuds feuilles restant à traiter) et de vocab_size à vocab_size + pos2 (pour les nœuds non feuille)
- A chaque pas on crée un noeud non feuille (un parent) à l'indice vocab+i la position du prochain noeud à créer (Il y a vocab_size -1 nœuds à créer ou vocab_size -2 si on exclue le nœud racine qui n'apporte pas de code Huffmann).
  - dans l'algorithme de Huffmann une fois terminé on montre que pour N noeud feuilles il y a N-1 noeuds non feuilles
- A la fin de l'itération i sont encore dans la liste à traiter (du schéma ci dessus) les nœuds feuilles dont l'indice dans la cable Vocab de ***0 à pos1*** et les nœuds non feuilles dont l'indice va de de ***pos2 a vocab_size+i*** CF. Les 2 noeuds non feuilles dans le schéma (ci dessus à la dernière itération). L'indice vocab_size+i il devient l'indice du parent de min1 et min2 et son count est la somme des counts de min1 et min2 (comme dans l'algorithme de Huffmann classique)
  - Pour ce qui est du binary de min1 il reste à 0 rien à faire car le tableau (liste python) est composée de zéros
  - Pour ce qui est du binary de min1 il est mis à 1 car entre le terme en position min1 et le terme en position min2 il est celui qui a le plus grand nombre d'occurences
- Noter que l'on appelle 2 fois la condition *if pos1 >= 0: if count[pos1] < count[pos2]:* la première mois pour trouver min1 la position du terme le moins fréquent des 2 la seconde fois pour trouver min2 la position du terme le plus fréquent des 2.
  - En effet, à la fin d'une itération on a le moins fréquent des nœud feuilles à l'indice pos1 face au moins fréquent des nœud non feuilles à l'indice pos2. Ce sont ces 2 noeuds que l'on va comparer à l'itération suivante

NB : Au tout début count[pos2] est 1e15 donc jamais la plus petite fréquence !
NB : Le tableau count n'ést utilisé que dans cette boucle principale afin de comparer les fréquences des termes non feuilles avec ceux des termes feuilles ou non comme expliqué dans l'algorithme Huffmann de base

Après cette boucle principale, une autre boucle pour associer à chaque vocab_item son Huffmann code

A partir d'ici on sort de l'usage traditionnel du code de Huffman (algorithme de compression) pour l'adapter à notre besoin à savoir **le Hierarchical SoftMax**

Rappel ; la class Vocab a une méthode __iter__ ce qui permet de lancer un *for i, vocab_item in enumerate(self):* afin de récupérer tous les vocab_items de mon vocabulaire
Pour chaque vi (vocab_item) trouvé :
on a sa position dans la Vocabulaire et donc dans nos tableaux **parent** et **binary**.
NB : L'indice 2 * vocab-size – 2 est l'indice du nœud racine d'un Huffman Tree (vocab_size nœuds feuilles et vocab_size -1 nœuds parents) bien que les tableaux parent et binary s'arrêtent juste avant, tout simplement parce que le nœud racine n'a pas de parent et que ce nœud racine n'apporte pas de code Huffmann (1 ou 0)
Le tableau vi .path :
est l'ensemble des entrées du tableau parent (à condition qu'elles soient au-delà de vocab_size -1) donc les nœuds non feuilles de mon premier parent au parent juste avant la racine
La formule *vocab_item.path = [j - vocab_size for j in path[::-1]]* transforme ce tableau en un tableau en un tableau allant du nœud précédent la racine au nœud parent de mois même et on ramène les position dans Vocab de mes nœuds parents à des nœuds du Vocabulaire (j – vocab_size : le parent juste avant la racine étant associé à la valeur la plus fréquente de nœud.
Le tableau vi.code :
ici aussi on inverse les positions des 1 et 0 : vocab_item.code = code[::-1] le premier 1 ou 0 correspond au code arrivant sur le nœud avant la racine et le dernier correspond au 1 ou 0 pour le vocab item…
Les tableaux vi.code et vi.path :
Vont donc :

