



Attention mechanism and Transformers

Clément Romic (Hugging Face & Inria)

clement.romac@inria.fr

<https://github.com/ClementRomic/Teaching/tree/main/ENSC3A LLMs 2024-2025>

Petit sondage

Que vous évoque le mot
“Transformer” ?

Contenu

- Rappels RNNs et Seq2Seq
- Attention mechanism
- Self-Attention
 - Multi-hop
 - Multi-head
- Transformer architecture

A retenir

- Limites des RNNs
- Compréhension du principe d'attention + self-attention
- Compréhension globale de l'architecture Transformer
- Intuition des très nombreux tricks (*Positional Encoding, Layer Norm, Residual connections...*)

Ressources

Cours:

- Waterloo university: https://www.youtube.com/watch?v=OyFJWRnt_AY

Talks:

- Arthur Szlam: <https://www.youtube.com/watch?v=M-HCvbdQ8wA>

Lectures:

- <https://jalammar.github.io/illustrated-transformer/>
- <https://lilianweng.github.io/posts/2018-06-24-attention/>
- <https://lilianweng.github.io/posts/2020-04-07-the-transformer-family/>
- <https://transformersbook.com/>

Rappels sur les RNNs et Seq2Seq

RNNs pour les séquences

Comment gérer les
dépendances
temporelles ?



Exemple 1 :

J' aime le ... Learning

Exemple 2 :

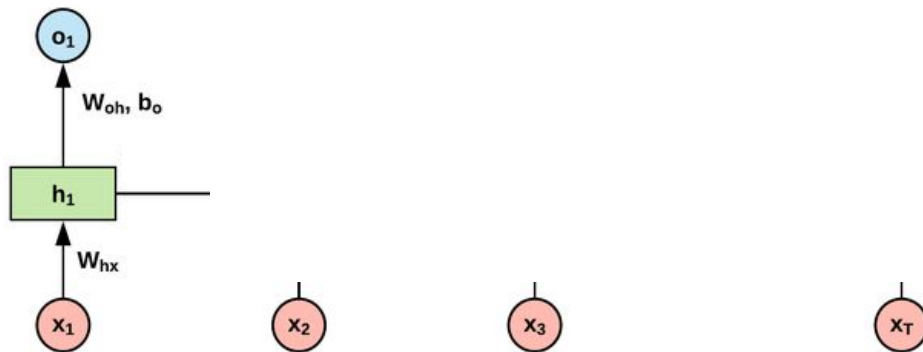
[02, 10, 2015, 1500] [03, 10, 2015, 1205] [04, 10, 2015, 1820] ... [05, 10, 2015, 1900]

Jour, mois, année, valeur

RNNs pour les séquences

Pour un élément:

- 1) Représentation (hidden state)
- 2) Sortie

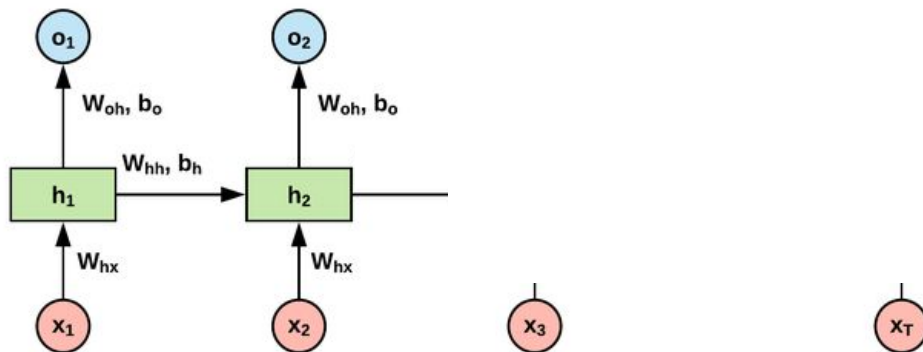


- **hidden state** $h_1 = \sigma_h(W_{hx} x_1)$
- **sortie** $o_1 = \sigma_o(W_{oh} h_1 + b_o)$

RNNs pour les séquences

Pour un élément:

- 1) Représentation (hidden state)
- 2) Sortie



- **hidden state**

$$h_1 = \sigma_h(W_{hx}x_1)$$

- **sortie**

$$o_1 = \sigma_o(W_{oh}h_1 + b_o)$$

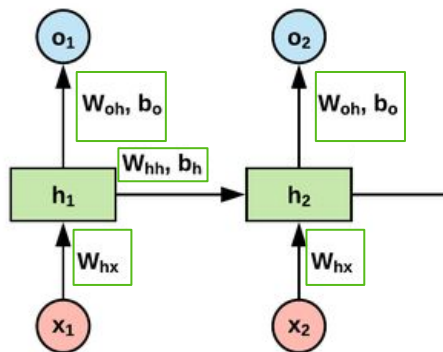
$$h_2 = \sigma_h(W_{hx}x_2 + W_{hh}h_1 + b_h)$$

$$o_1 = \sigma_o(W_{oh}h_1 + b_o)$$

RNNs pour les séquences

Pour un élément:

- 1) Représentation (hidden state)
- 2) Sortie



On réutilise le même réseau !

- hidden state

$$h_1 = \sigma_h(W_{hx}x_1)$$

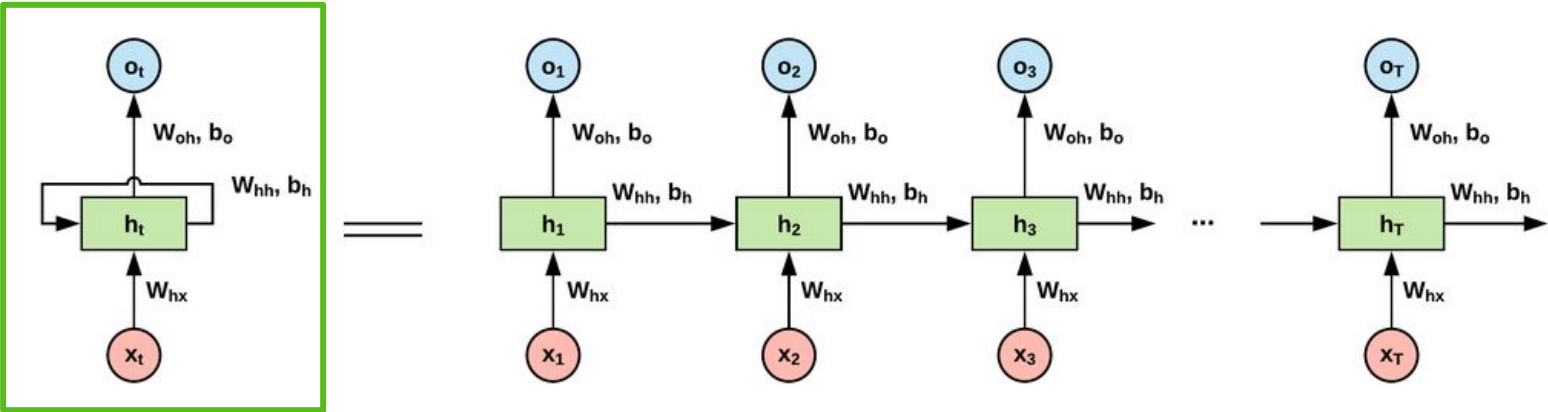
- sortie

$$o_1 = \sigma_o(W_{oh}h_1 + b_o)$$

$$h_2 = \sigma_h(W_{hx}x_2 + W_{hh}h_1 + b_h)$$

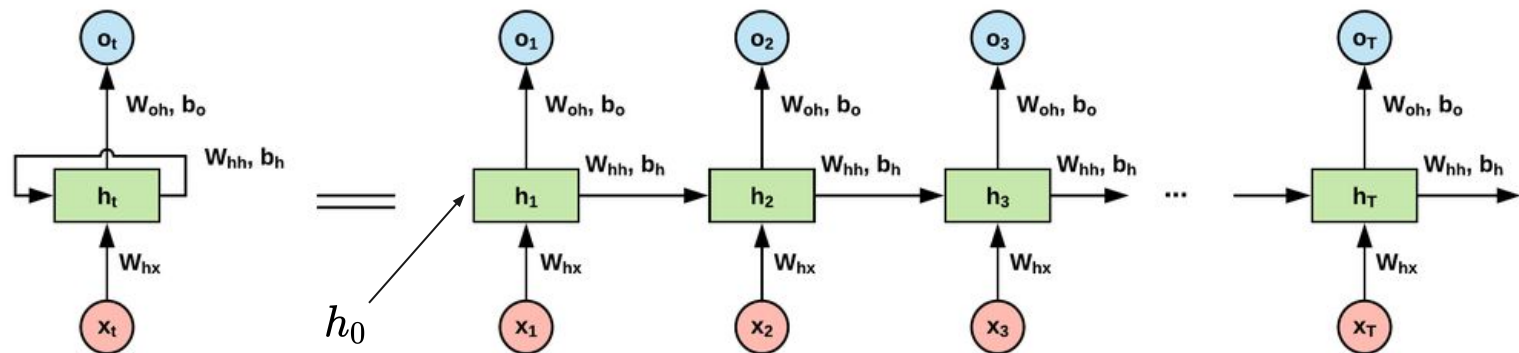
$$o_1 = \sigma_o(W_{oh}h_1 + b_o)$$

RNNs pour les séquences



- hidden state $h_1 = \sigma_h(W_{hx}x_1)$ | $h_2 = \sigma_h(W_{hx}x_2 + W_{hh}h_1 + b_h)$
- sortie $o_1 = \sigma_o(W_{oh}h_1 + b_o)$ | $o_1 = \sigma_o(W_{oh}h_1 + b_o)$

RNNs pour les séquences



- hidden state

$$h_1 = \sigma_h(W_{hx}x_1 + W_{hh}h_0 + b_h)$$

- sortie

$$o_1 = \sigma_o(W_{oh}h_1 + b_o)$$

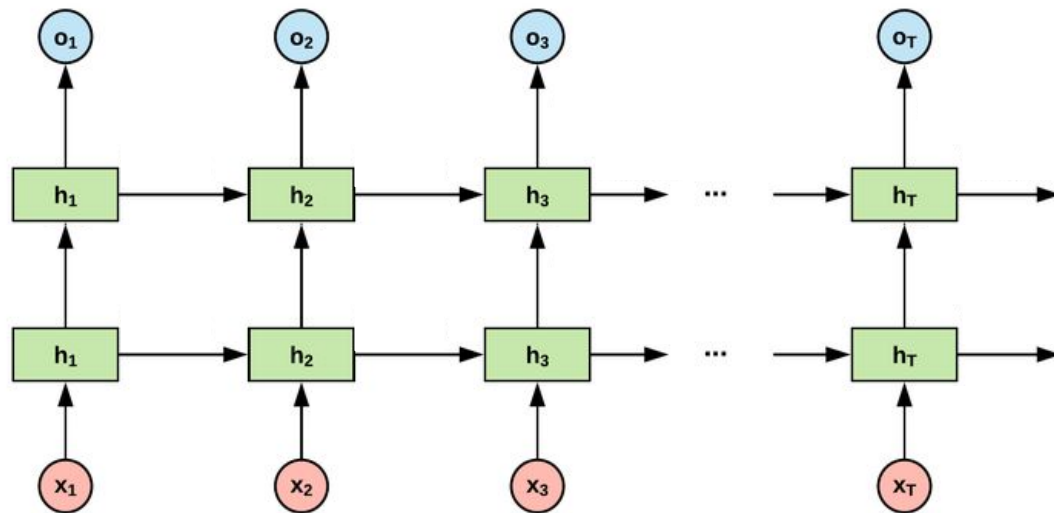
$$h_2 = \sigma_h(W_{hx}x_2 + W_{hh}h_1 + b_h)$$

$$o_1 = \sigma_o(W_{oh}h_1 + b_o)$$

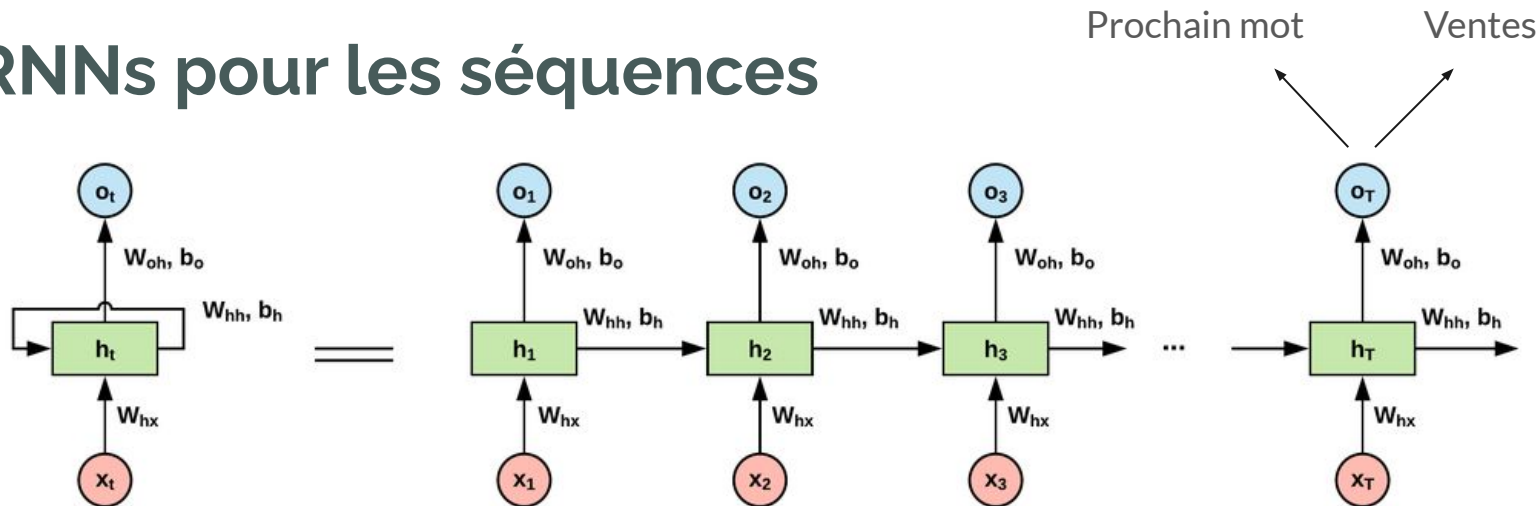
RNNs pour les séquences

On peut enchaîner les couches:

- Comme dans un Perceptron, on peut approximer des **fonctions plus complexes**
- Chaque couche a ses poids



RNNs pour les séquences



Exemple 1:

J' aime le ... Learning

Example 2:

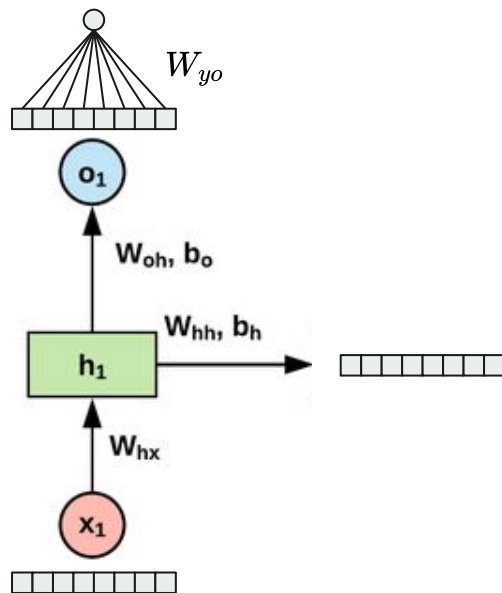
[02, 10, 2015, 1500] [03, 10, 2015, 1205] [04, 10, 2015, 1820] ... [05, 10, 2015, 1900]

