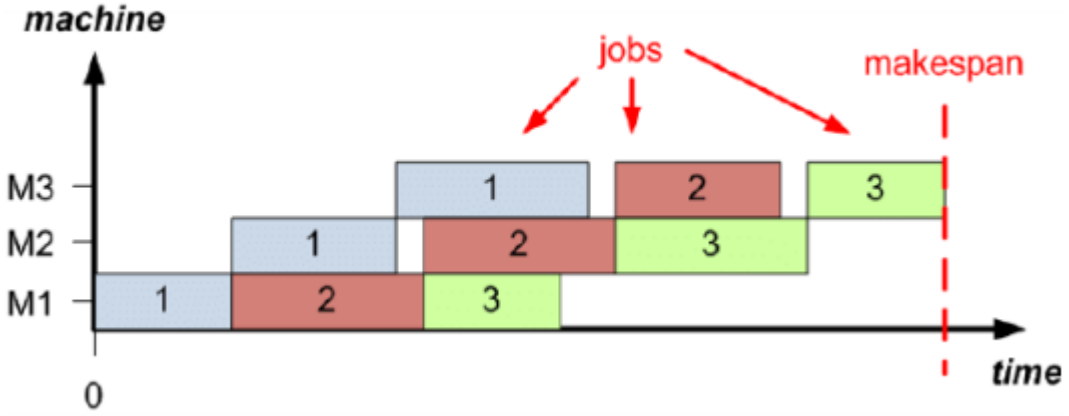
Projet IA Semestre 9

Sommaire :

I - Présentation du sujet

Le problème du Flowshop à 2 machines avec minimisation du temps total de fin de traitement sur la deuxième machine aussi appelé F2| |P Ci dans la notation à 3 champs de (Graham, 1979) est un problème d'ordonnancement populaire dans le domaine de la recherche opérationnelle. Il est catégorisé comme un problème NP difficile. Le problème du flowshop est composé d'un ensemble de processus, ce sont des couples de nombre qui indiquent le temps de traitement du processus sur la machine 1 et 2, on peut les noter : $P \in N\ 2$. Chacun de ces processus doit être exécuté sur la machine 1 puis sur la machine 2 dans un ordre précis, l'ordre des processus est identique pour la machine 1 et 2. Le but est alors de trouver un ordonnancement / une séquence de ces processus afin de minimiser la date de fin de traitement de chacun de ces processus sur la machine 2.



Une nouvelle approche a été publiée (Federico Della Croce, 2011). Il y est présenté une approche par matheuristique qui doit rivaliser, si ce n'est dépasser les approches actuelles concernant ce problème. Cette méthode utilise une approche plus mathématique d'où le terme de matheuristique.

Elle crée une séquence initiale à partir d'un ensemble de processus non

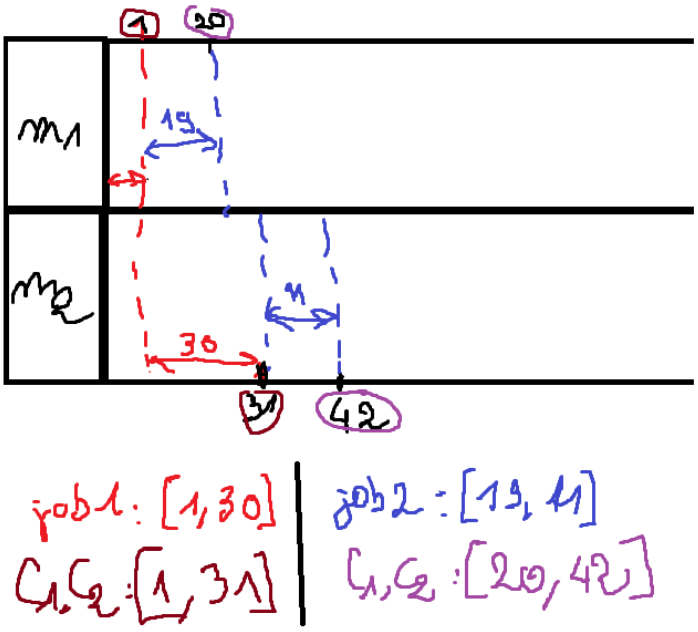
ordonnés, puis elle va appliquer un algorithme itératif d'optimisation de fenêtre afin d'améliorer la séquence.