- pour path : du terme racine au terme de mon parent direct (en redescentant l'arbre)
  - rapportés (j - vocab_size) aux termes du vocabulaire
- pour code : du code du parent avant la racine à mon code (en redescendant l'arbre)
- code et path on donc la même taille

Usage du Huffman Encoding :

Dans la recherche de termes à associer au mot recherché usage du Hierarchical SoftMax dans le path créé pour un VocabItem on a d'abord le terme juste avant le root_idx translaté (position dans Vocab) dans les termes les plus fréquents et ensuite le fils parmi les terme d'un peu moindre fréquence etc ...

le Unigramm Table :

Cette Table est destiné ayant un mot donné de me renvoyer un nombre N de negative samples (des indices dans Vocab qui ne pointent pas vers le mot recherché).

The Unigram algorithm :

http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/ explique le pourquoi du negative sampling et notamment pourquoi la puissance 3/4 = 0.75 (chapitre ***Selecting Negative Samples***).
If you play with some sample values, you'll find that, compared to the simpler equation, this one has the tendency to increase the probability for less frequent words and decrease the probability for more frequent words.
Comme expliqué toujours dans le même post :

They have a large array with 100M elements (which they refer to as the unigram table). They fill this table with the index of each word in the vocabulary multiple times, and the number of times a word's index appears in the table is given by P(wi)
** table_size. Then, to actually select a negative sample, you just generate a random integer between 0 and 100M, and use the word at that index in the table. Since the higher probability words occur more times in the table, you're more likely to pick those.*

Le constructeur, la méthode init :

Elle prend en paramètre une instance de vocabulaire.
En effet dans ce grand tableau de 100000000 on associe aux indices i de ce tableau la valeur j du terme de vocabulaire (du plus fréquent au moins fréquent) tant que i / (taille du tableau) est inférieur à la probabilité cumulée (cumulée car pour les indice du tableau on ne repart pas à 0).
A la fin de méthode cette table devient un attribut de notre instance de UnigrammTable

La méthode sample :

Elle prend en paramètre count, à savoir le nombre de terme de vocabulaire que l'on veut tirer au hasard.
On tire count fois un indice au hasard entre 0 et 100000000. on récupère les indice des termes de vocabulaire correspondant dans un tableau.
On a plus de chance de tirer les termes d'indice faibles (de fréquence la plus élevée) mais le fait de calculer la probabilité à partir du count^0.75 donne plus de chance aux terme les moins fréquent !

Pourquoi une Unigramm table :

Dans l'algorithme CBOW le negative sampling consiste à tirer au hasard neg (paramètre du programme par défaut à 4) termes (indices de la classe Vocab) qui ne soient pas le terme recherché. On fait confiance à la taille du vocabulaire pour ne pas tomber sur le terme recherché (token_indice)

La méthode principale train :

Elle récupère tous les paramètres passés en ligne de commande avec leurs valeurs par défaut !
Elle crée un vocabulaire à partir du texte (classe Vocab) et pour ce vocabulaire

- calcule le Huffmann Encoding adapté au (Hierarchical Softmax) pour chaque terme de Vocabulaire.
- Crée une table Unigramm Table qui nous servira pour le negative sampling.

Elle crée les matrices syn0 et syn1 via la méthode *init_net*.

init_net :

- initilialise syn0 (équivalent de Wh dans le cours RCP209)
  - comme les différents termes du texte sont encodés en one-hot encoding la dimension d'entrée d = vocab_size = len(Vocab) dim (méta paramètre) est l'équivalent de la dimension L du cours
  - matrice dont les lignes sont les projection des différents VocabItems de la classe Vocab, les premières lignes représentant les mots les plus fréquents et les dernières lignes les mots les moins fréquents. Ce sont les lignes de cette matrice (après apprentissage) que la méthode save enregistre (avec le terme correspondant devant) dans le fichier de sortie au format texte (choix par défaut) ou binaire (Nous exploiterons le fichier au format texte dans le programme distance.py). Cette matrice est initialisée par des zéros
- intialise syn1 (équivalent de transposée(W) dans le cours RCP209)
  - Le but étant de retrouver une représentation one-hot en sortie la dimension K de sortie de W devient ici vocab_size = len(Vocab) et la dimension d'entrée (dimension de la couche cachée) L = dim. Cette matrice est initialisée par des valeurs aléatoires comprises entre -0.5/dim et +0.5/dim
- syn0 et syn1 sont traduites en Array de valeurs décimales traduites en c_type (pour accélérer les calculs