Journal, month, year, value

RNNs pour les séquences

$$y_1 \in \mathbb{R}$$

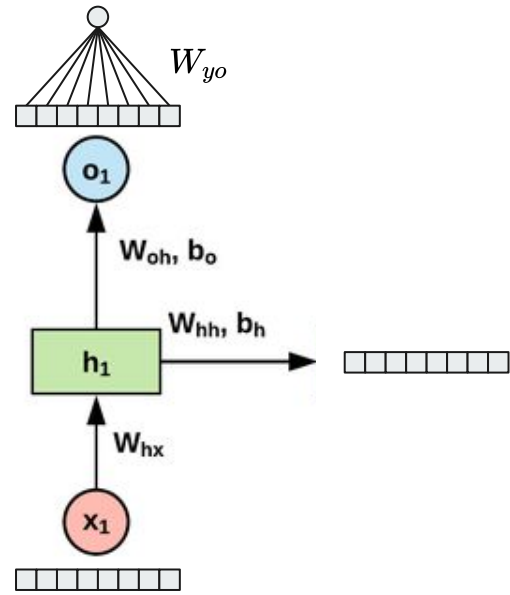
$$y_1 = W_{yo} o_1$$



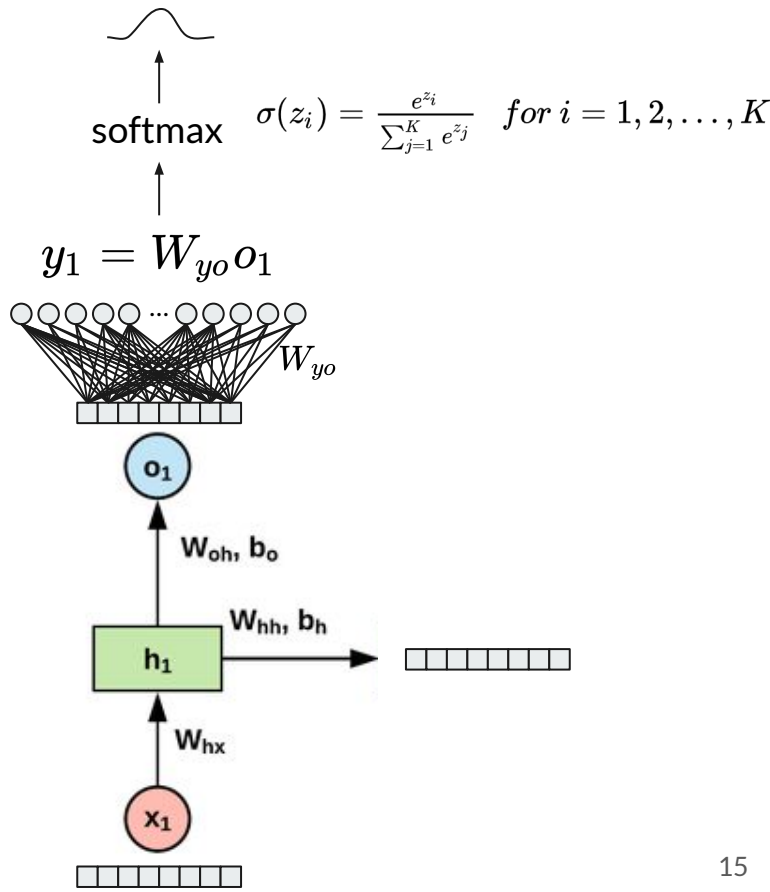
RNNs pour les séquences

$y_1 \in \mathbb{R}$

$$y_1 = W_{yo} o_1$$



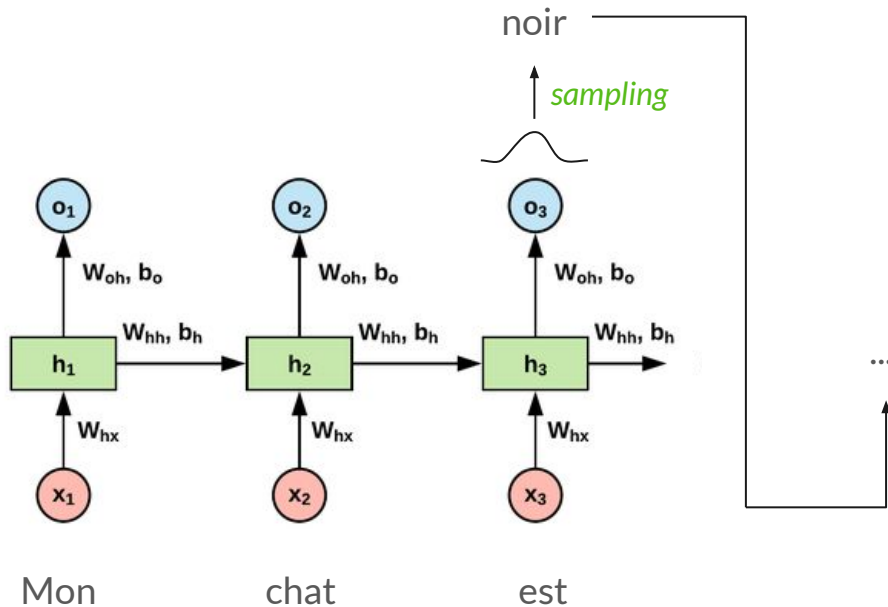
$y_1 \in \mathcal{Y} = \{u_0, u_1, \dots, u_N\}$



RNNs pour les séquences

Utilisation à l'inférence:

- Prédiction du prochain timestep
- Réutilisation des prédictions
- Accumulation des erreurs



(Causal) Language Modeling

Given a corpus of tokens: $U = \{u_1, \dots, u_N\}$

$$\max_{\theta} L(U) = \sum_i \log P_{\theta}(u_i | u_{i-k}, \dots, u_{i-1})$$

model

context window

(Causal) Language Modeling

En pratique:

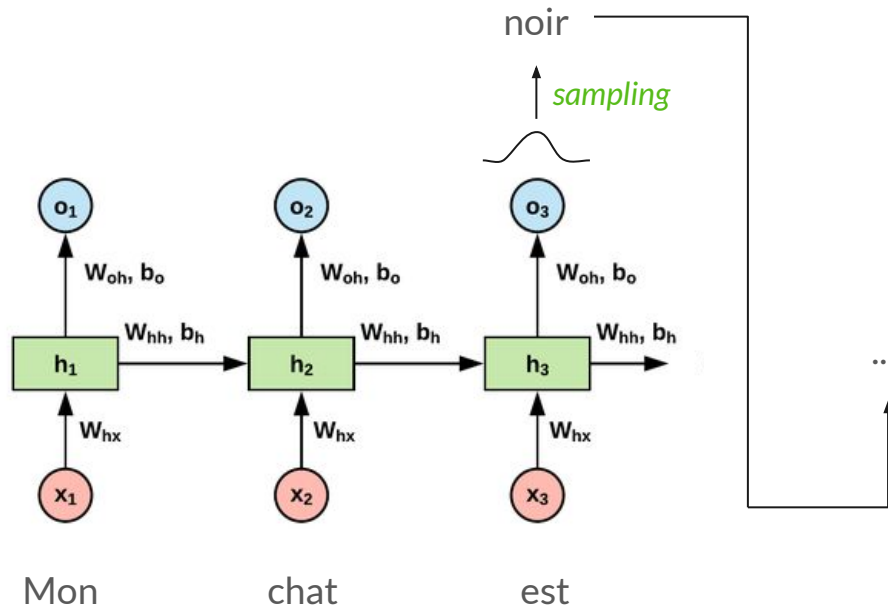
- Etant donné un **corpus de texte tokenisé**
- On casse le corpus en **blocs de taille k**
- On souhaite apprendre un **modèle** qui **maximise la probabilité de chaque token** d'apparaître dans sa séquence

RNNs pour les séquences

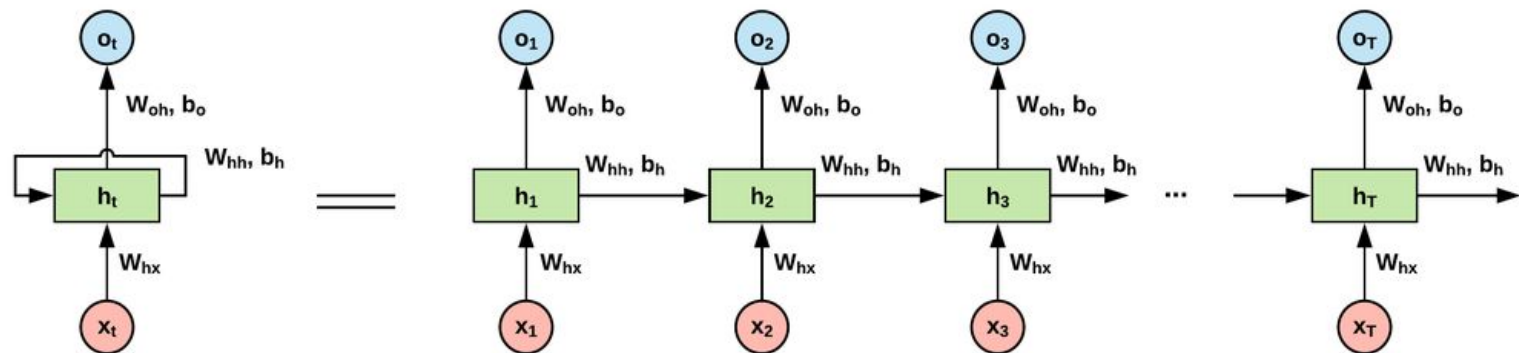
Utilisation à l'inférence:

- Prédiction du prochain timestep
- Réutilisation des prédictions
- Accumulation des erreurs

La génération de texte dans les LLMs utilise toujours cette méthode !



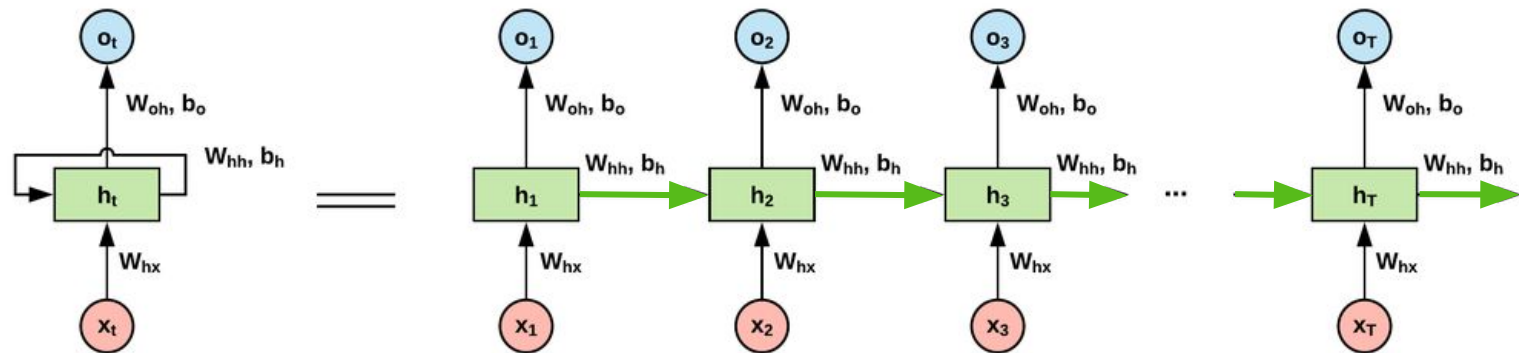
RNNs pour les séquences



Inconvénients:

- Dépendances long terme
- Gradient vanishing
- Impossible de paralléliser

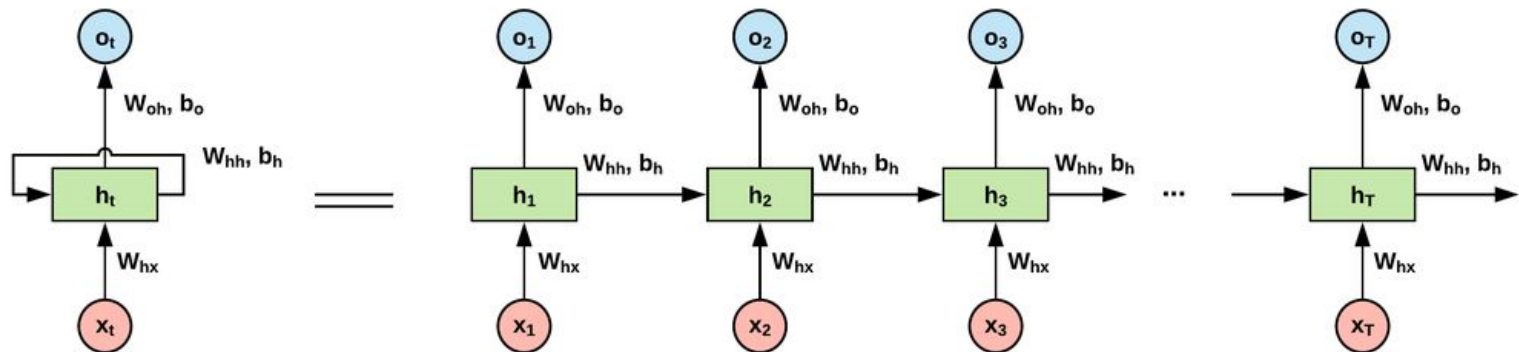
RNNs pour les séquences



Inconvénients:

- **Dépendances long terme => synthétisation de l'information dans h**
- Gradient vanishing
- Impossible de paralléliser

RNNs pour les séquences



Inconvénients:

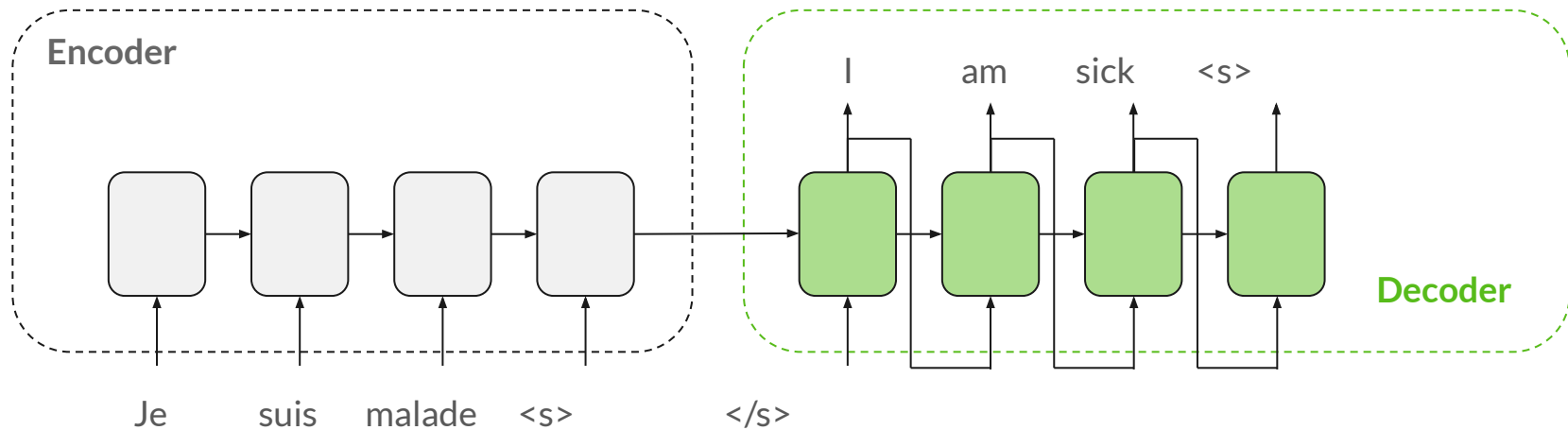
- Dépendances long terme
 - Gradient vanishing
 - Impossible de paralléliser
- } → Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997)

Le problème Seq2Seq

Une séquence en entrée => une séquence en sortie

On apprend à prédire seulement la séquence de sortie.

Exemple: la traduction



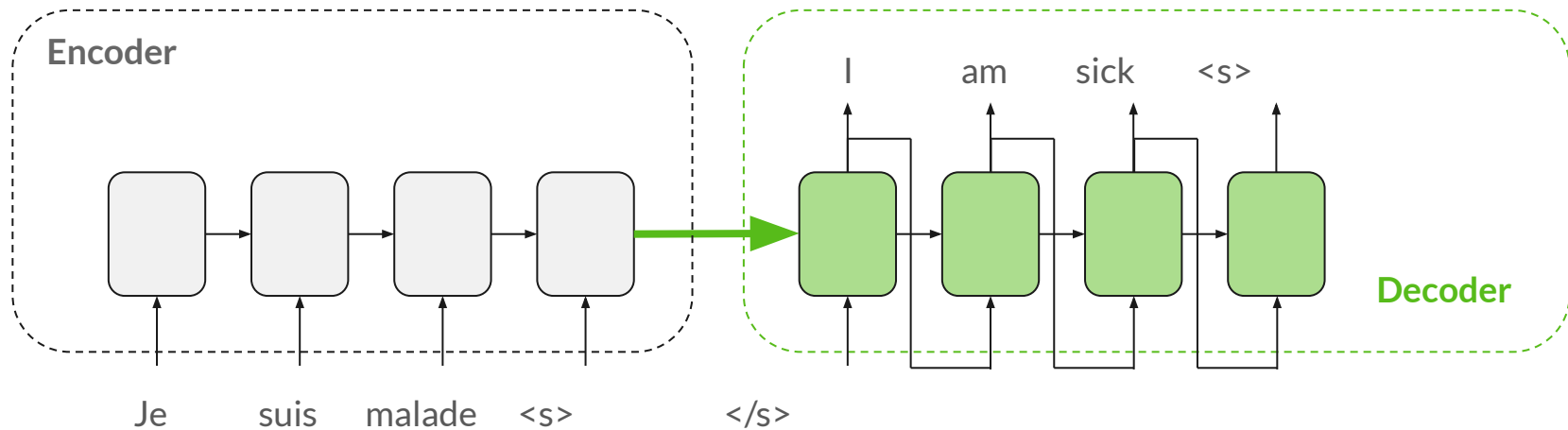
Le problème Seq2Seq

Une séquence en entrée => une séquence en sortie

On apprend à prédire seulement la séquence de sortie.