Un des problèmes majeurs que l'on retrouve dans cette solution est la façon dont les fenêtres sont trouvées. Afin d'optimiser la séquence, les fenêtres, que l'on peut définir comme un intervalle de la séquence initiale avec une position et une taille sont trouvées de façon aléatoire. L'optimisation de toutes les fenêtres trouvées aléatoirement est la partie prenant le plus de temps dans la matheuristique,

Ce sujet cherche à corriger ce problème en appliquant des méthodes supervisées de deep learning pour la prédiction de la position et taille de la fenêtre à optimiser dans la séquence.

Ainsi, nous modélisons la séquence d'ordonnancement sous d'une liste de tuple représentant les processus, chaque processus étant représenté par un tuple dont les 2 premières valeurs sont les processing time sur la machine 1 et 2, et les 2 dernières valeurs étant les temps auxquels le processus se termine sur la machine 1 et 2.

Par exemple :



```
num_instance: 4448
X_train shape :  (10670, 100, 4)
 =>  [ 1 30  1 31] [19 11 20 42]
Y_train shape :  (10670, 100, 1)
 =>  [0]
X_test shape :  (3552, 100, 4)
 =>  [ 1 14  1 15]
X_validation shape :  (3553, 100, 4)
 =>  [ 1 19  1 20]
85.34854799509048  % of zeros in the labels
```

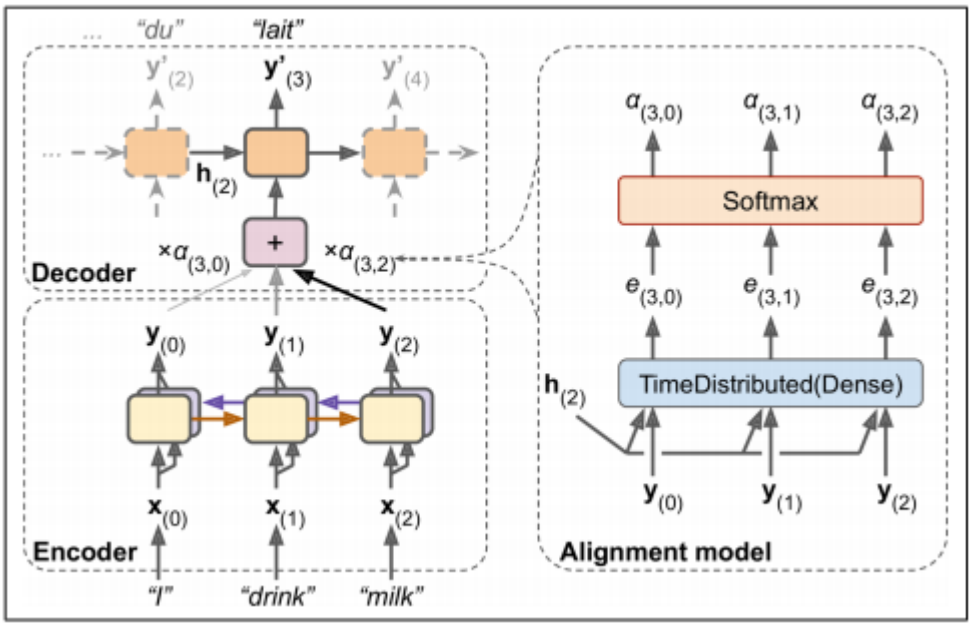
Aussi, l'approche envisagée dans ce projet est de modéliser la position et la hauteur de la fenêtre comme une séquence de binaire (1 : appartient à la séquence, 0 : non).