- Array est une classe de la librairie multiprocessing de python et permet de créer des tableaux partagés en processus de la classe Pool, ici un processus qui travaille sur syn0 ou syn1 ne le verrouille pas (chaque processus travaille sur des zones de lignes différentes de syn0 et syn1).
- Value est une classe de la librairie *multiprocessing* de python qui permet de partager le comptage total de mots traités

Retour à la procédure principale train :

Train va ensuite appeler le Pooling de la librairie multiprocessing de Python avec pour chaque process du Pool initialisation d'un processus via la méthode __init_process et lancement du processus via la méthode train_process.
Création d'un objet de la classe Pool avec son initialisation via la méthode __init_process
On map chaque processus de pid allant de 0 à num_process -1 (num_process est par défaut à 2) avec la méthode train_process via la méthode map_async de l'objet Pool de la librairie multiprocessing de Python.

__init_process

Le travail principal de cette méthode est de préparer les variables partagées par un même processus

```
global vocab, syn0, syn1, table, cbow, neg, dim, starting_alpha
global win, num_processes, global_word_count, fi
```

Ici global ne signifie par que la variable est partagée par tous les fonctions du programme mais elle n'est partagée uniquement entre la méthode __init_process et train_process d'un même processus.
Ici par défaut (sauf spécification en ligne de commande) seuls 2 processus sont appelés.
Syn0 et syn1 reprennent l'adresse des Array correspondants en les traduisant en c-types (toujours pour accélérer les opérations sur les coefficients) ils font partie des variables partagées au sein d'un même processus (mot clé global).