Le dernier hidden state doit représenter toute la séquence d'entrée /\

Exemple: la traduction





TP: Partie 2

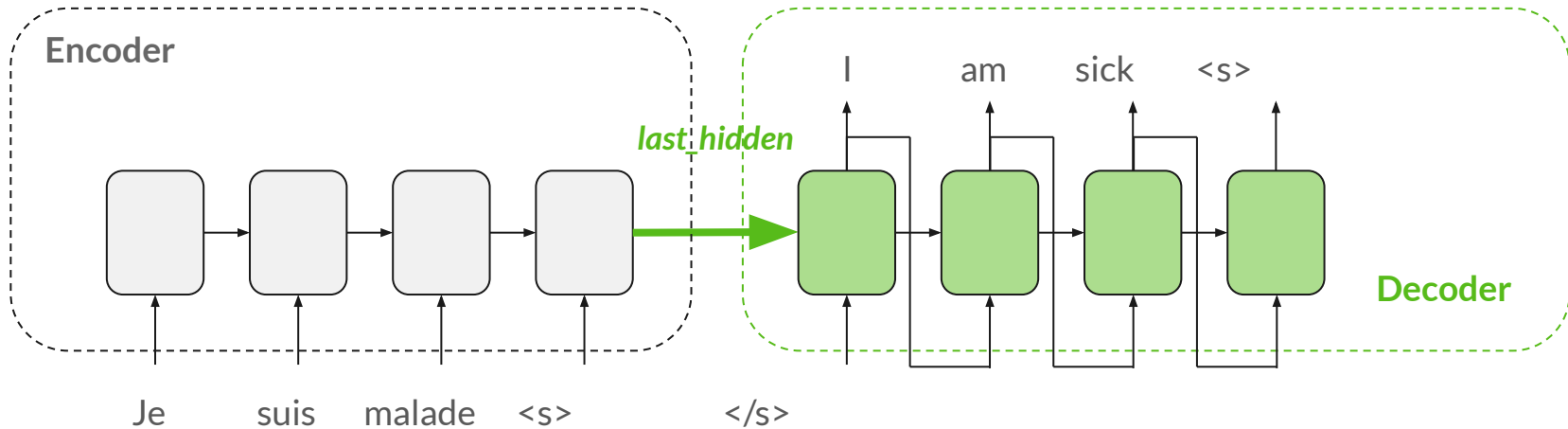
Le problème Seq2Seq

Une séquence en entrée => une séquence en sortie

On apprend à prédire seulement la séquence de sortie.

Le dernier hidden state doit représenter toute la séquence d'entrée /\

Exemple: la traduction



Attention mechanism

Attention mechanism

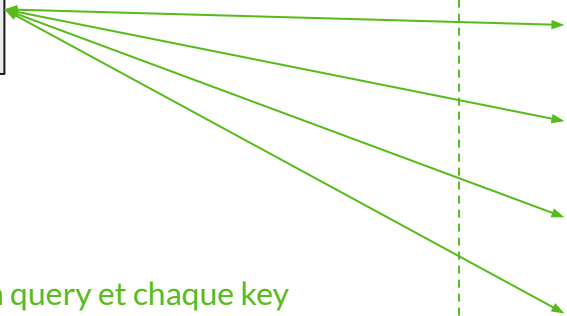
Imitation d'un principe de retrieval



| Database | |
|----------|---------|
| Key | Value |
| Key_1 | Value_1 |
| Key_2 | Value_2 |
| Key_3 | Value_3 |
| Key_4 | Value_4 |
| ... | ... |

Attention mechanism

Imitation d'un principe de retrieval



Database

| Key | Value |
|-------|---------|
| Key_1 | Value_1 |
| Key_2 | Value_2 |
| Key_3 | Value_3 |
| Key_4 | Value_4 |
| ... | ... |

1. Calcul d'une "similarité" entre la query et chaque key

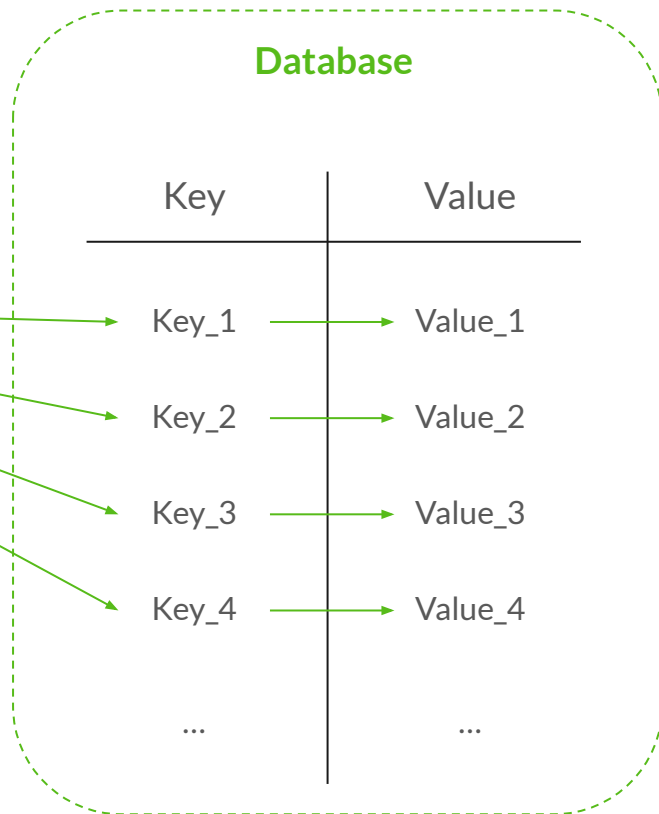
Attention mechanism

Imitation d'un principe de retrieval



1. Calcul d'une "similarité" entre la query et chaque key
2. "Utilisation" de cette similarité sur chaque value

$$attention(q, K, V) = \sum_i similarity(q, k_i) \times v_i$$

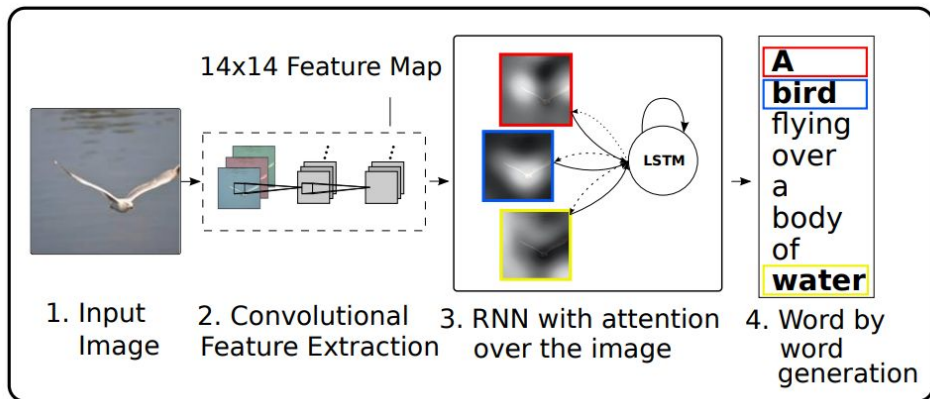


Attention mechanism

Bref historique

Vision:

- Mnih et al., 2014 => Sequence of focusing (RL)
- Xu et al., 2015 => Captioning with attention on feature maps



Attention mechanism

Bref historique

Vision:

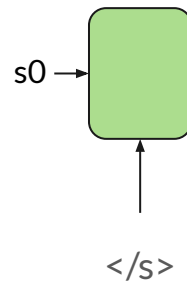
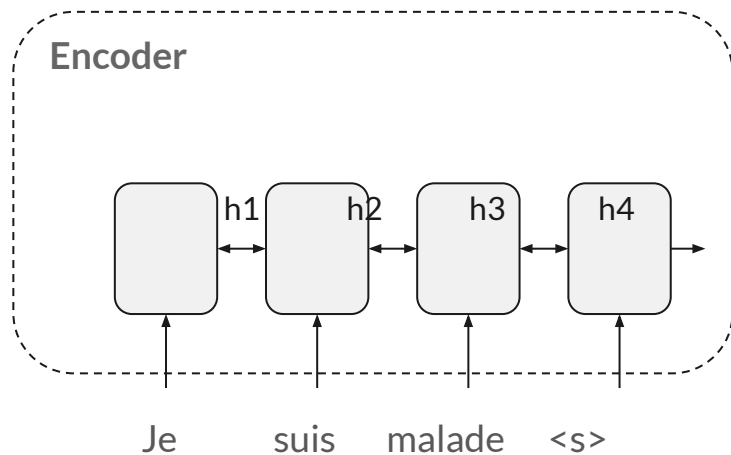
- Mnih et al., 2014 => Sequence of focusing (RL)
- Xu et al., 2015 => Captioning with attention on feature maps

NLP:

- Brown et al., 1993 => Alignment in translation (Hard attention)
- Bahdanau et al., 2015 => Attention as focusing (SOTA in NMT /!\)

Additive Attention *(Bahdanau et al., 2015)*

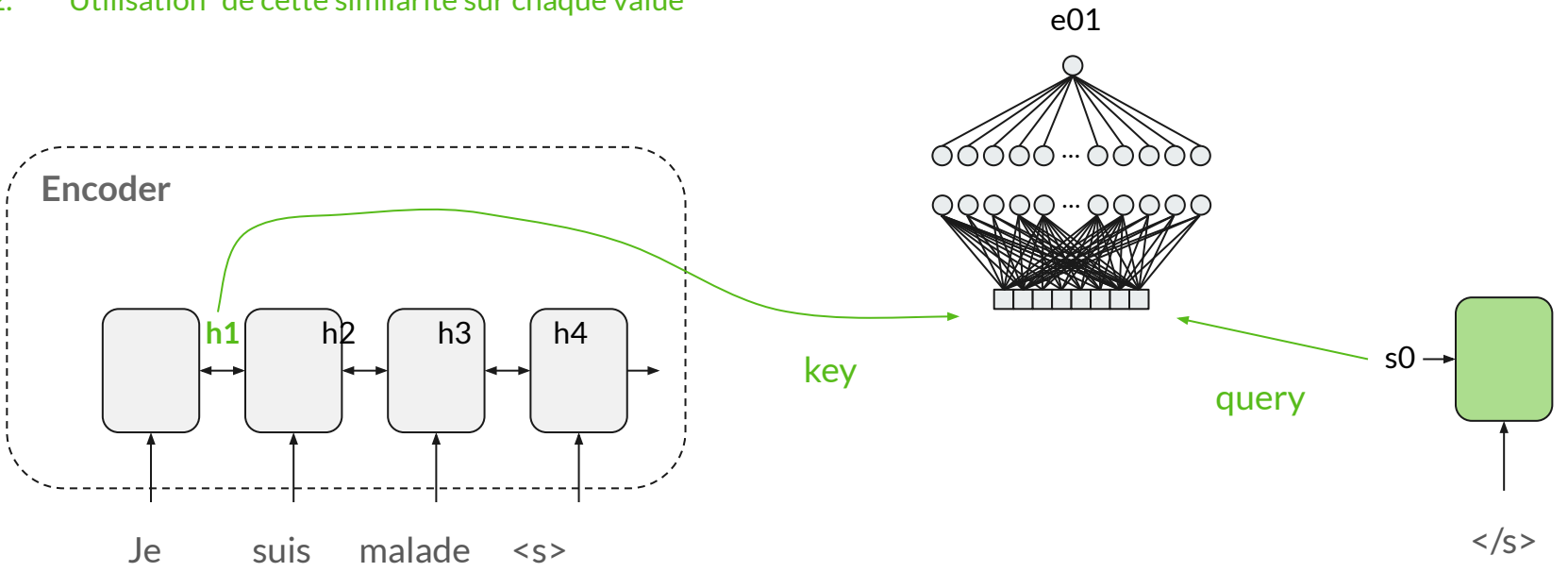
1. Calcul d'une "similarité" entre la query et chaque key
2. "Utilisation" de cette similarité sur chaque value



Additive Attention (Bahdanau et al., 2015)

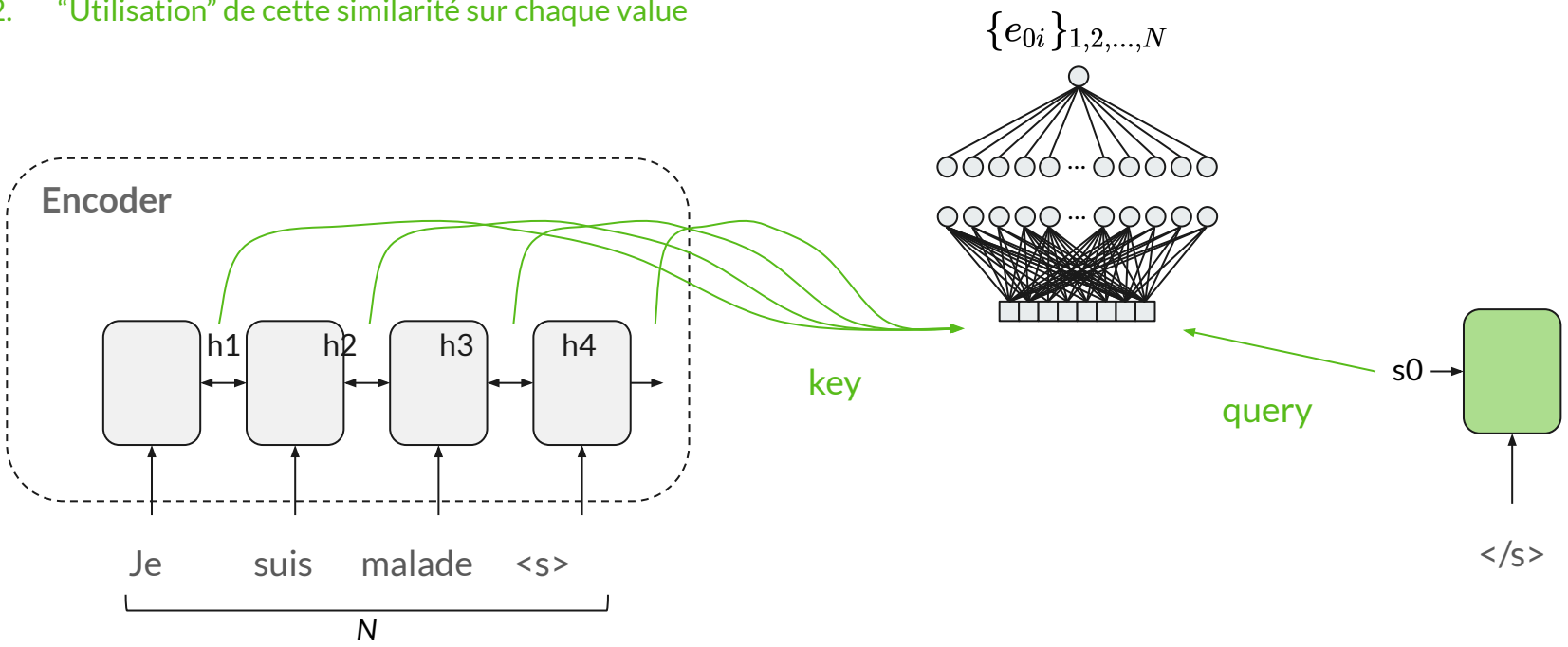
- 1. Calcul d’une “similarité” entre la query et chaque key
- 2. “Utilisation” de cette similarité sur chaque value

$$\text{score}(s_t, h_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[s_{t-1}; h_i])$$



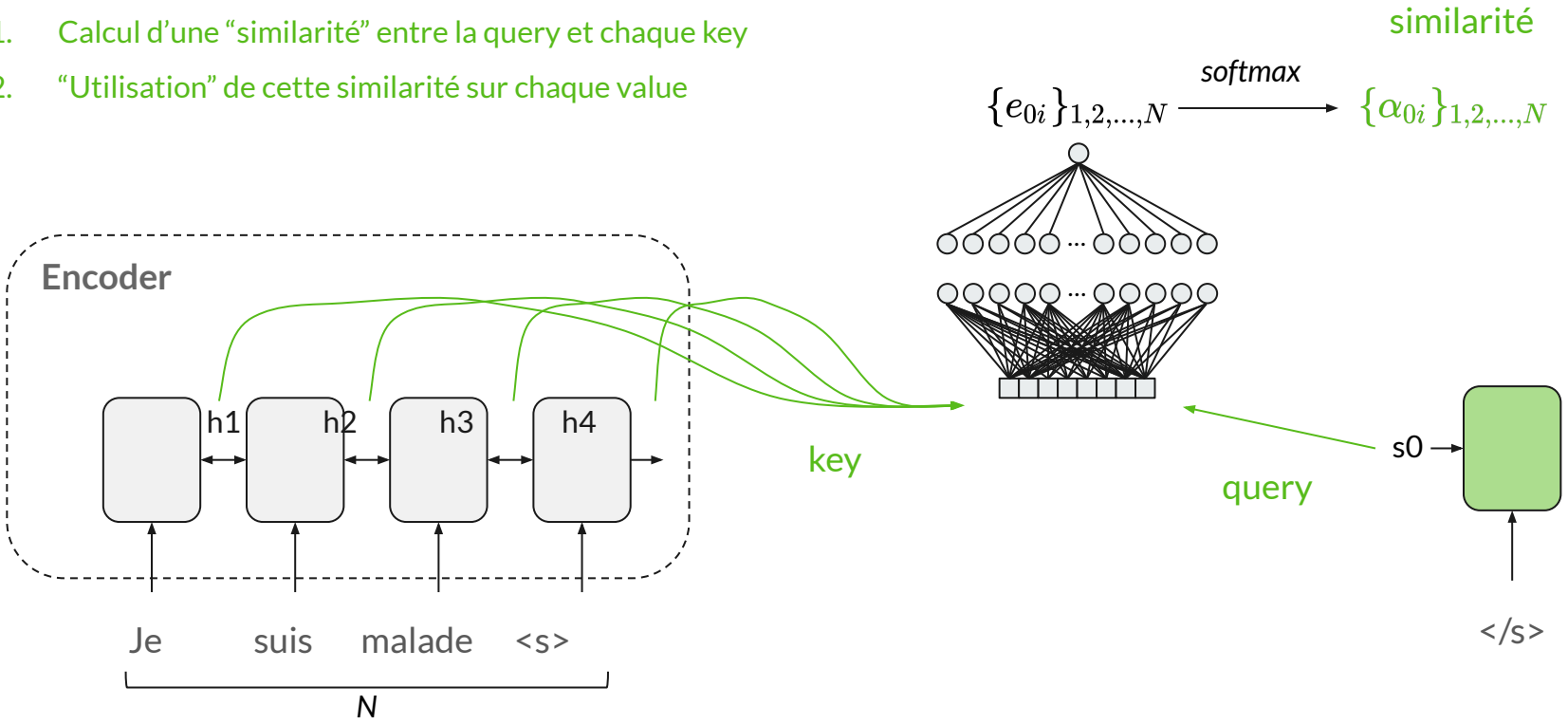
Additive Attention (Bahdanau et al., 2015)