II - Approche envisagée

On est donc dans une approche "sequence to sequence", et le premier modèle envisagé est un encodeur décodeur à base de cellules récurrentes avec mécanisme d'attention additive (ou attention de Bahdanau). C'est une architecture qui a eu un succès énorme dans le domaine du NLP (Natural Language Processing) en particulier.

Elle est divisée en 2 parties : un **encodeur** qui encode la séquence d'entrée sous la forme d'un "context vector" transmis au **décodeur** qui lui va générer la séquence de sortie.

Le mécanisme d'attention est utilisé pour renforcer le lien entre encodeur et décodeur en ajoutant une notion *d'attention* à chaque étape temporelle du décodeur à l'égard de la séquence de l'encodeur. Il s'agit simplement de trouver des paramètres qui servent de coefficients d'importance des éléments de la séquence de l'encodeur pour chaque étape temporelle du décodeur. Ces paramètres sont trouvés par un modèle d'alignement entraîne conjointement avec le modèle et prenant en entrée les sorties de l'encodeur et l'état caché de l'étape temporelle précédente du décodeur pour chaque étape temporelle du décodeur.



Les hyperparamètres d'un encodeur décodeur à base de RNN (Recurrent Neural Network) sont :

- l'algorithme d'optimisation utilisé pour les descentes de gradient
- les fonctions d'activation de chaque couche
- l'ajout ou non de mécanismes minimisant le surajustement
- **la taille des états cachés des cellules récurrentes**
- **le type de mécanisme d'attention**
- **la fonction de perte**

III - Problème de déséquilibre des classes

Un problème soulevé initialement lors des tests du modèle décrit précédemment portait sur les inégalités de valeurs des labels dans les séquences de sortie. En effet, les fenêtres à trouver (1) sont plus petites que la séquence en entier (0), en moyenne elles représentent 15% de la taille de la séquence de sortie avec un écart-type assez faible. Ce problème s'explique par le fait que le modèle, cherchant à minimiser la fonction de perte, s'oriente naturellement par l'option de facilité qui est de mettre tous les éléments de la séquence prédite à 0.

Ceci étant un problème typique en deep learning, tensorflow offre des outils pour y remédier afin de pondérer les labels lors du calcul de la fonction de perte : `tf.nn.weighted_cross_entropy_with_logits` permet d'ajouter une pondération pour contrebalancer le poids des valeurs 0 dans le calcul de la fonction de perte.

```
return K.mean(
    tf.nn.weighted_cross_entropy_with_logits(labels=y_true, logits=y_pred,pos_weight=pos_weight), axis=-1)
```

```
qz * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x))
```

IV - Développement

A - Mécanismes d'attention

J'ai implémenté 2 nouveaux mécanismes d'attention dans le fichier notebook original :

- l'attention multiplication (ou attention de Luong)
- une variante appelée attention générale

Ces 2 mécanismes sont réputés donner de meilleurs résultats sur les tâches de NLP que que l'attention de Bahdanau premièrement introduite.