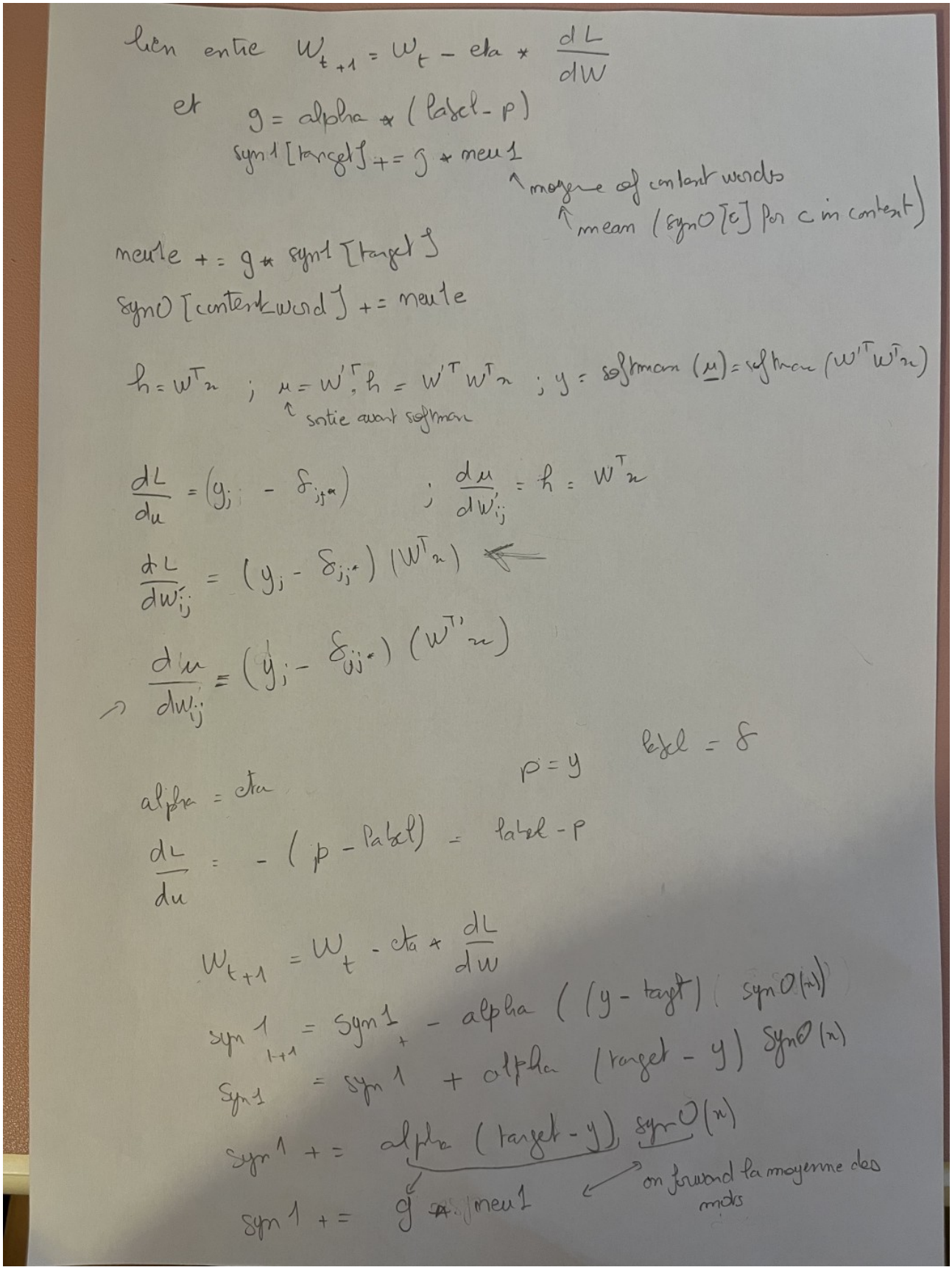
train_process:

Cette méthode reçoit en paramètre pid qui est une valeur qui lui a été attribué par map_async de l'objet pool. Rappel cette valeur va de 0 à num_process-1 c'est à dire par défaut elle vaut 0 ou 1 suivant le processus dans lequel on est.
Cette valeur de pid est pratique pour partager le fichier texte que l'on relit de start à end calculés en fonction de ce pid et du nombre de processus
Remarque : on se partage le fichier texte en nombre d'octets et non nombre de mots, même si une ligne n'est pas lue complètement par un process, elle sera lue complètement par le process précédent et le mot coupé ira dans le hash <unk> de Vocab. Le morceau de phrase coupé sera lu 2 fois, on accepte le risque ! (on n'a pas beaucoup de processus par rapport à la taille du Vocabulaire)
Remarque :
On met à jour la Value de multiprocessing uniquement tous les 10000 mots traités et on en profite pour diminuer alpha d'autant (mais pas au dessous de de 0.001 * valeur de début de alpha)
On traite ligne par ligne et pour chaque ligne on traite chaque terme dans sa fenêtre.
Cette fenêtre est de taille 2 * (valeur aléatoire entre 1 et win) sachant que win est le paramètre de la ligne de commande dont la valeur par défaut est 5 !
Ici on en traitera que du **CBOW** (Continuous Bag of Words)
Le Batch est ici de 1 et est répété (pour le même h = neu1) pour différents target qui ont un label qui vaut 1 si je suis h = neu1 (target = token_indice) que j'essaie de deviner et 0 dans le cas des negative samples (autres indices que token_indice) !