- 1. Calcul d’une “similarité” entre la query et chaque key
- 2. “Utilisation” de cette similarité sur chaque value



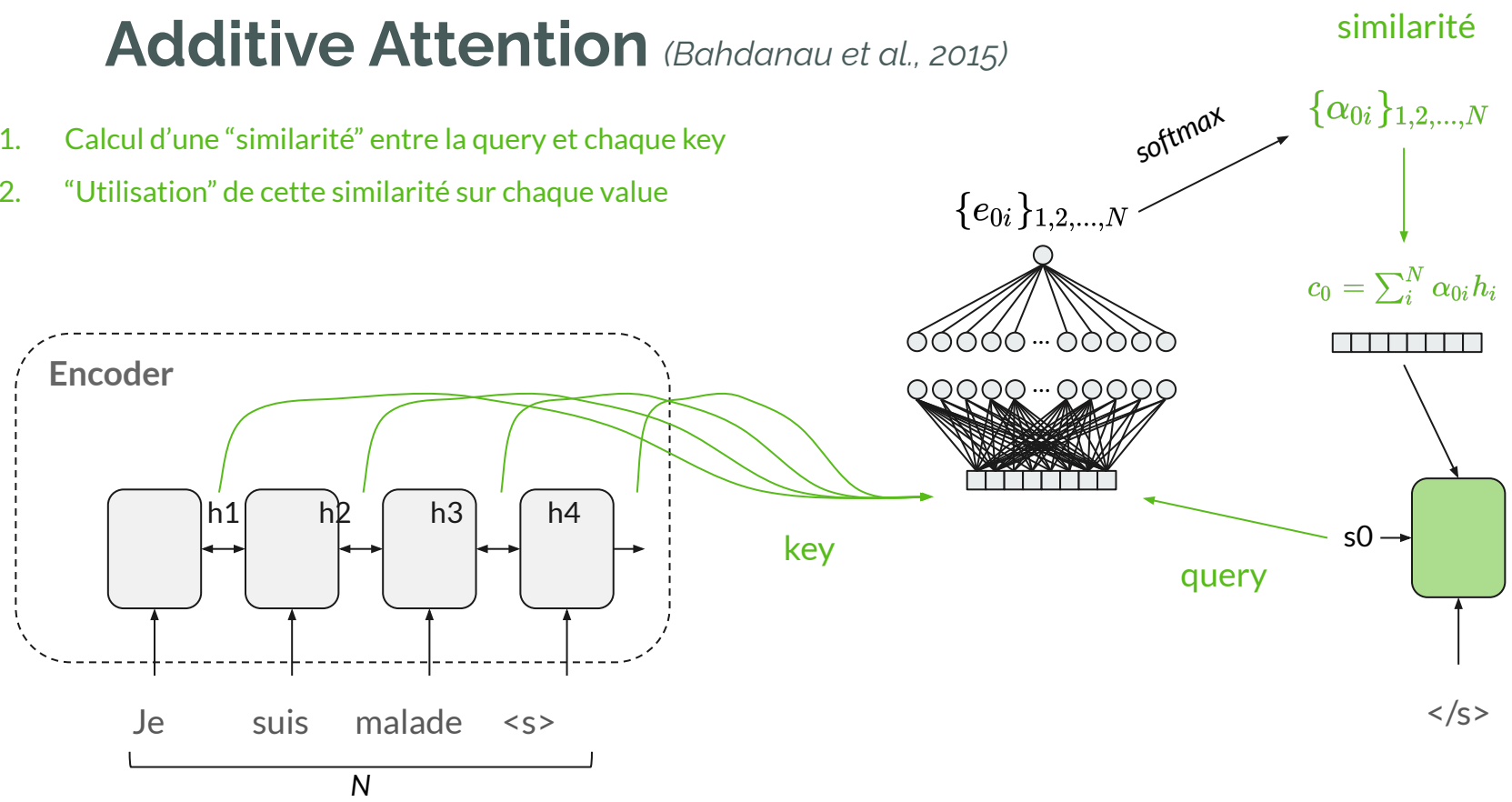
Additive Attention (Bahdanau et al., 2015)

- 1. Calcul d’une “similarité” entre la query et chaque key
- 2. “Utilisation” de cette similarité sur chaque value

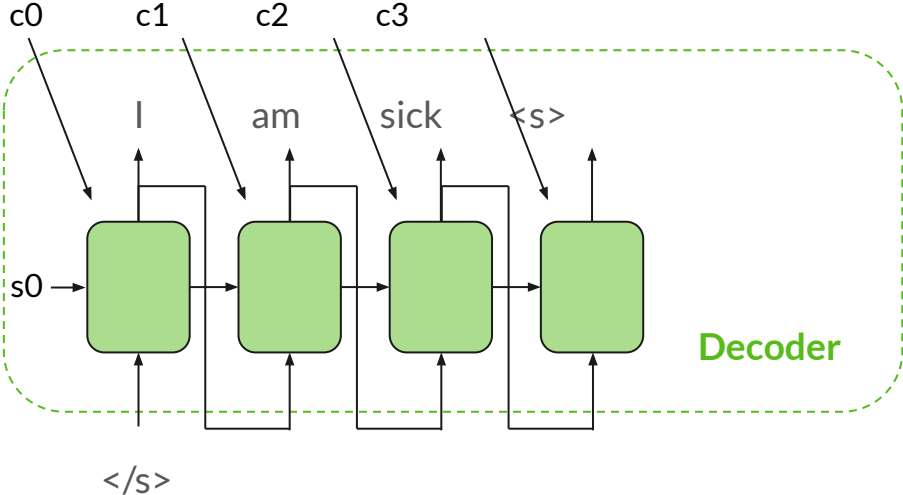
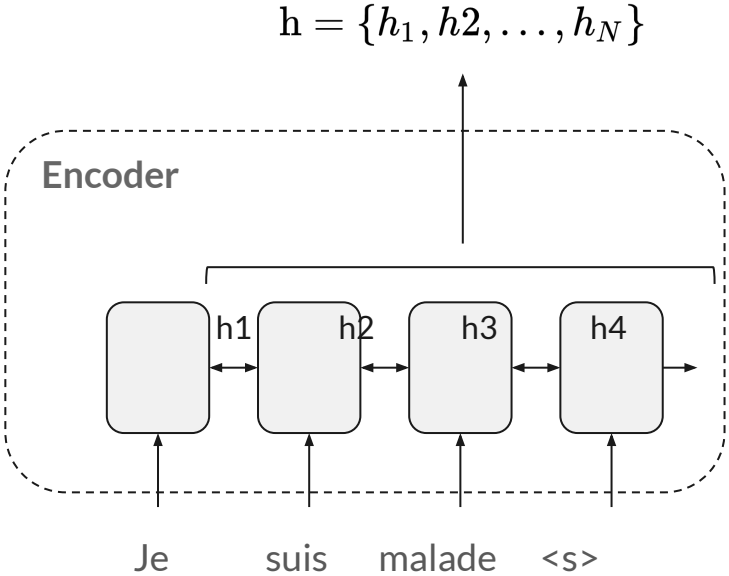


Additive Attention (Bahdanau et al., 2015)

- 1. Calcul d’une “similarité” entre la query et chaque key
- 2. “Utilisation” de cette similarité sur chaque value

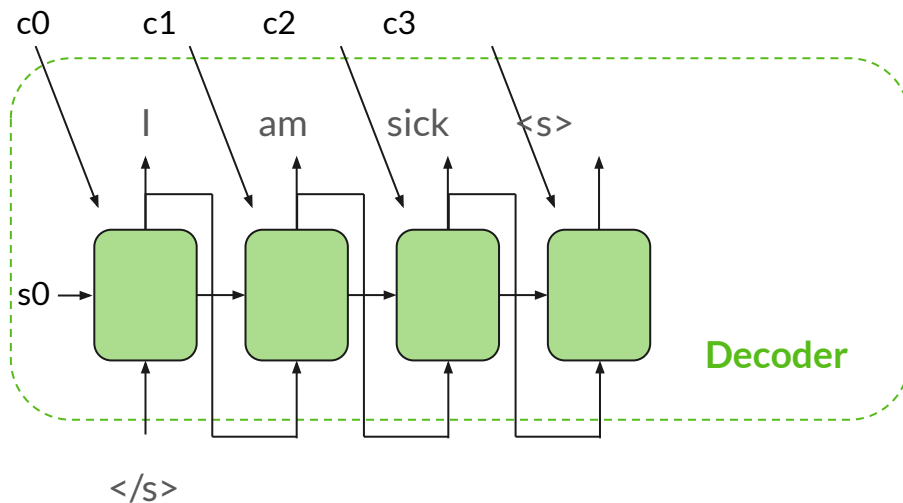
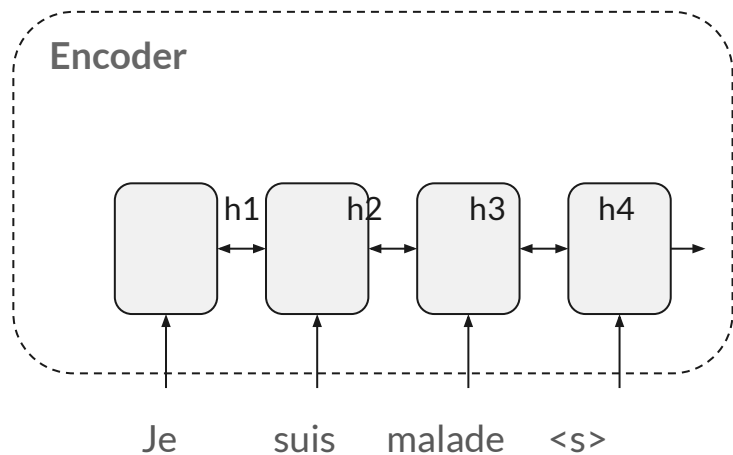


Additive Attention *(Bahdanau et al., 2015)*

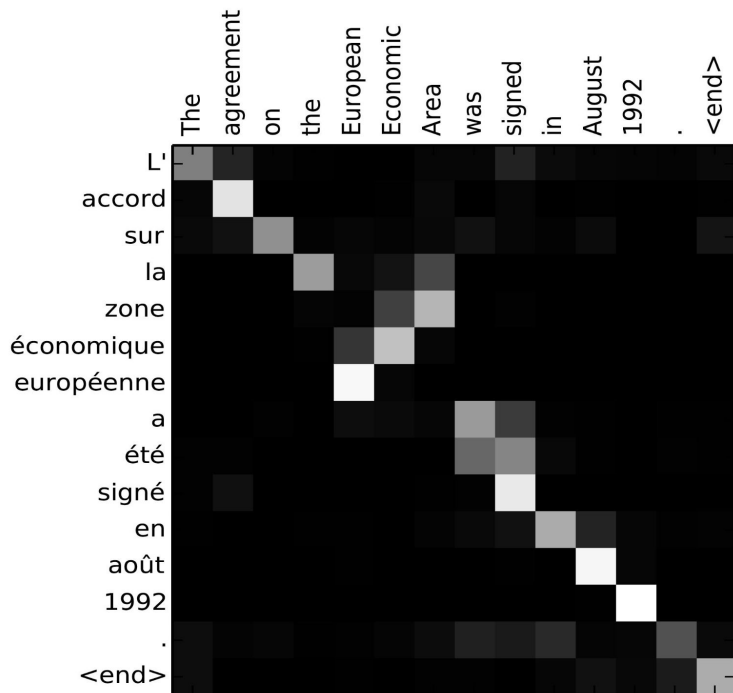


Additive Attention *(Bahdanau et al., 2015)*

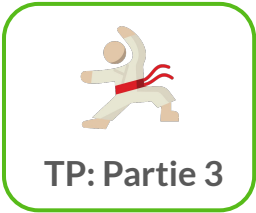
- Il y a désormais un contexte résumant la phrase d'entrée pour chaque mot du Decoder
- Chaque mot décodé a son propre contexte (query = $s-1$)
- Les hidden states de l'Encoder sont à la fois les clés et valeurs, ceux du Decoder les queries



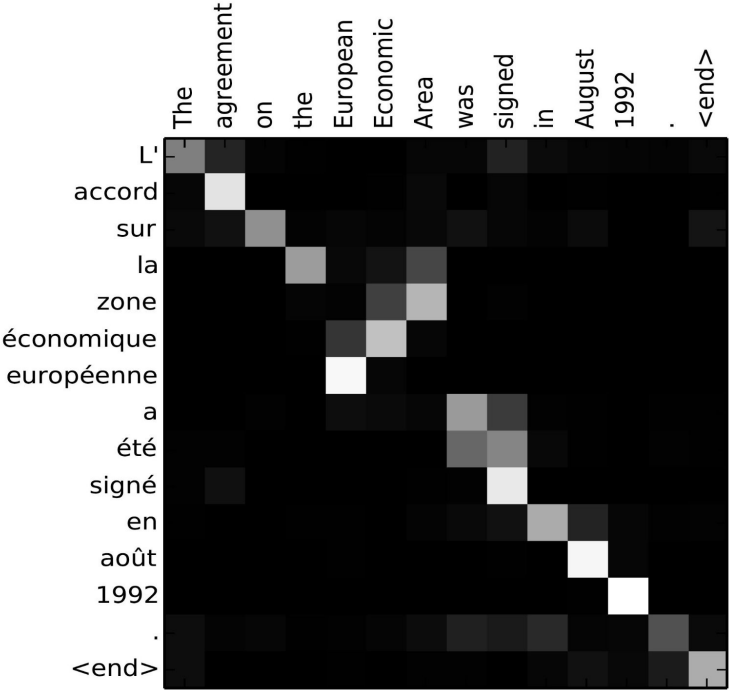
Additive Attention *(Bahdanau et al., 2015)*



L'attention permet à chaque mot décodé de se "concentrer" sur certains mots d'entrée



Additive Attention *(Bahdanau et al., 2015)*



L'attention permet à chaque mot décodé de se "concentrer" sur certains mots d'entrée

Attention mechanisms (<https://lilianweng.github.io/posts/2018-06-24-attention/>)

| Name | Alignment score function | Citation |
|------------------------|---|------------------------------|
| Content-base attention | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \text{cosine}[\mathbf{s}_t, \mathbf{h}_i]$ | Graves2014 |
| Additive(*) | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_{t-1}; \mathbf{h}_i])$ | Bahdanau2015 |
| Location-Base | $\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position. | Luong2015 |
| General | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$ where \mathbf{W}_a is a trainable weight matrix in the attention layer. | Luong2015 |
| Dot-Product | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{h}_i$ | Luong2015 |
| Scaled Dot-Product(^) | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state. | Vaswani2017 |

(*) Referred to as “concat” in Luong, et al., 2015 and as “additive attention” in Vaswani, et al., 2017.

(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

Attention mechanisms (<https://lilianweng.github.io/posts/2018-06-24-attention/>)

| Name | Alignment score function | Citation |
|------------------------|---|-------------------------------------|
| Content-base attention | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \text{cosine}[\mathbf{s}_t, \mathbf{h}_i]$ | <u>Graves2014</u> |
| Additive(*) | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_{t-1}; \mathbf{h}_i])$ | <u>Bahdanau2015</u> |
| Location-Base | $\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position. | <u>Luong2015</u> |
| General | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$ where \mathbf{W}_a is a trainable weight matrix in the attention layer. | <u>Luong2015</u> |
| Dot-Product | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{h}_i$ | <u>Luong2015</u> |
| Scaled Dot-Product(^) | $\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state. | <u>Vaswani2017</u> |

(*) Referred to as “concat” in Luong, et al., 2015 and as “additive attention” in Vaswani, et al., 2017.