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

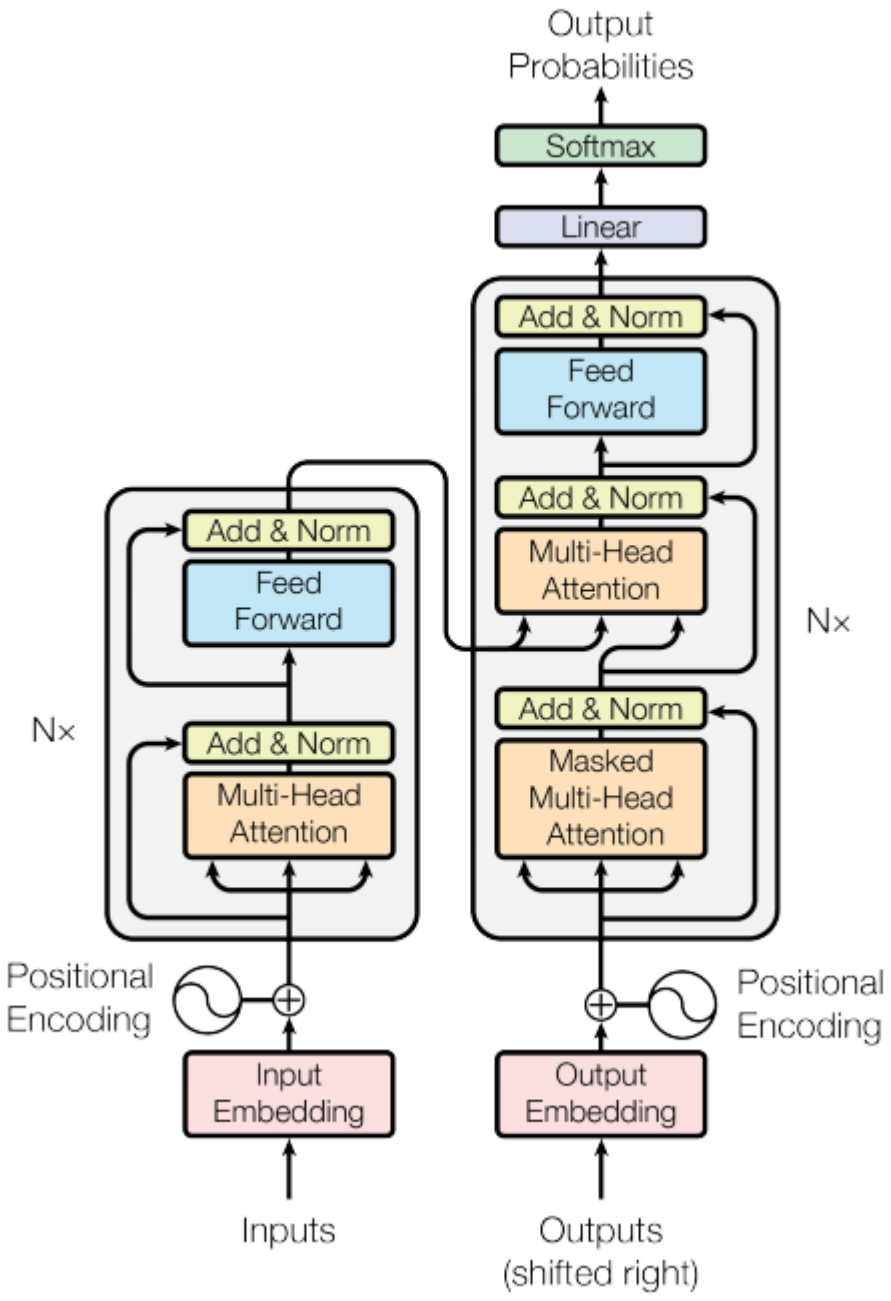
$$\text{with } \alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)}$$

$$\text{and } e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}{}^{\top} \mathbf{y}_{(i)} & \textit{dot} \\ \mathbf{h}_{(t)}{}^{\top} \mathbf{W}\, \mathbf{y}_{(i)} & \textit{general} \\ \mathbf{v}^{\top} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & \textit{concat} \end{cases}$$

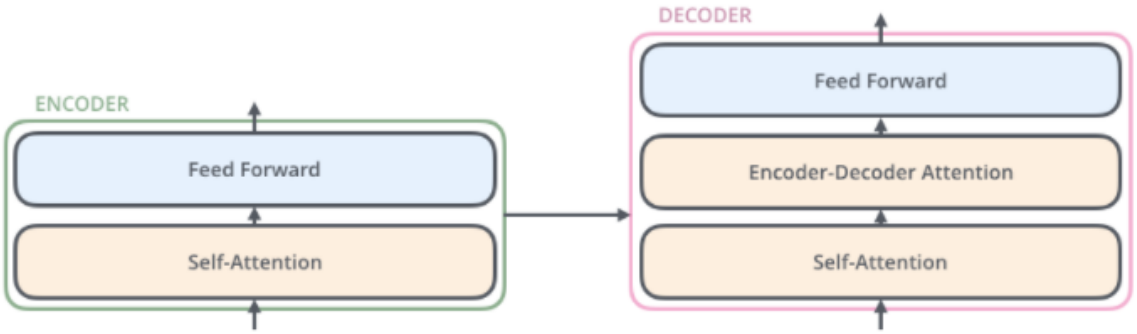
On voit avec les formules "dot" et "general" qu'il s'agit de faire un produit scalaire entre les matrices d'état caché du décodeur et de sorties de l'encodeur afin de mesurer la similarité et obtenir les coefficients d'attention. Dans la variante, les sorties de l'encodeur subissent tout d'abord une transformation linéaire (une couche fully connected distribuée temporellement sans ajout de biais).