Mise à jour se syn1 pour CBOW (syn1 = transposée de W du cours) :

Il s'agit de comprendre les lignes 356 à 364 de mycbowskipgram.py (méthode train une fois la fenêtre préparée)

Si je reprends le cours
https://cedric.cnam.fr/vertigo/Cours/ml2/coursDeep1.html
Et les travail de WAFA suivant :



p est la valeur (à la coordonnée target) de Wt * h (transposée de W multipliées par vecteur colonne h).

target va de 0 à K-1 (K=vocab_size dans le cas de notre auto encodeur).
Ce qui donne dans notre cas syn1 * neu1 et la coordonnée qui nous intéresse yi,target = p (i le ième mot étudié, target de 0 à K-1 la coordonnée du vecteur Y de sortir) est le produit scalaire syn1[target] avec neu1
(label – p) calcule le ***- Deltai,target*** (ici i=1 car le batch est limité à target), - car on multipliera par alpha) car notre batch n'est que de 1
La ligne d'indice target de syn1 que je veux mettre à jour est la colonne d'indice target de W (car on se souvient que syn1 = Wt -la transposée de W du lien Cédric précédent et chez nous L devient dim et K devient voca_size car on est une sorte d'auto encodeur)
la colonne hi1, .., hiL du cours est chez nous la colonne neu1
dl/dW(limité à la colonne target) = Deltai,target * neu1 ((p-label) * neu1 de dimension L à savoir dim chez nou) qui est ce qui mettra à jour la colonne target de W donc la ligne d'indice target de syn1 à savoir ***syn1(target)***
La mise à jour se traduit bien ce qui est écrit ligne 364 syn1[target] += g * neu1 (avec *g = - alpha * (p-label)*)
Remarque : Attention neu1 est la moyenne des projections dans la fenêtre du mot (sans le mot en question) car on veut apprendre un mot à partir de ses voisins dans la fenêtre

Mise à jour de syn0 pour CBOW (syn0 est Wh dans le cours) :

Si je reprends le cours
https://cedric.cnam.fr/vertigo/Cours/ml2/coursDeep1.html
Reprise du cours sur la mise à jour de Wh = syn0 :

$$\frac{\partial L}{\partial s_i} = \begin{pmatrix} y_{1i} \\ y_{c^*i} \\ y_{Ki} \end{pmatrix} - \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$ la Jacobienne est en ligne

$s_i = W^T h_i + b$ — i de 1 à N

Matrice Jacobienne

$$\begin{pmatrix} s_1 \\ s_K \end{pmatrix} = \begin{pmatrix} w_{11} & w_{1K} \\ w_{L1} & w_{LK} \end{pmatrix} \begin{pmatrix} h_1 \\ h_L \end{pmatrix}$$

$$\frac{\partial s}{\partial h} = \begin{pmatrix} \frac{\partial s_1}{\partial h_1} & \frac{\partial s_1}{\partial h_L} \\ \frac{\partial s_K}{\partial h_1} & \frac{\partial s_K}{\partial h_L} \end{pmatrix} = \begin{pmatrix} w_{11} & w_{L1} \\ w_{1K} & w_{LK} \end{pmatrix}$$

$$\begin{pmatrix} h_1 \\ \vdots \\ h_L \end{pmatrix} = \sigma \begin{pmatrix} u_1 \\ u_L \end{pmatrix}$$

$$\frac{\partial h}{\partial u} = \begin{pmatrix} \frac{\partial h_1}{\partial u_1} & \frac{\partial h_1}{\partial u_L} \\ \frac{\partial h_L}{\partial u_1} & \frac{\partial h_L}{\partial u_L} \end{pmatrix} = \begin{pmatrix} \sigma(u_1)(1-\sigma(u_1)) & \cdots & 0 \\ 0 & \cdots & \sigma(u_L)(1-\sigma(u_L)) \end{pmatrix}$$

W^T

$$\frac{\partial L}{\partial u} = (y_1 \; y_{c^*1} \; y_K) \begin{pmatrix} w_{11} & w_{L1} \\ \vdots & \\ w_{1K} & w_{LK} \end{pmatrix} \begin{pmatrix} \sigma(u_1)(1-\sigma(u_1)) & & \\ & \ddots & \\ & & \sigma(u_L)(1-\sigma(u_L)) \end{pmatrix}$$

(W^T)