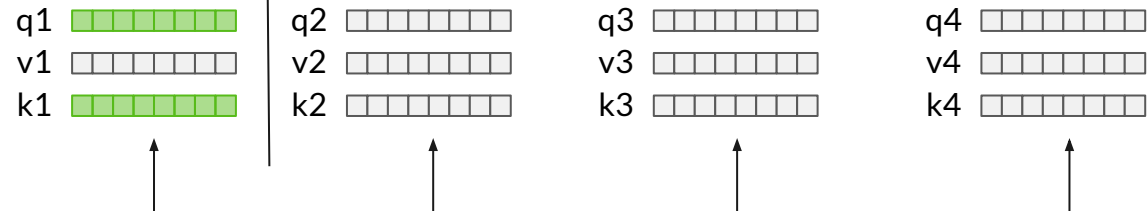
(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

Self-Attention

Self-attention basics

Q, K, V viennent du même ensemble

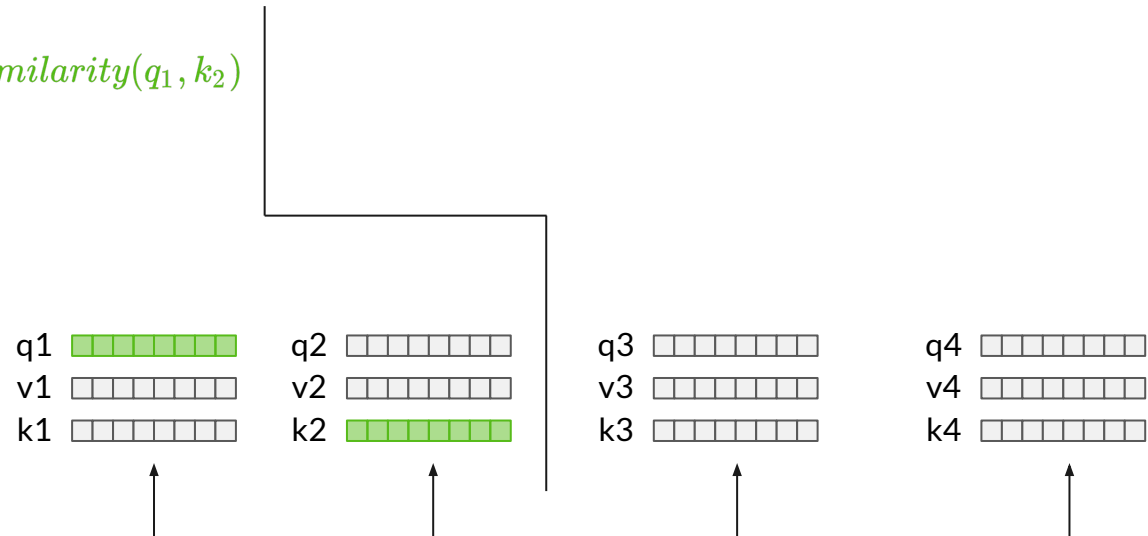
$a_{11} = \textit{similarity}(q_1, k_1)$



Self-attention basics

Q, K, V viennent du même ensemble

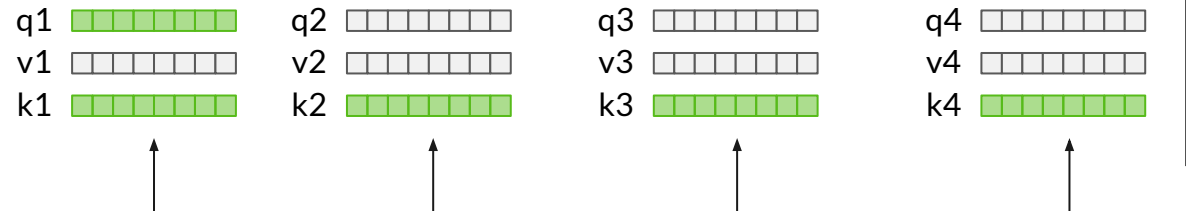
$a_{12} = \textit{similarity}(q_1, k_2)$



Self-attention basics

Q, K, V viennent du même ensemble

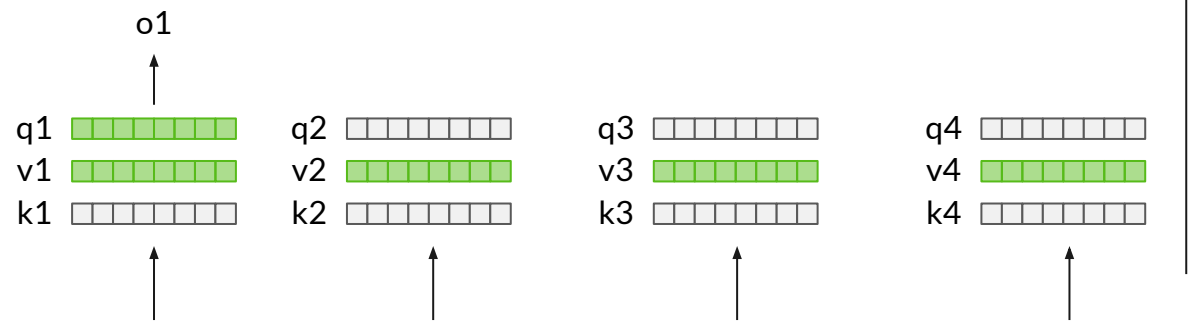
$\{a_{11}, a_{12}, \dots, a_{1N}\}$
 $a_{1j} = \textit{similarity}(q_1, k_{2j})$



Self-attention basics

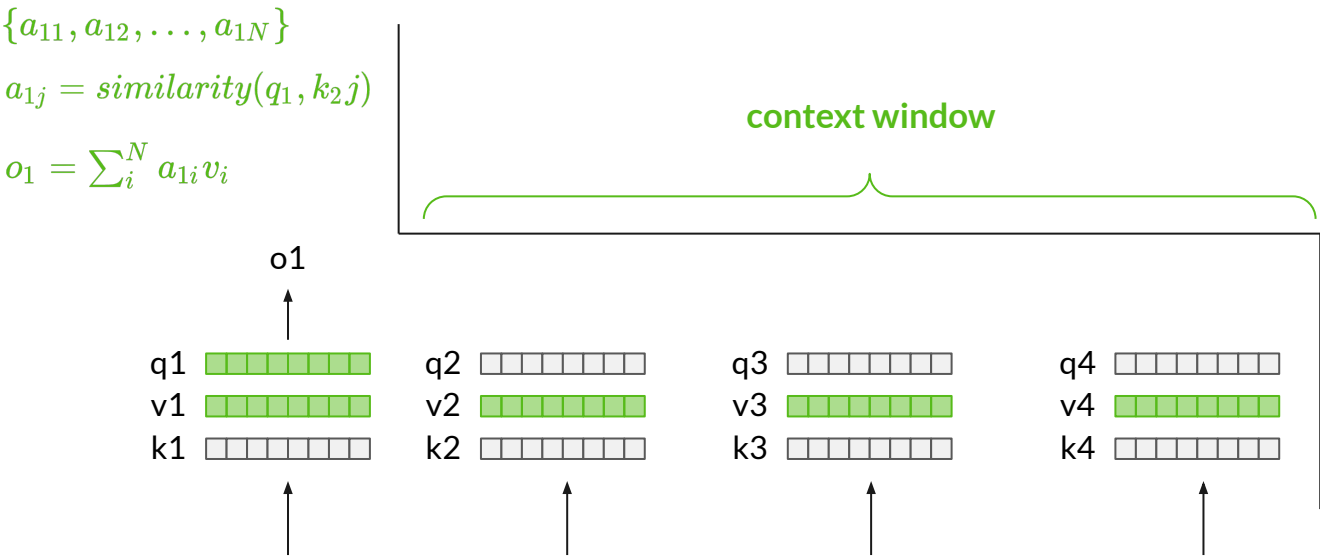
Q, K, V viennent du même ensemble

$\{a_{11}, a_{12}, \dots, a_{1N}\}$
 $a_{1j} = \text{similarity}(q_1, k_{2j})$
 $o_1 = \sum_i^N a_{1i} v_i$



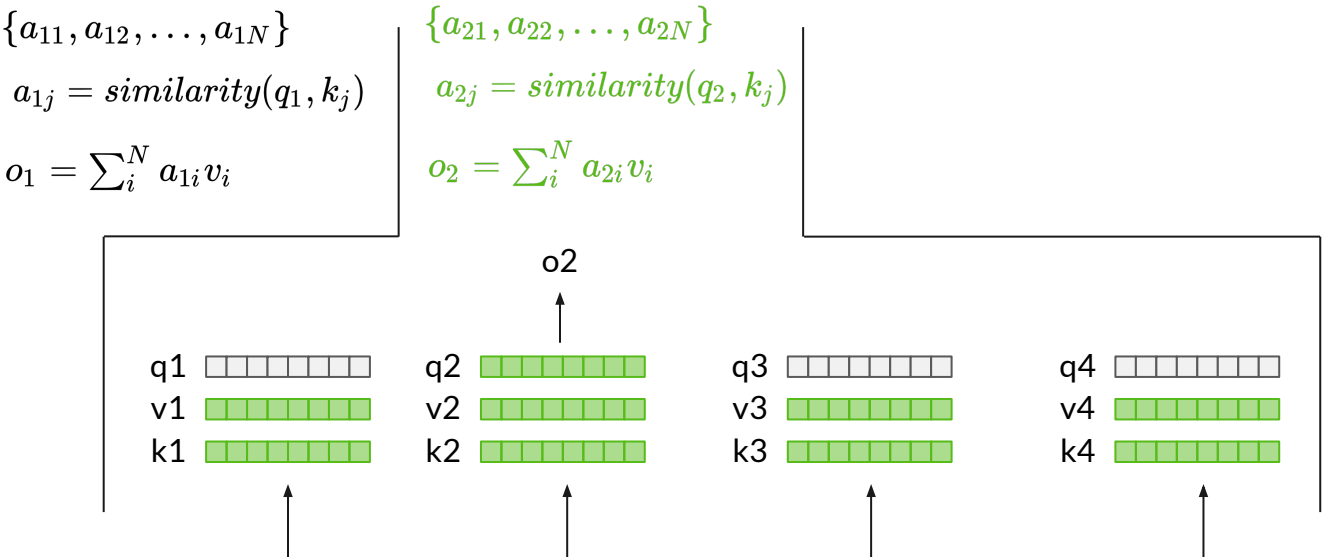
Self-attention basics

Q, K, V viennent du même ensemble



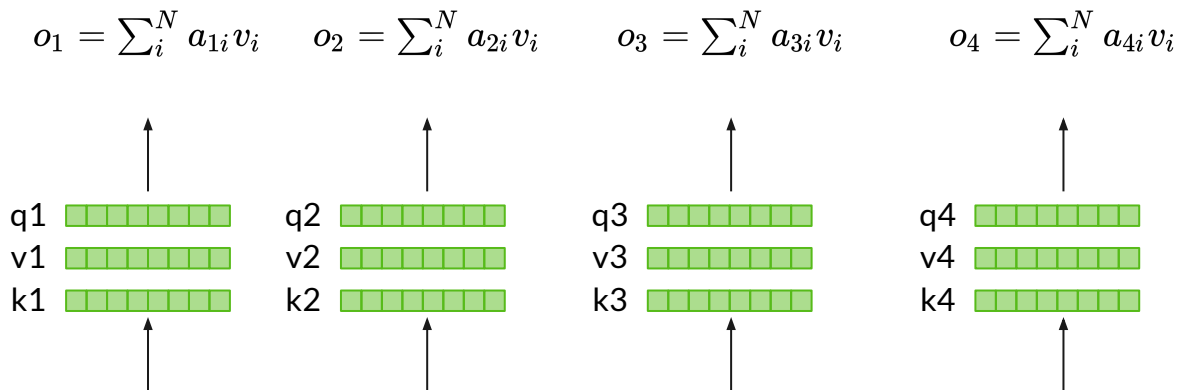
Self-attention basics

Q, K, V viennent du même ensemble



Self-attention basics

Q, K, V viennent du même ensemble

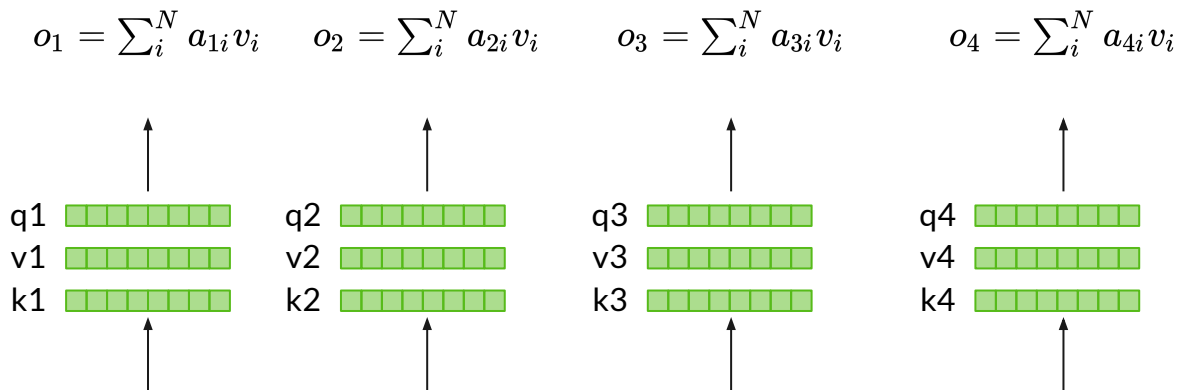


Self-attention basics

Q, K, V viennent du même ensemble

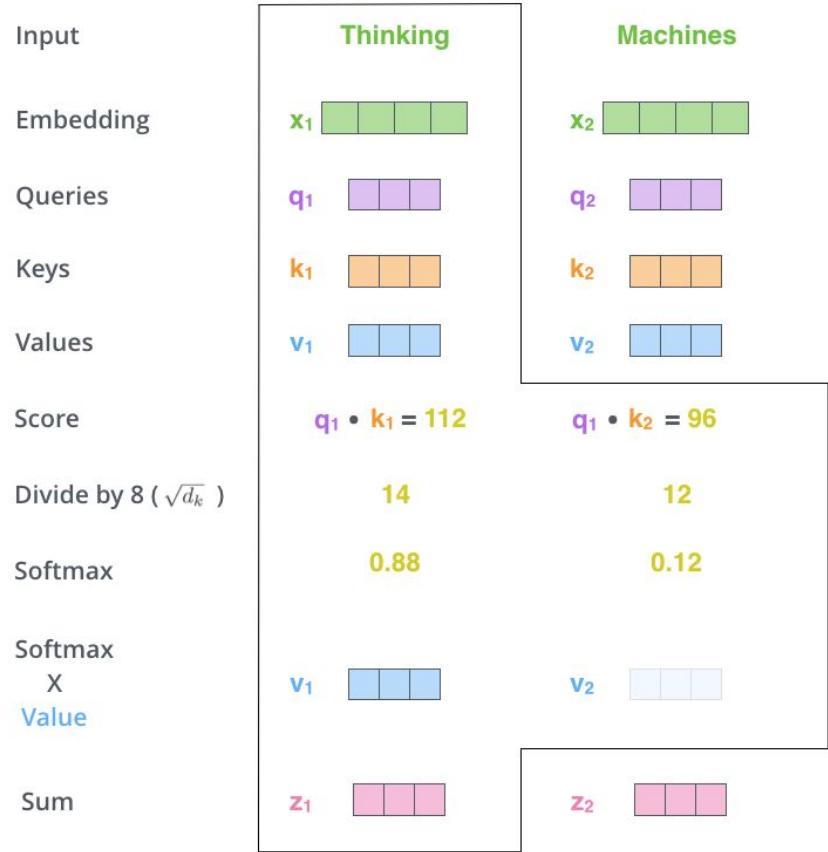
Scaled Dot-Product Attention

$$a_{ij} = \text{softmax}\left(\frac{q_i k_j^\top}{\sqrt{d_k}}\right)$$



Self-attention basics

(<https://jalammar.github.io/illustrated-transformer/>)



Self-attention basics

(<https://jalammr.github.io/illustrated-transformer/>)



Input

Embedding

Queries

Keys

Values

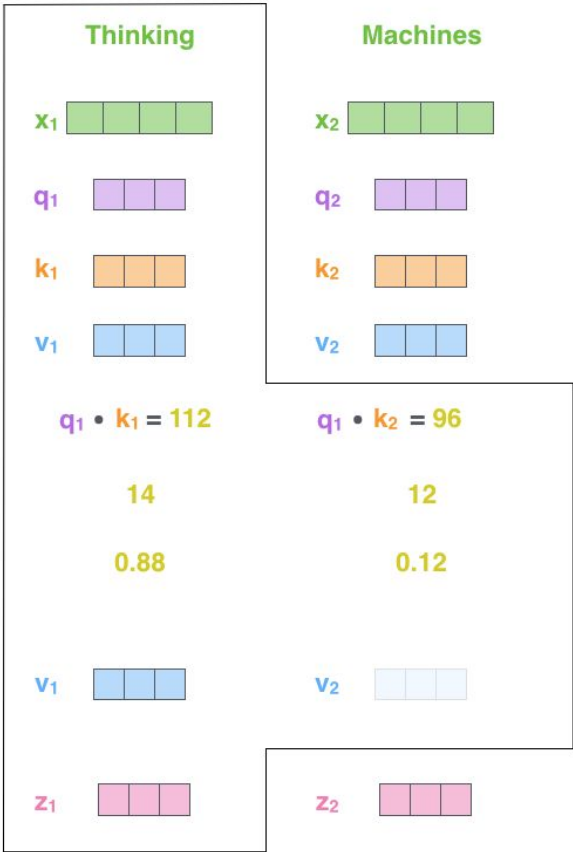
Score

Divide by 8 ($\sqrt{d_k}$)

Softmax

Softmax
X
Value

Sum



Transformer architecture

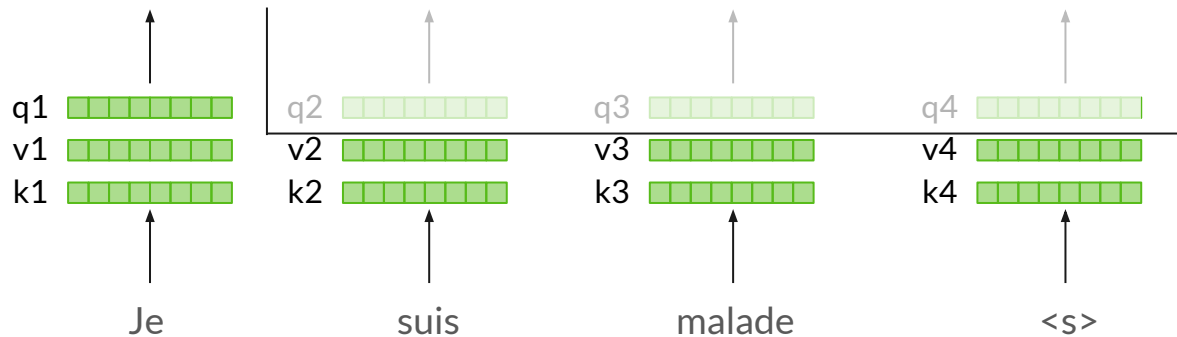
(Vaswani et al., 2019)

Attention is all you need ! *(Vaswani et al., 2019)*

On enlève les RNNs !