Les implémentations sont disponibles dans le fichier Flowshop.ipynb original.

B - transformer

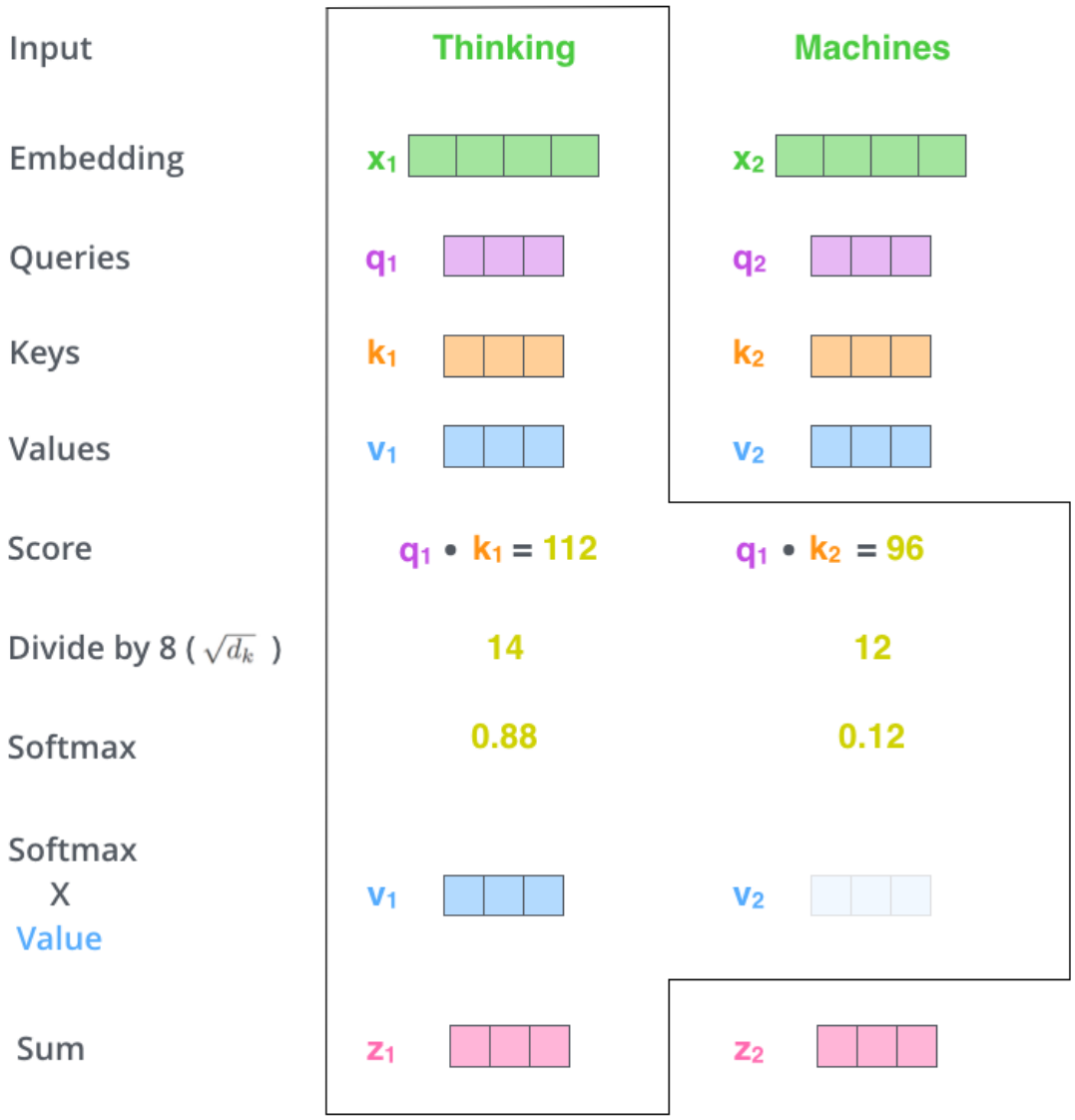


Les transformers, dès leur introduction dans le célèbre article “attention is all you need” en 2017, ont remplacé les architectures à base de cellules récurrentes en NLP. Le schéma global est similaire : un encodeur passe des vecteurs de Key et Value au décodeur afin que celui-ci applique un mécanisme d’attention encodeur-décodeur.