$1 \times L$

$\odot (\sigma(u)(1-\sigma(u)))$

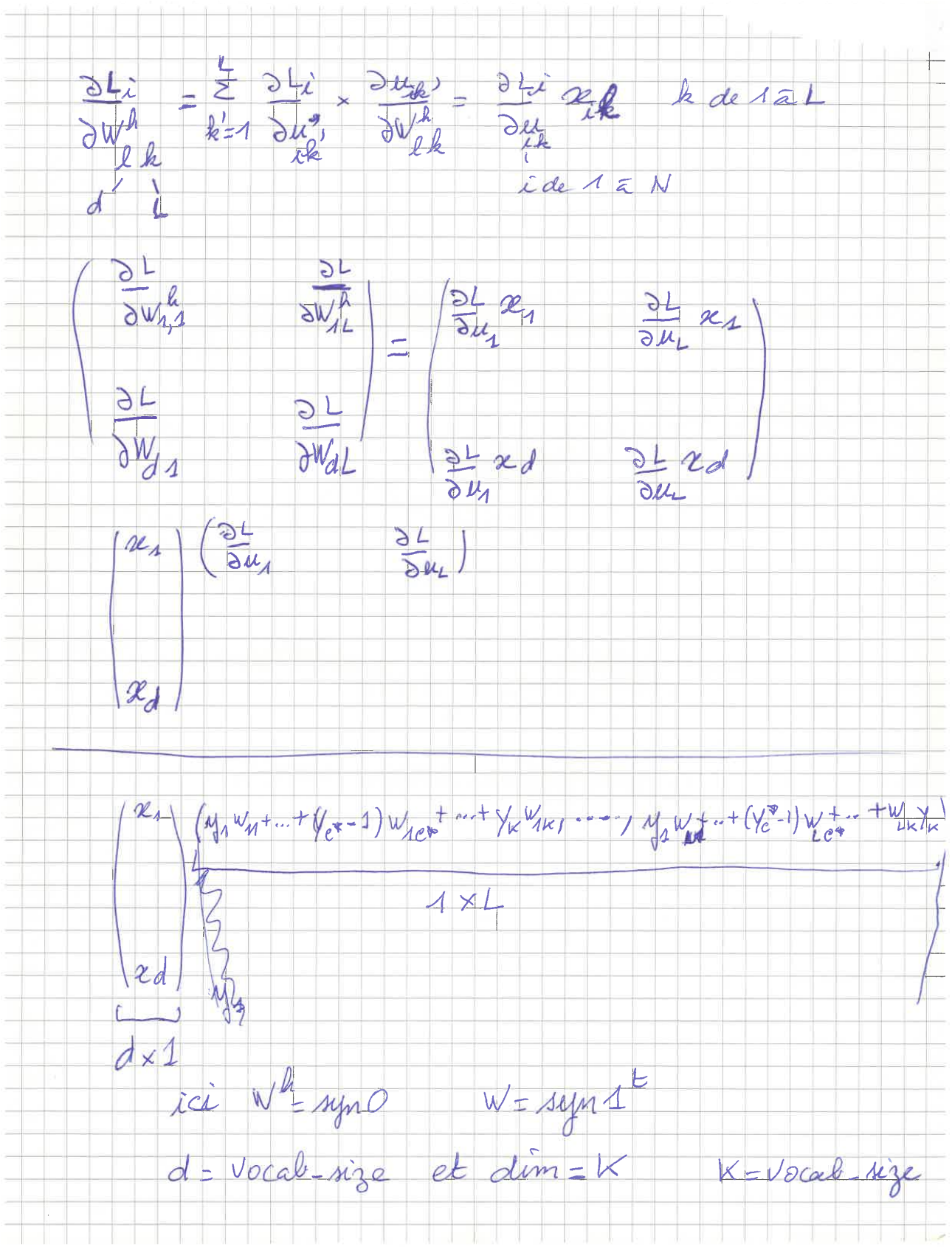
$1 \times L$

$$W^h = \begin{pmatrix} w_{11} & w_{1L} \\ w_{d1} & w_{dL} \end{pmatrix}$$

$$u_m = \sum_{p=1}^{d} x_p W_{pm} + b_m$$

$$\frac{\partial u}{\partial W_h} = \begin{pmatrix} \frac{\partial u_1}{\partial w_{11}} & \frac{\partial u_1}{\partial w_{m0}} & \frac{\partial u_1}{\partial w_{dL}} \\ \frac{\partial u_m}{\partial w_{m0}} & & \\ \frac{\partial u_L}{\partial w_{11}} & \frac{\partial u_L}{\partial w_{m0}} & \frac{\partial u_L}{\partial w_{dL}} \end{pmatrix}$$

$$\frac{\partial u_m}{\partial w_{m0}} = \frac{\partial}{\partial w_{m0}} \left(\sum_{p=1}^{d} x_p W_{pm} + b_m \right) = \begin{cases} x_m \\ 0 \end{cases} \text{ si } m = 0$$



C'est remplacé dans le code de mycbowskipgram.py lignes 363 et 367,368

C’est comme si tous les deltai,k (i de 1 à N, k de 1 à K=vocab_size, cf. fin d la seconde feuille ci dessus) étaient à 0 sauf pour l’indice de notre target ! L’activation en sortie de la couche caché (sigma dans le cours) est ici non une sigmoïde mais la fonction identité !
NB : pour la mise à jour de syn0, on met à jour les coefficients correspondants aux voisins dans la fenêtre de notre mot étudier (et pas les coefficients correspondant au mot) cf. ligne 367,368 de mycbowskipgram.py car ce sont les projections de ces derniers qui vont servir en syn1 à redécouvrir le mot que l’on a en entrée).

Sauvegarder :

Sauvegarder est appelé par le programme principal train une fois que les processus on terminé leurs exécutions !
Par défaut on sauvegarde en mode texte à moins que l’on spécifie différemment sur la ligne de commande)
En mode texte on a voca_size lignes chaque ligne est un terme de Vocab dans l’ordre des indices on récupère en début de ligne le token/mot correspondant suivi de la projection correspondante (syn0[indice du mot dans le vocabulaire]) chaque valeur est séparée par un blanc.