Éléments clés:

- Self-attention (multi hidden-state propagation)



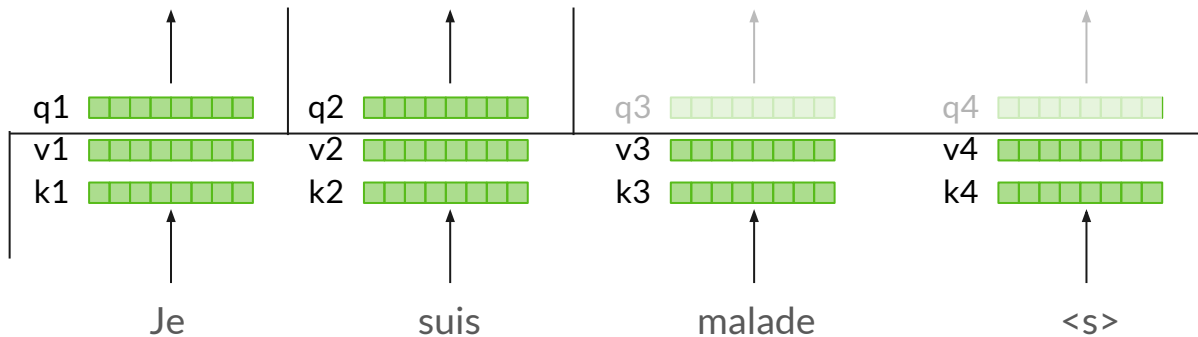
Attention is all you need ! *(Vaswani et al., 2019)*

On enlève les RNNs !

Éléments clés:

- Self-attention (multi hidden-state propagation)

Tous les O_i sont produits en même temps (un peu comme dans un RNN bi-directionnel)



Attention is all you need ! *(Vaswani et al., 2019)*

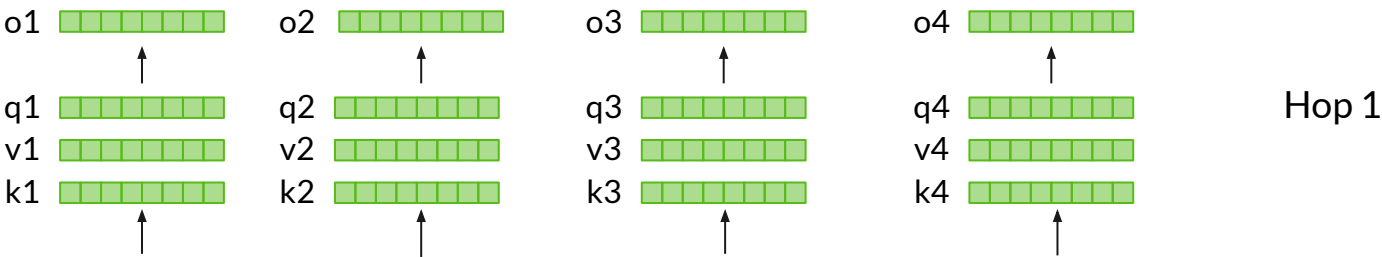
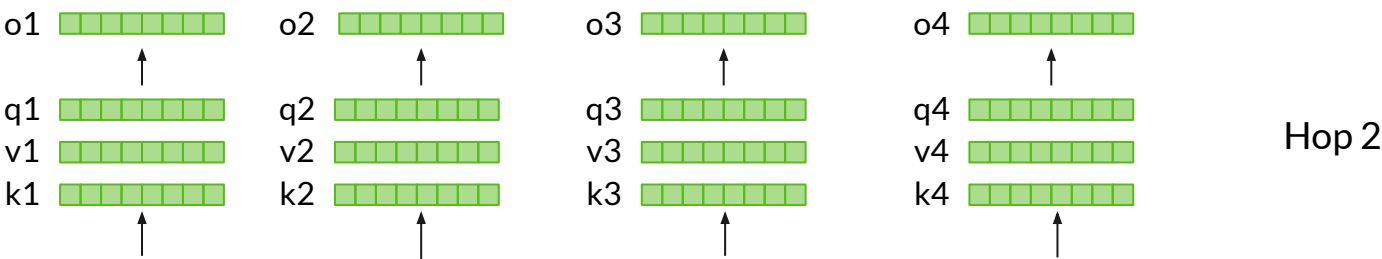
On enlève les RNNs !

Éléments clés:

- Self-attention (multi hidden-state propagation)
- Multi-hop (layers)

Multi-hop

Plusieurs itérations

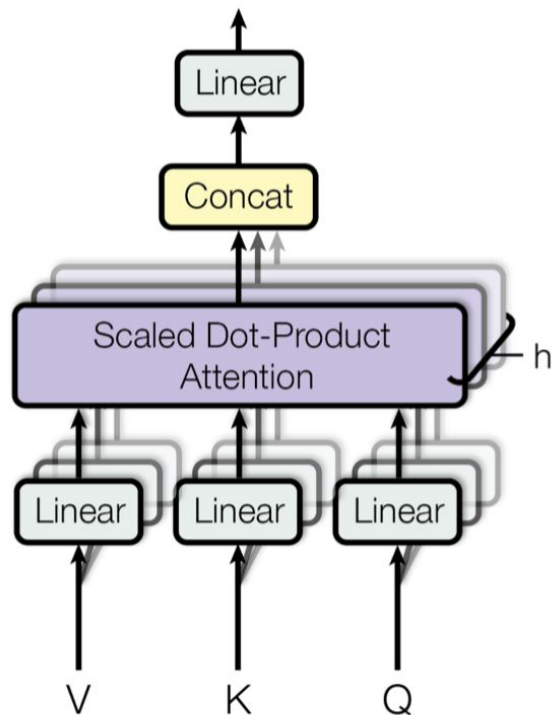


Attention is all you need ! *(Vaswani et al., 2019)*

On enlève les RNNs !

Éléments clés:

- Self-attention (multi hidden-state propagation)
- Multi-hop (layers)
- Multi-head



Attention is all you need ! *(Vaswani et al., 2019)*

- 1 **matrice de poids** par tête pour Q, K, V

- **Projection** avec une transformation linéaire

- **Dimensions** de Q, K, V définies par celle des poids

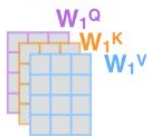
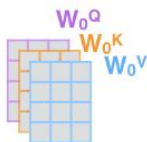
- On peut "**batcher**" les opérations

1) This is our input sentence*

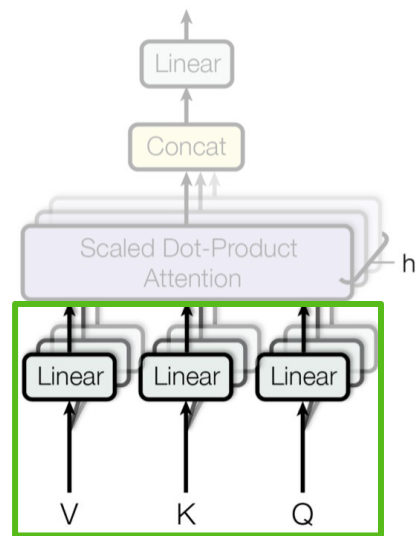
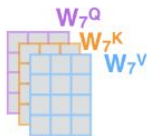
2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

Thinking Machines



...



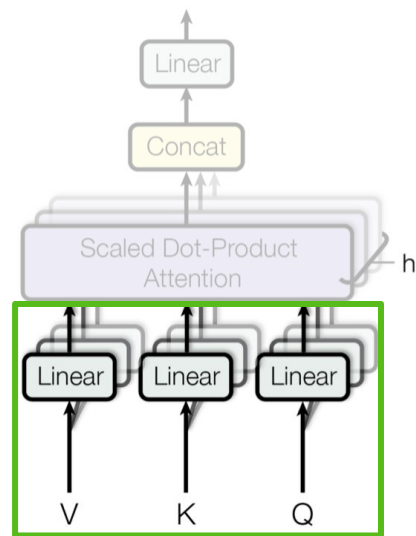
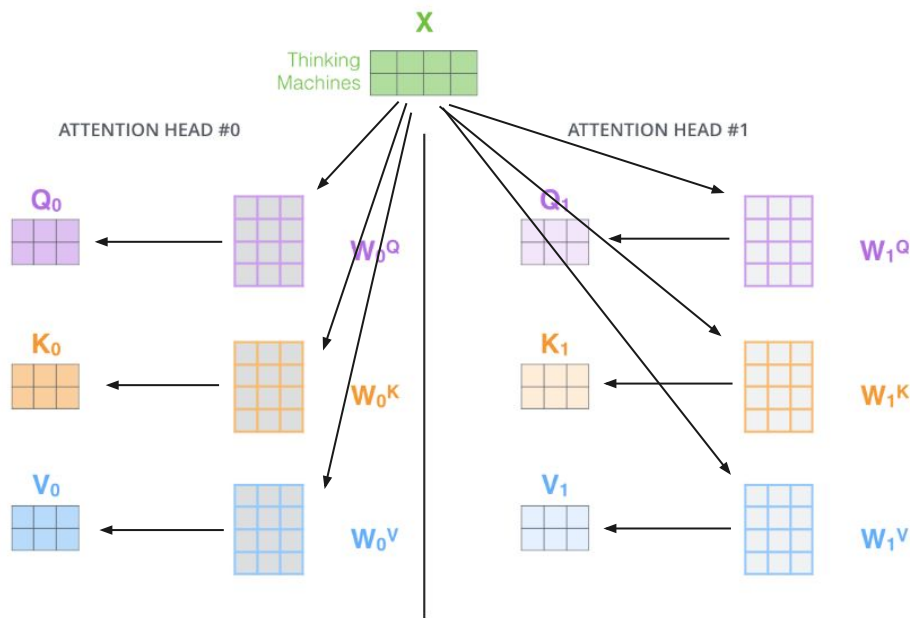
Attention is all you need ! (Vaswani et al., 2019)

- 1 **matrice de poids** par tête pour Q, K, V

- **Projection** avec une transformation linéaire

- **Dimensions** de Q, K, V définies par celle des poids

- On peut “**batcher**” les opérations



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the $W_Q/W_K/W_V$ matrices to produce Q/K/V matrices.

Attention is all you need ! (Vaswani et al., 2019)

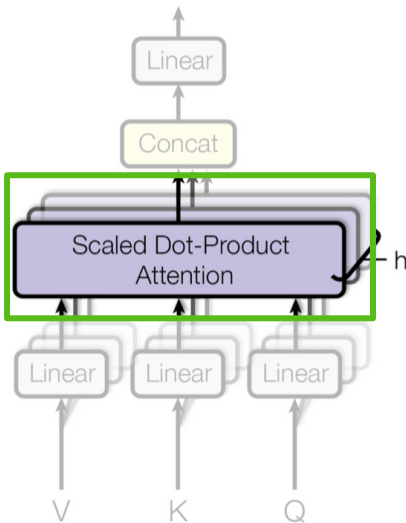
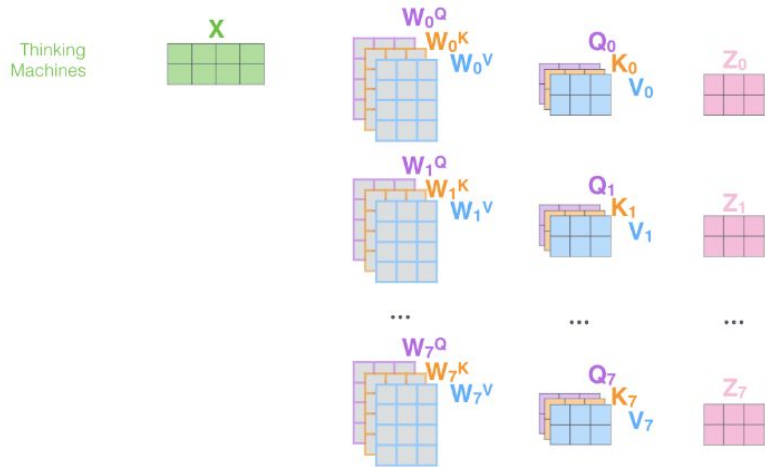
- 1 **matrice de poids** par tête pour Q, K, V

- **Projection** avec une transformation linéaire

- **Dimensions** de Q, K, V définies par celle des poids

- On peut “**batcher**” les opérations

- 1) This is our input sentence*
- 2) We embed each word*
- 3) Split into 8 heads. We multiply **X** or **R** with weight matrices
- 4) Calculate attention using the resulting **Q/K/V** matrices



$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V$$
$$= Z$$

Attention is all you need ! *(Vaswani et al., 2019)*

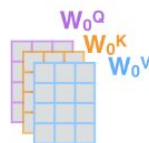
- On **concatène** les vecteurs produits par chaque tête (= un grand vecteur par entrée / mot)

1) This is our input sentence*
Thinking Machines

2) We embed each word*



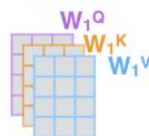
3) Split into 8 heads. We multiply X or R with weight matrices



4) Calculate attention using the resulting $Q/K/V$ matrices



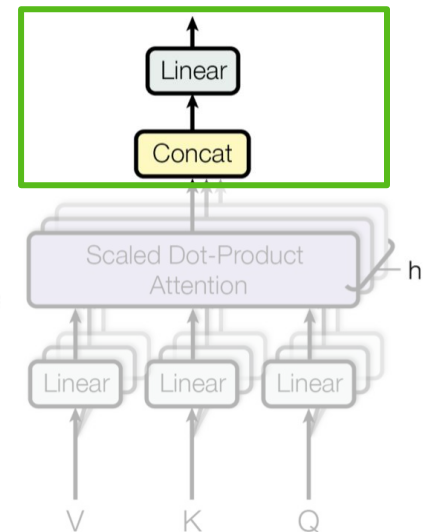
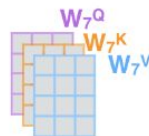
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



...

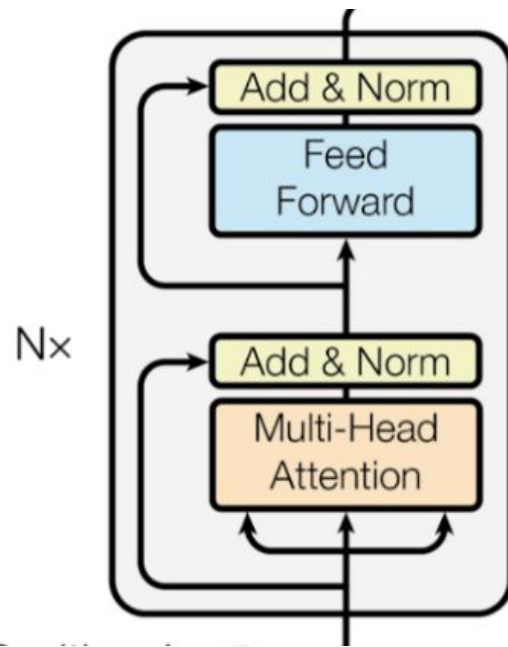
...

...