Mais comme on peut le constater ci-dessus sur le schéma simplifié, les couches de RNN de l'encodeur et décodeur ont été remplacées par des couches de self-Attention (des blocs multi-head attention en réalité, qui ne sont que : plusieurs self-attention appliqués à la même séquence).

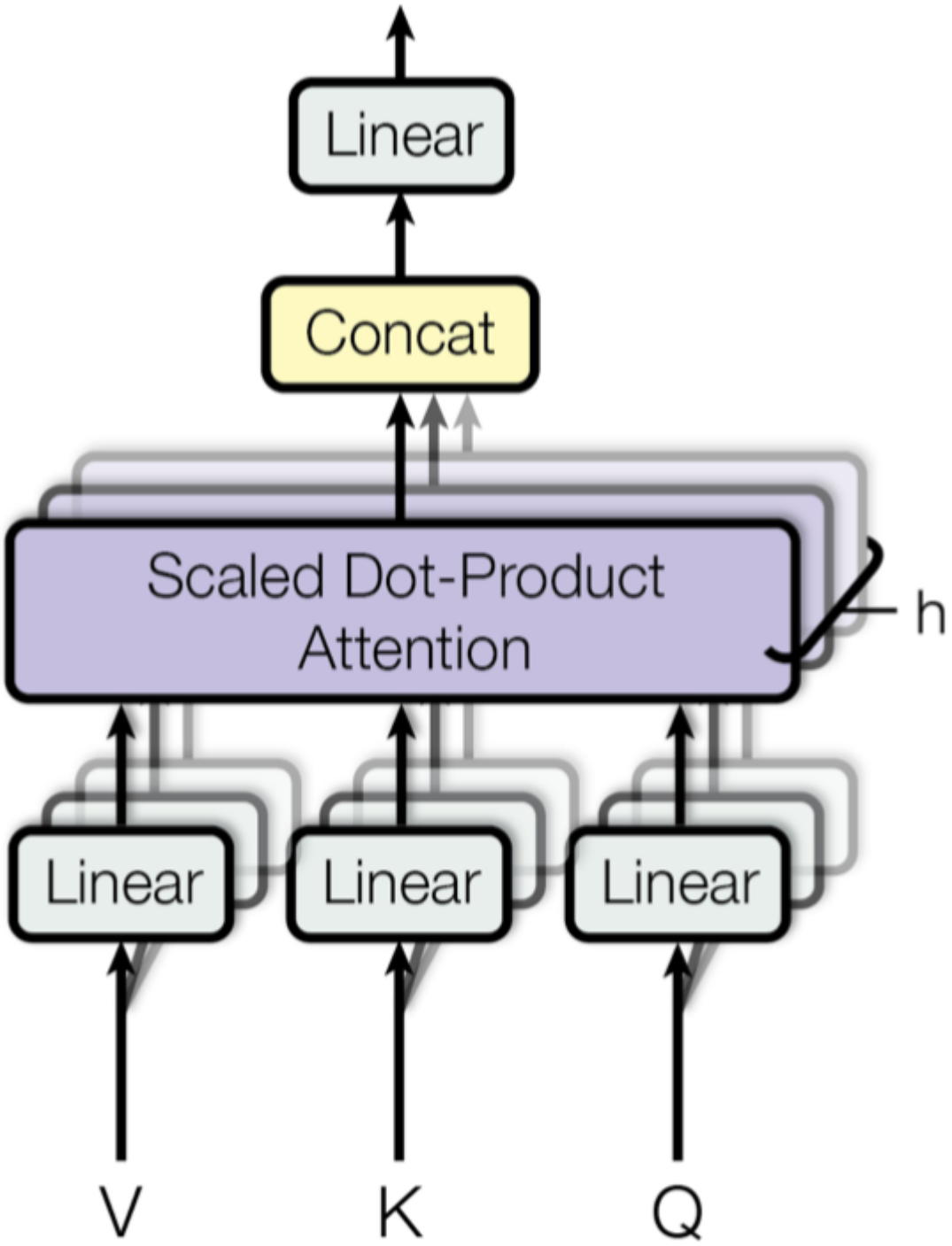
Voici le déroulement d'une couche self-attention : ce schéma fait bien apparaître le moment où les calculs des tokens s'entrecroisent pour évaluer les scores d'attention. Autrement, l'ensemble des calculs peut être réalisé en parallèle sur chaque token (et en totalité sur les autres couches du modèle).