C’est ce fichier que j’exploite dans ***distance.py***.

Les fichiers de tests qui m’ont permis de comprendre le code :

Ils sont tous dans le répertoire tests :

tests/huffman.py :

Comme expliqué précédemment ce fichier permet de tester l’agorithme de Huffman de base, algorithme utilisé pour compresser un texte, comme expliqué sous
https://towardsdatascience.com/huffman-encoding-python-implementation-8448c3654328

tests/shared_arrays_and_values.py

Permet de tester la librairie multiprocessing telle qu’elle est utilisée dans word2vec.
Notamment l’usage du np.array partagé Array et Value partagée quand on lance on voit que les valeurs du tableau on été incrémentées 2 fois (une fois par chaque processus) :

Lancement :

jpmena@LAPTOP-E2MJK1UO:~/CONSULTANT/word2vecpy$ ***./tests/shared_arrays_and_values.py***
[initialize_worker] Initializing worker pid 1308 parent pid 1307 no execution 1...
[initialize_worker] Initializing worker pid 1309 parent pid 1307 no execution 2...
[task_0] Worker executing task... pid 1308 parent pid 1307 no execution 1
[task_1] Worker executing task... pid 1309 parent pid 1307 no execution 2
le tableau est devenu 3.0|2.0|3.4|4.5
le global word count vaut 4
Le tableau à l’origine est tmp = np.array([1.0, 2.0, 3.4, 4.5]) chaque processus en augmente la valeur d’indice 0 de 1 ! Il permet de tester la classe Array du multiprocessing de Python
le global word count à l’origine 0 est incrémenté de 2 par chaque processus. Il permet de tester la classe Value du multiprocessing de Python

tests/simple_array.py

Version du *multiprocessing* de python plus simple que précédemment. Juste pour tester l'enchaînement des opérations, le pool.stop et le pool.join.