- On utilise une transformation linéaire pour revenir à la taille de l'embedding

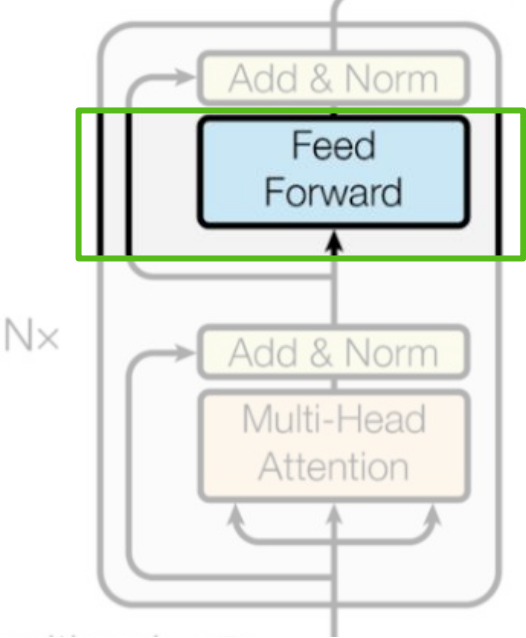
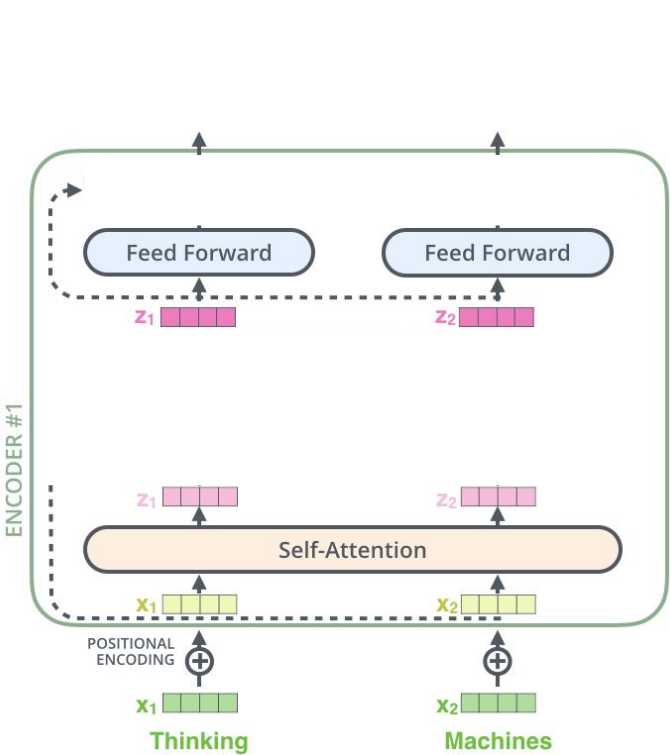
Transformer block



Transformer block

Feedforward

- Introduction de non-linéarités grâce à un feedforward



(<https://jalammar.github.io/illustrated-transformer/>)

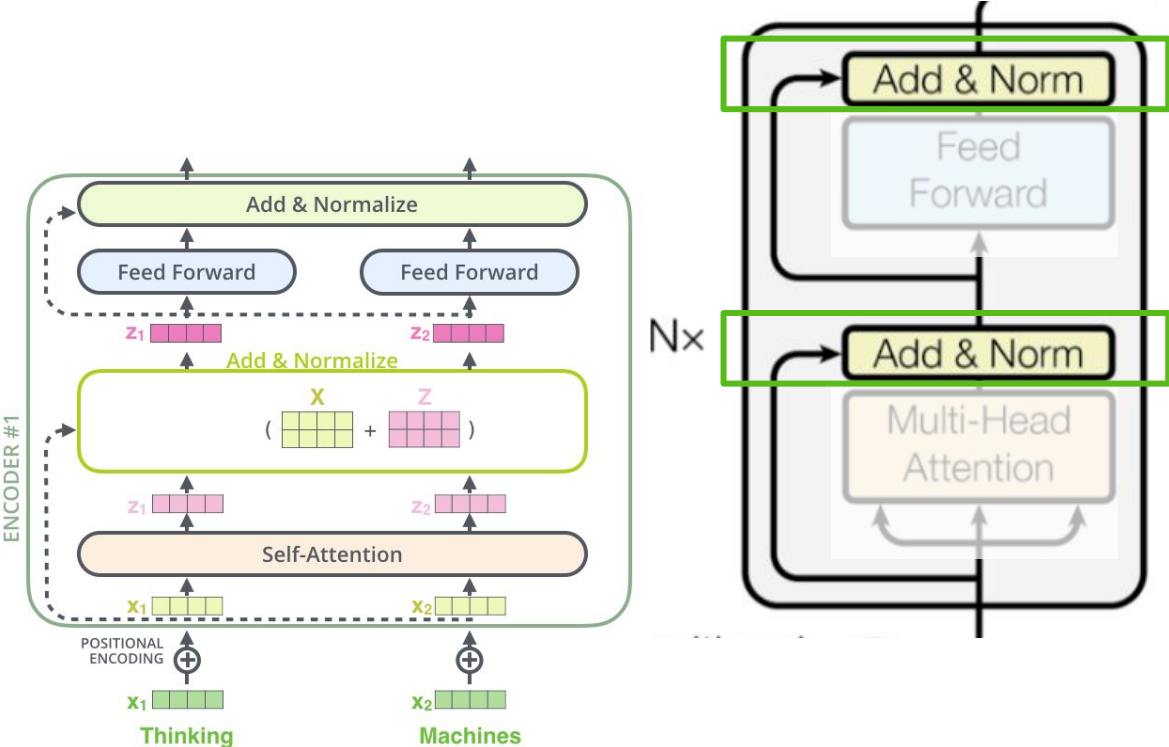
Transformer block

Residual connections

Intuition => garder de l'information sur l'embedding avant l'attention

En pratique: juste une somme

Note: ResNet (He et al., 2015)



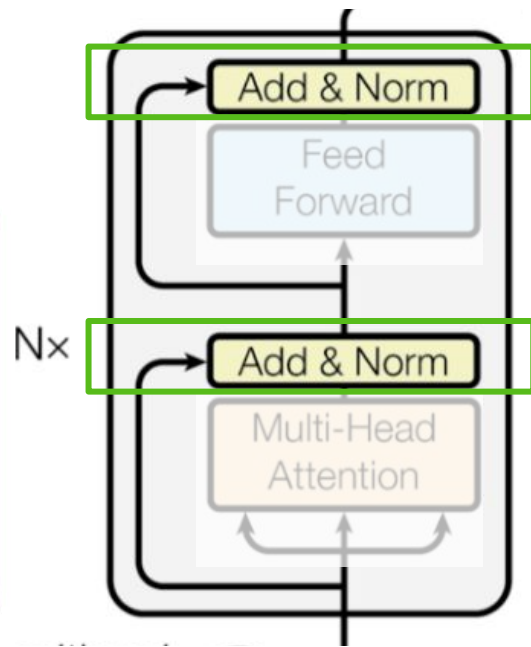
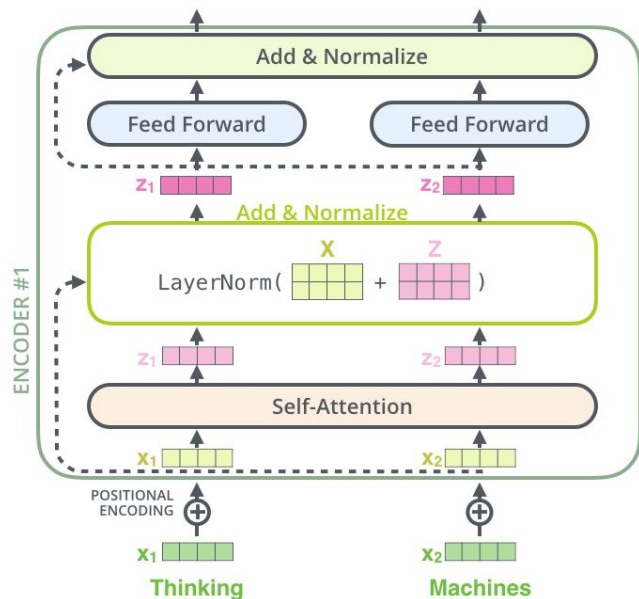
Transformer block

Layer normalization

Intuition => stabiliser
l'apprentissage et éviter
l'explosion de gradients

En pratique: on **normalize**
(moyenne = 0, variance = 1)
chaque vecteur de sortie

*Note: fonctionne avec les petits
batches à la différence d'une
BatchNorm*

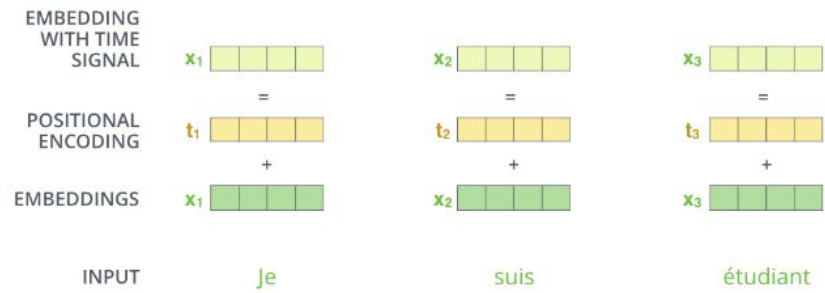


Positional Encoding

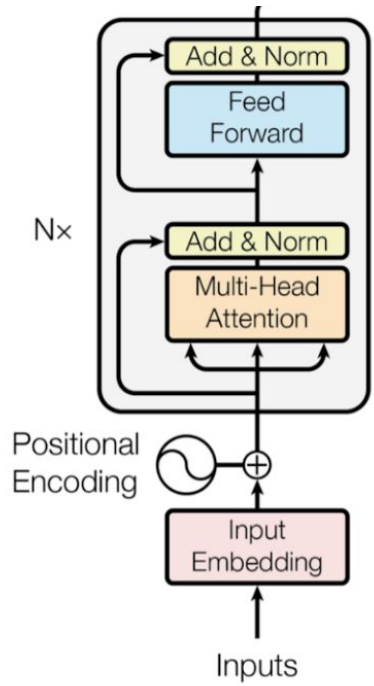
Rappel: La (Self-)Attention opère sur des ensembles != RNNs

=> L'opération n'est pas impactée par l'ordre des mots

Solution: Ajouter une information à l'embedding dépendant de la position



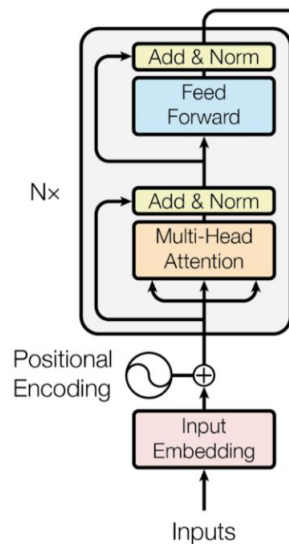
Note: Sinusoidal PE dans le papier d'origine



Encoder-Decoder

L'architecture complète !

Encoder: Produire une représentation de l'input (un vecteur / input => concaténés dans une seule matrice de dimension N)

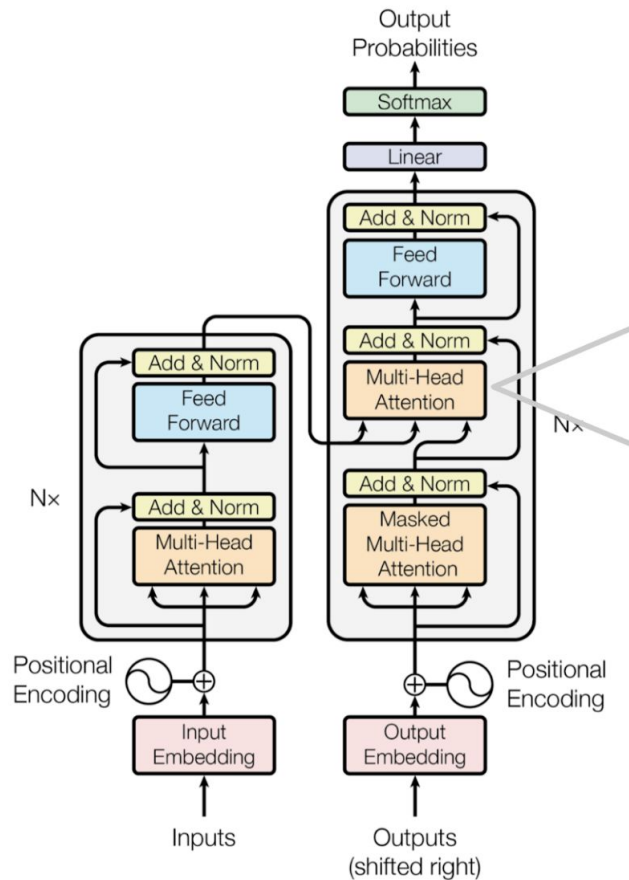


Encoder-Decoder

L'architecture complète !

Encoder: Produire une représentation de l'input (un vecteur / input => concaténés dans une seule matrice de dimension N)

Decoder: Générer à partir de cette représentation

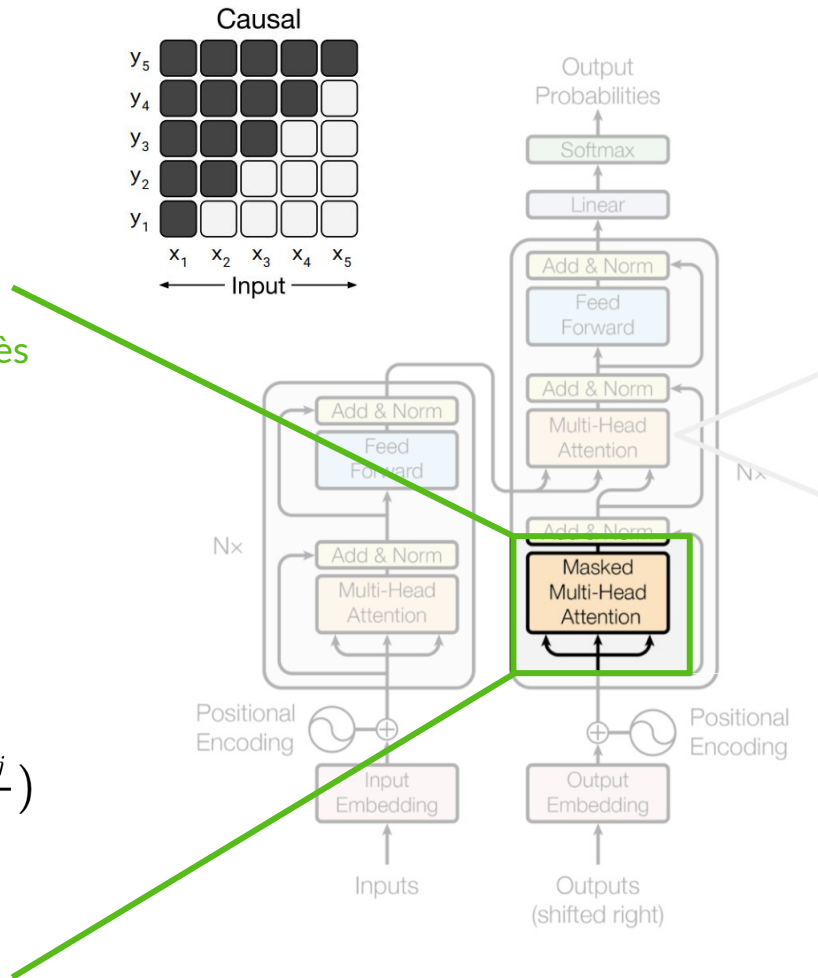


Encoder-Decoder

Causal Masking

- Quand on génère du texte, **on ne connaît pas le mot d'après** (on y a pourtant accès à l'entraînement)
- **Causal Attention**: L'attention ne peut utiliser que les **K, V** d'avant le Q
- Solution: Appliquer un **masque** sur la sortie de l'attention avant la softmax

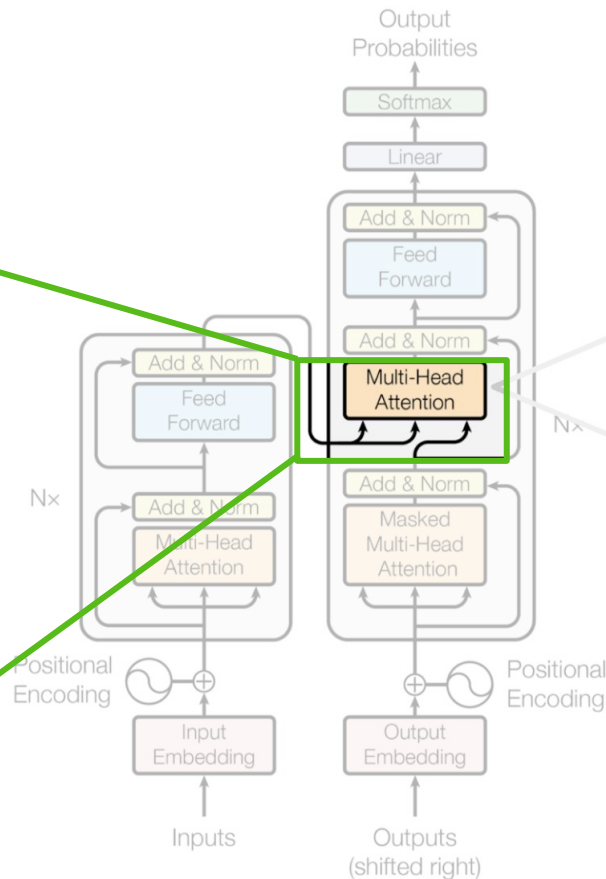
$$a_{ij} = \text{softmax}\left(\frac{q_i k_j^T}{\sqrt{d_k}}\right) \quad \longrightarrow \quad a_{ij} = \text{softmax}\left(\frac{q_i k_j^T + m_j}{\sqrt{d_k}}\right)$$
$$m_j = \begin{cases} 0 & \text{si } j \leq i \\ -\text{inf} & \text{si } j > i \end{cases}$$