L'opération réalisée :

$$Attention(Q, K, V) = softmax_k(\frac{QK^T}{\sqrt{d_k}})V$$

Un bloc multi-head attention se compose de plusieurs couche de self-Attention :



Il introduit de nouveaux hyperparamètres tels que :

- le nombre de fois qu'on répète le bloc
- le nombre de têtes utilisées

Dans un contexte de faible dimension comme dans l'analyse de time series ou le problème suivant, il est courant d'ajouter une autre couche "input layer" en amont de la couche positional encoding afin d'augmenter la dimension des tokens de la couche d'entrée. Cette dimension est un nouvel hyperparamètre.

D'ailleurs cette couche positional encoding est un moyen pour représenter la séquentialité des données dans le modèle, cela vient du fait que l'on n'utilise plus de RNN.

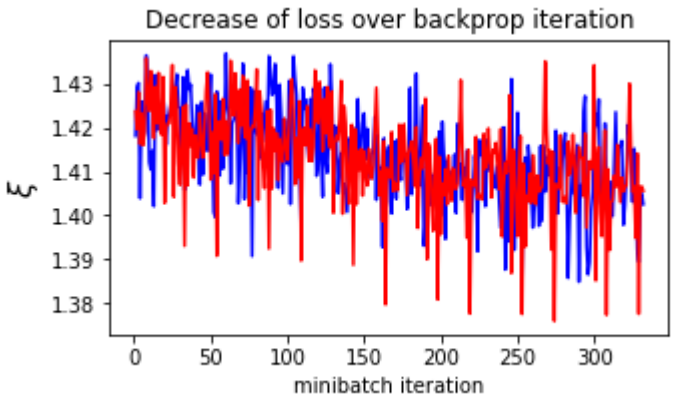
L'algorithme d'optimisation utilisé pour les transformers est celui introduit dans l'article "Attention is all you need", il s'agit d'un algorithme d'Adam avec une politique de learning rate telle que ci-dessous :

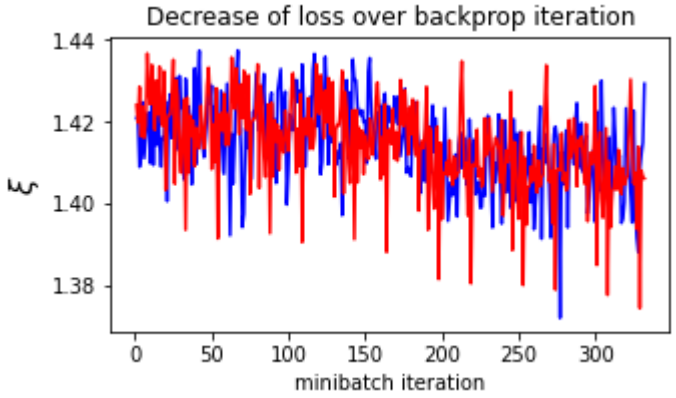
$$lrate = d_{model}^{-0.5} * min(step\_num^{-0.5}, step\_num * warmup\_steps^{-1.5})$$