tests/distance2words.py :

Version simple pour calculer la distance entre 2 mots à partir du fichier sortie du programme principal *mycbowskipgram.py*. Est à l'origine de du programme ***distance.py*** !

tests/File/read.py :

Tester le déplacement dans un fichier d'entrée texte et la coupure d'une ligne telle que utilisée par train_process qui coupe le fichier d'entrée en fonction du pid donné par pool_async

Lancer l'application mycbowskipgram.py

Ici on prend en entrée le fichier texte des fleurs du mal utilisé au TP RNN pour la génération de texte. Je l'ai mis dans files !
On laisse les paramètres par défaut on ne donne dans la ligne de commande que les 2 paramètres obligatoires, à savoir :

- *-train* le fichier texte en entrée que l'on va traiter pour créer notre vocabulaire et notre sac de mots
- *-model* le fichier en sortie on garde le format texte par défaut c'est le fichier des words des vocab items avec les projections correspondantes à savoir les lignes associées de syn0.
- -processes on le met à 2, nombre de processus en parallèle pour le traitement du fichier d'entrées (la valeur par défaut est de 1)

jpmena@LAPTOP-E2MJK1UO:~/CONSULTANT/word2vecpy$ ***./mycbowskipgram.py -train files/fleurs_mal.txt -model files/fleurs_mal_mots.txt -processes 2***
[train] the input training file: files/fleurs_mal.txt
[train] the output model file: files/fleurs_mal_mots.txt
[train] I create a un Continuous Bag Of Words
[train] the number of negative examples (if 0 hierarchival softmax): 5
[train] the size of the hidden layer (word embedding): 100
[train] coeffcient for gradient descending: 0.025
[train] the window size of words: 5
[train] the minimum number of words used for learning the model: 5
[train] the number of parallel processes for learning the model: 2
[train] the output model file files/fleurs_mal_mots.txt is in text format
Reading 38000 wordsunknown vocab size 8900
Total words in training file 38496
Total bytes in training file 183596
[train] initializing the UNIGRAMM table
[UnigramTable __init__] Filling the Unigram table
[UnigramTable __init__] Unigram table filled
[train] starting the 2 process(es)
[train] the total duration of the training took 0 minutes using 2 threads
[save]Saving output vectors to files/fleurs_mal_mots.txt in binary(False) mode
[__init_process] initialization of process 1094 (parent 1065) out of 2 processes ended
[train_process_0] Worker of number: 1 out of 2 starting to read the input file at (included) 0 position until (excluded) 91798 position
[train_process_1] Worker of number: 2 out of 2 starting to read the input file at (included) 91798 position until (excluded) 183596 position
[__init_process] initialization of process 1093 (parent 1065) out of 2 processes ended

Distance.py :

C’est le programme qui est lancé après avoir langé le programme principal
je prends 2 mots en paramètres je retrouve leurs vecteurs neu dans le fichier texte (je traduit ce qui est texte en float64)
La fonction appelée est find_words_vectors elle lit le fchier des projection créess de WordVectors (on y traduit les valeurs du vecteur neu1 directement en float64 !)

Et la lambda get_word_vector permet de retrouver le mot dans la tableau de WordVectors généré à partir du fichier de projections !

On calcule alors la distance cosinus entre 2 mot c’est une méthode statique de WordVector)

Lancement de la méthode :

En paramètre

1. le fichier des projections du CBOW obtenu via la méthode save précédemment expliqué, Ici on prend la version texte de ce fichier
2. Le mot 1 et le mot 2 dont on veut calculer la distance entre

jpmena@LAPTOP-E2MJK1UO:~/CONSULTANT/word2vecpy$ ***./distance.py -model files/fleurs_mal_mots.txt -mot1 cette -mot2 quand***
We found 656 word vectors in the file: files/fleurs_mal_mots.txt
[WordVector.calcdistance] between the words cette and quand the distance is 0.000552561110181806
[main] between cette and quand the distance is 0.000552561110181806