Encoder-Decoder

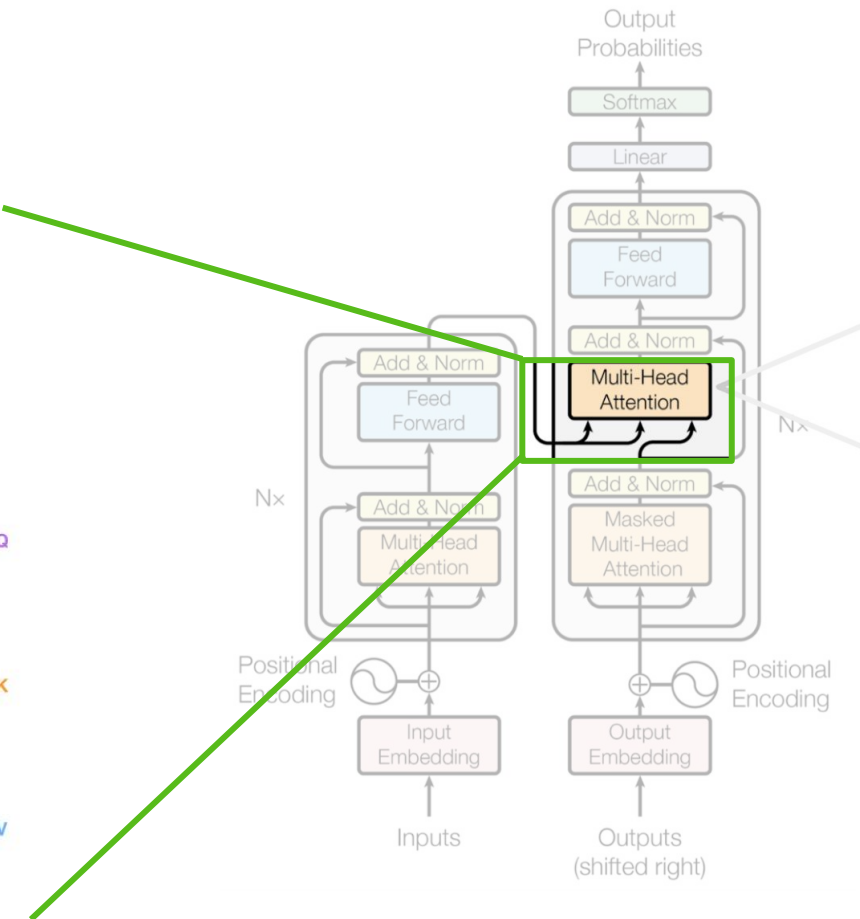
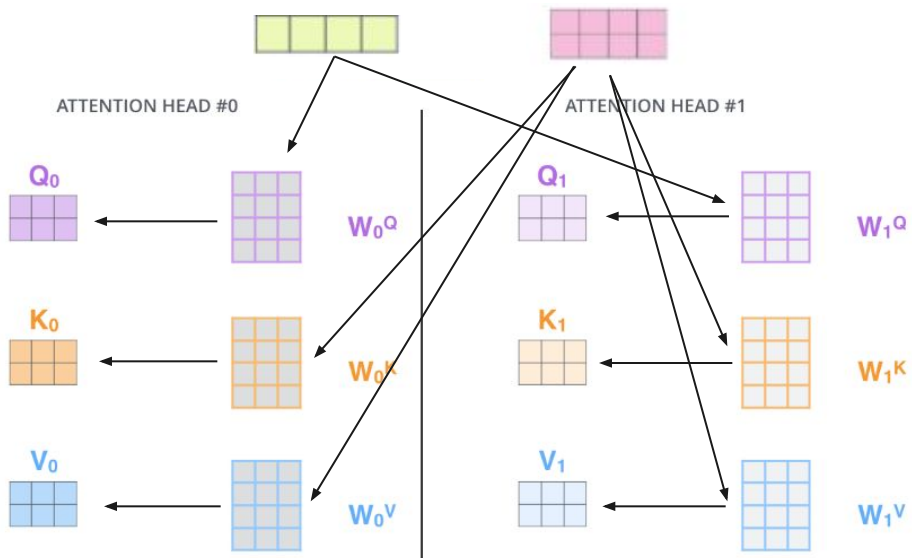
Attention over encoder's outputs

- Après un premier block de Self-Attention (+ residual & Layer Norm)
- **Attention sur l'encoding**: K, V viennent des vecteurs encodés par l'encoder, Q vient du Decoder
- **La sortie devient une combinaison des encodings**
- **Le residual** permet de garder aussi la représentation du Decoder



Encoder-Decoder

Attention over Encoder's outputs



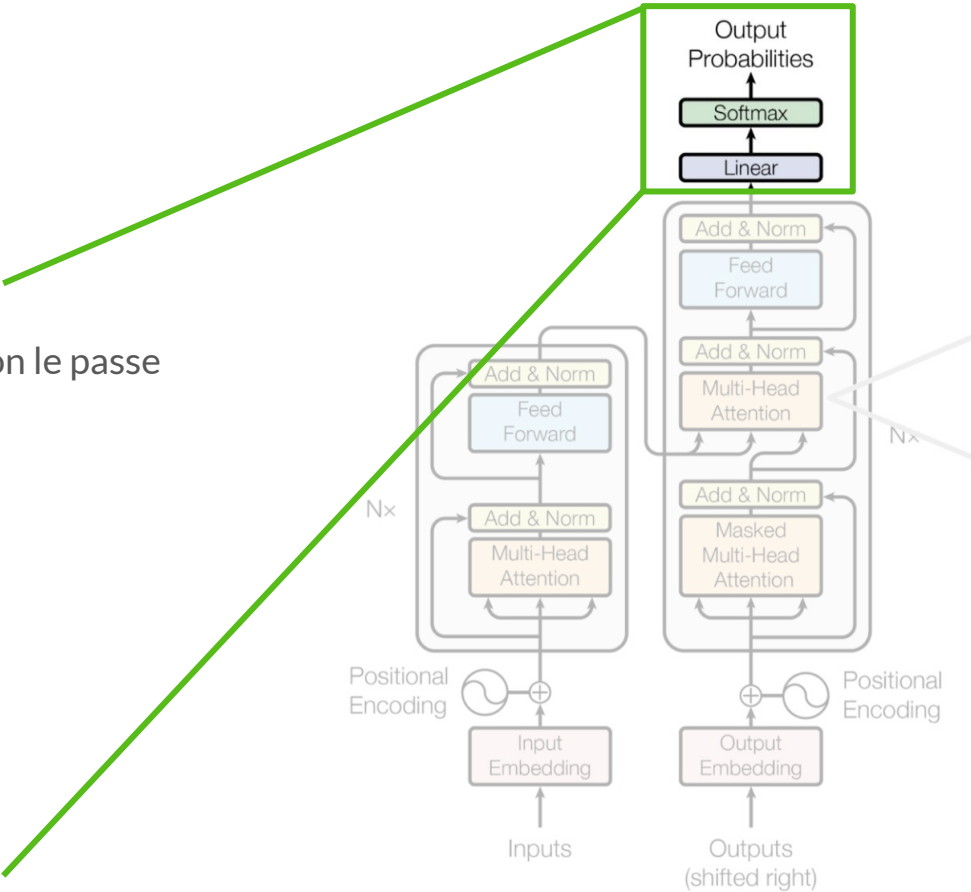
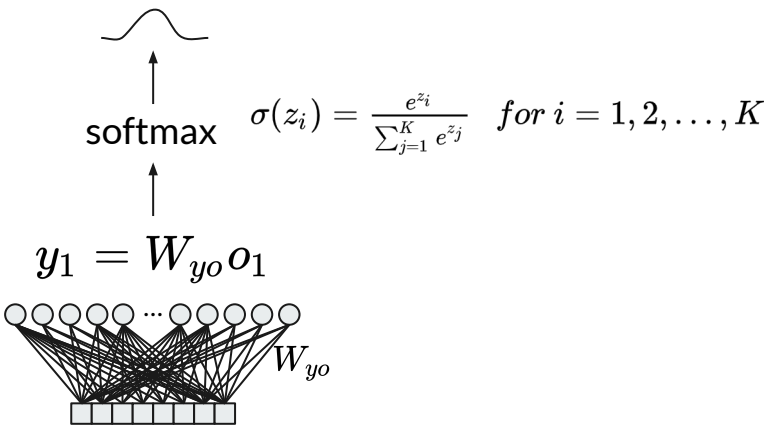
(<https://jalamar.github.io/illustrated-transformer/>)

Encoder-Decoder

Decoding

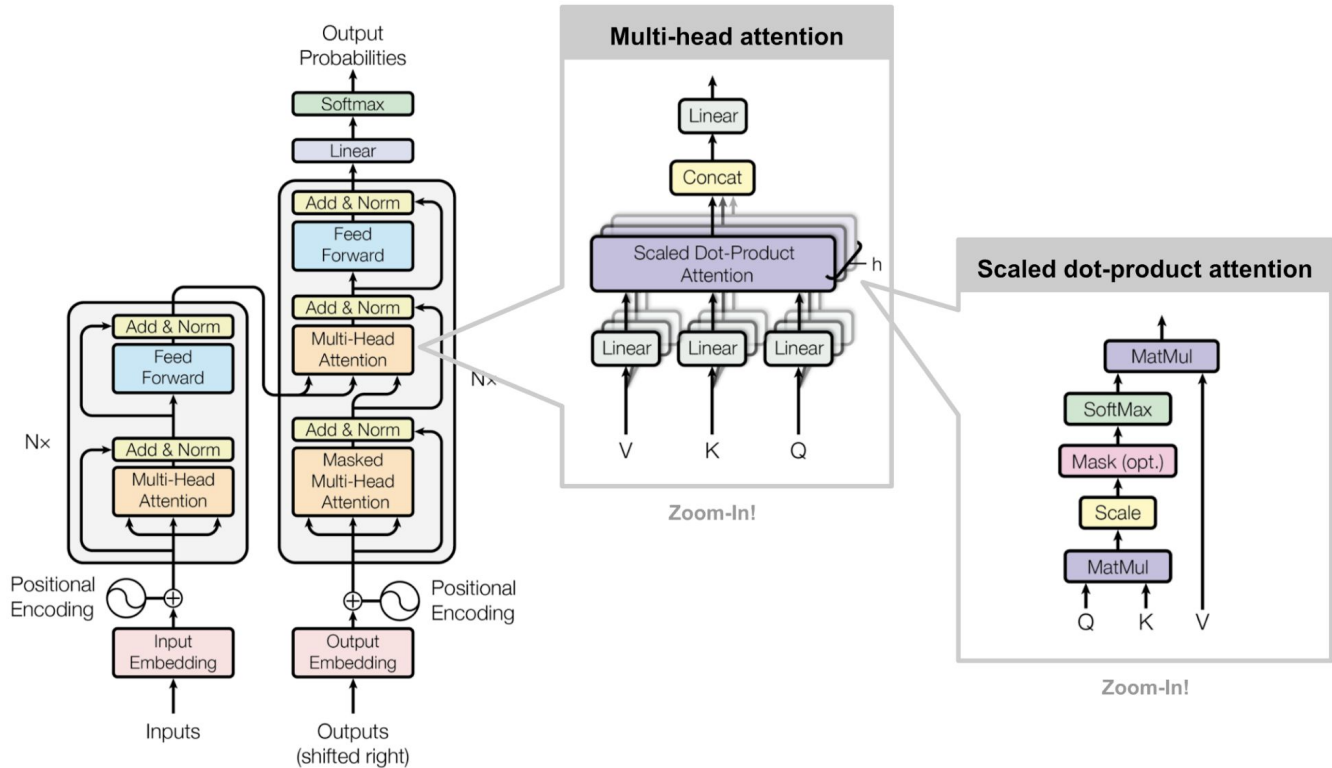
Comme avec les RNNs

=> on prend le vecteur produit par le Decoder et on le passe dans une couche linéaire + softmax



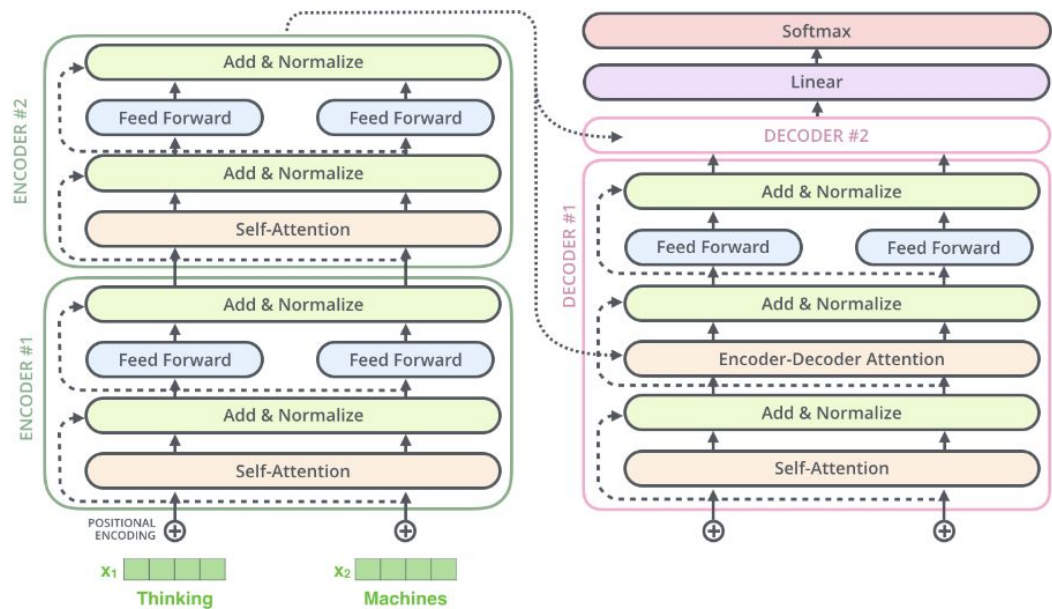
Encoder-Decoder

The full picture



Encoder-Decoder

The full picture

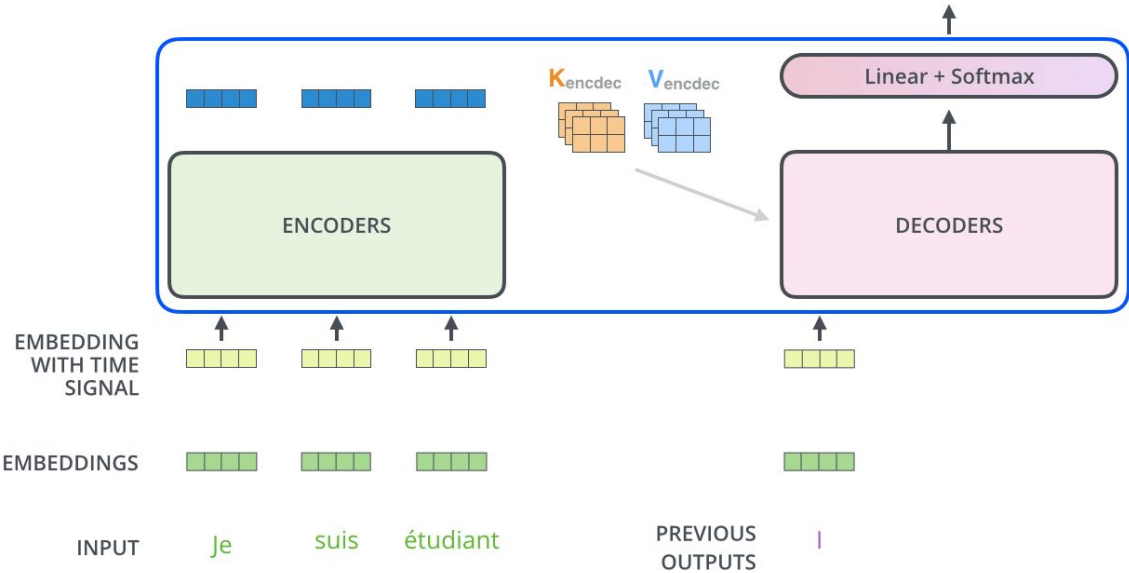


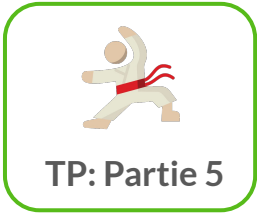
Encoder-Decoder

The full picture

Decoding time step: 1 2 3 4 5 6

OUTPUT |



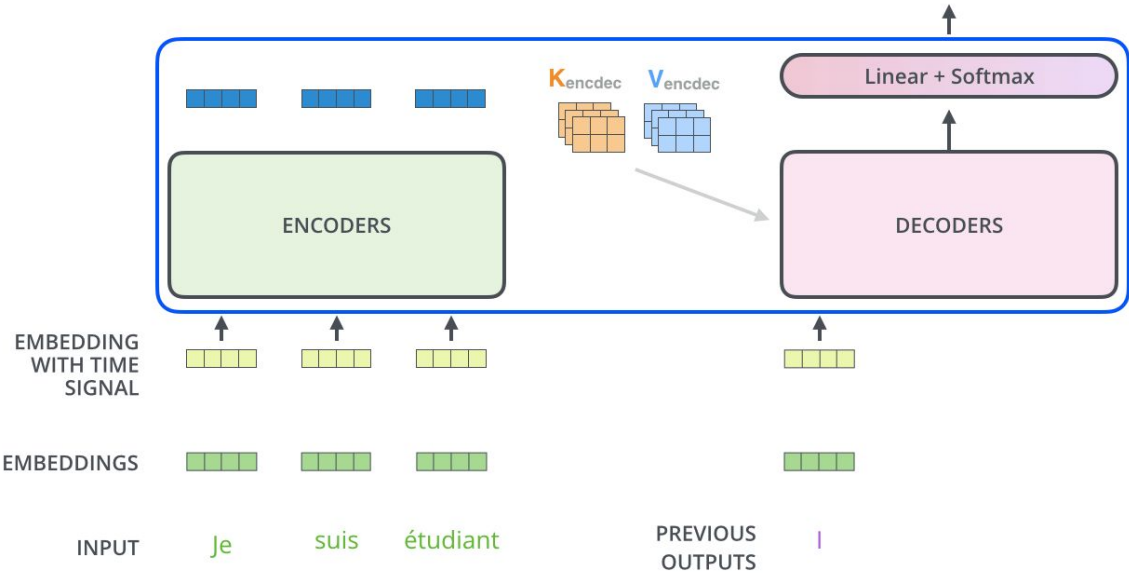


Encoder-Decoder

The full picture

Decoding time step: 1 2 3 4 5 6

OUTPUT |



Pour aller plus loin

Vision Transformer (ViT) (Dosovitskiy et al., 2020)