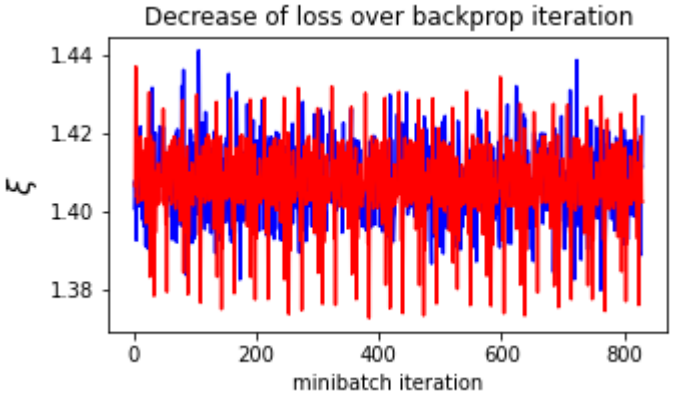
V - Tests et Résultats

A - Encodeurs Décodeurs RNN

Les tests ont été réalisés sur google colab en faisant varier les hyperparamètres énoncés ci-dessus.
Les résultats sont toujours les mêmes dans ce cas




Decrease of loss over backprop iteration
1.44
1.42
1.40
1.38
ξ
0
50
100
150
200
250
300
minibatch iteration


Decrease of loss over backprop iteration
1.44
1.42
1.40
1.38
ξ
0
200
400
600
800
minibatch iteration

```
0 [0.03714365] [0]
1 [0.04241011] [0]
2 [0.04770983] [0]
3 [0.05382562] [0]
4 [0.05855335] [0]
5 [0.06290479] [0]
6 [0.06702884] [0]
7 [0.07107183] [0]
8 [0.07516237] [0]
9 [0.07943238] [0]
10 [0.08402184] [0]
11 [0.08909728] [0]
12 [0.0948725] [0]
13 [0.10163382] [0]
14 [0.10978066] [0]
15 [0.11988232] [0]
16 [0.1327576] [0]
17 [0.14956778] [0]
18 [0.17187044] [0]
19 [0.2014592] [0]
20 [0.23963146] [0]
21 [0.28560105] [0]
22 [0.3349349] [0]
23 [0.38051108] [0]
24 [0.41650236] [0]
25 [0.4413083] [0]
26 [0.45675653] [0]
27 [0.46575257] [0]
28 [0.4707802] [0]
29 [0.47352484] [0]
```

B - Transformers

Les transformers ont quant à eux réussi à surprendre des données en passant de fonction de perte de 1,5 à 1.

VI - Nouvelles approches

Un problème majeur réside dans les dimensions des tokens qui sont représentés par des vecteurs à 4 composantes avec peu d'informations. Ces faibles dimensions s'opposent aux vecteurs de mots à des centaines voir des milliers de composantes dans le traitement du langage naturel. En effet, fournir des mots sous forme d'indice de vocabulaire d'entiers de 0 jusqu'à la taille du vocabulaire

(15 000 par exemple) à des architectures de réseaux de neurones à base de cellules récurrentes ou de self-attention est impensable.

Une idée serait donc de représenter différemment les tokens dans les données, peut-être via des traitements de deep learning préalables qui auraient pour but de plonger les tokens dans un espace vectoriel de plus grande dimension afin de capturer plus de nuances sémantiques entre tokens dans ce nouvel espace. En NLP, les méthodes modernes pour créer de tels espaces consistent à les apprendre via du self-supervised learning sur des corpus de textes pertinents pour les tâches à réaliser.