Transformers

Jérémie Cabessa

Laboratoire DAVID, UVSQ

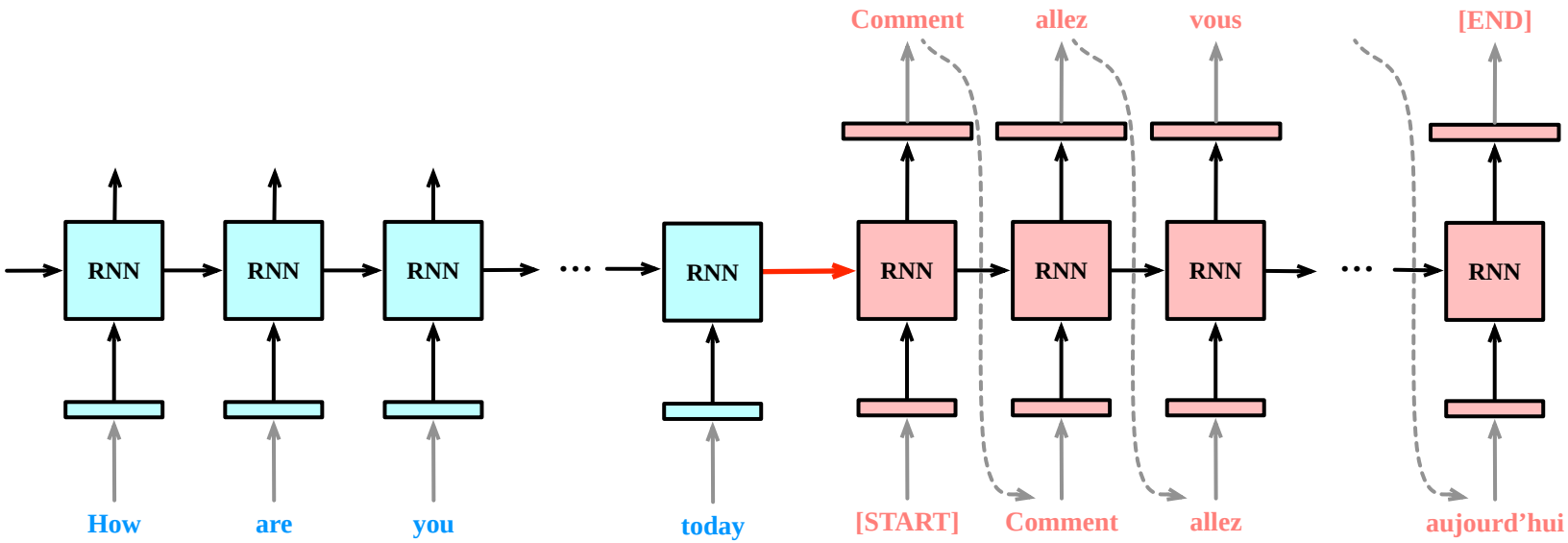
Architecture encodeur-décodeur

- Les **transformers** sont les modèles révolutionnaires qui ont donné lieu à toute la famille des **large language models (LLMs)**.
- Ils ont par la suite été généralisés dans d'autres domaines que le NLP: vision, tabular, etc.

Architecture encodeur-décodeur

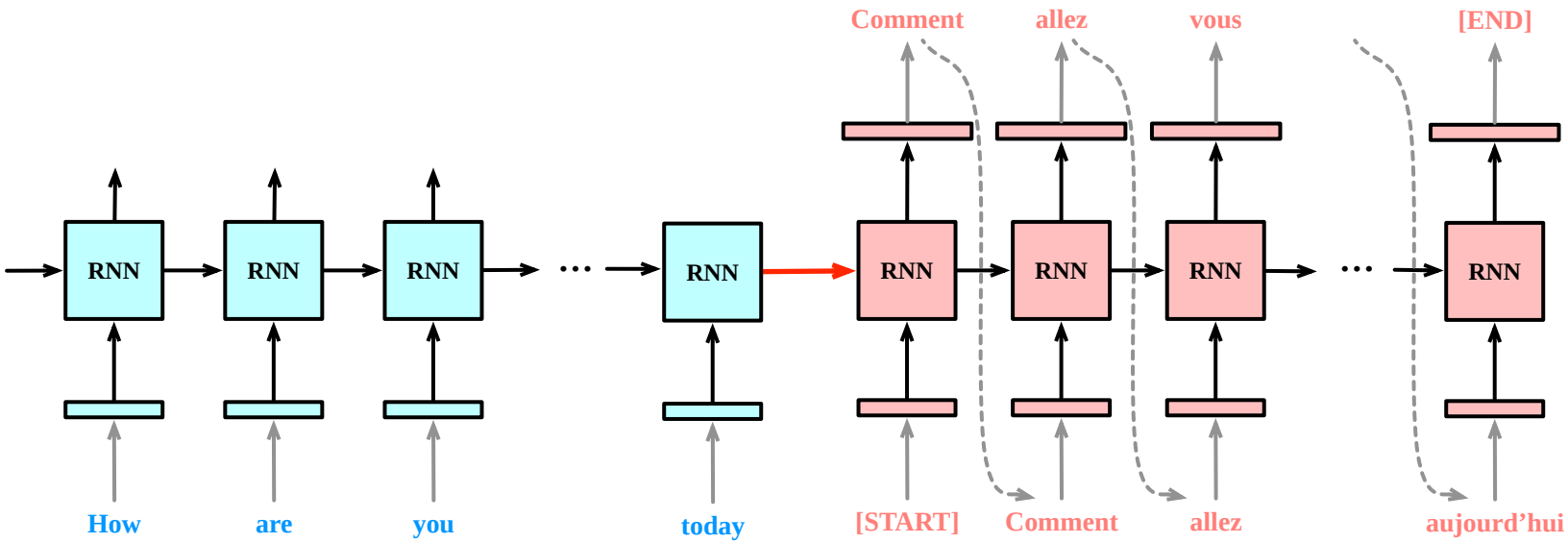
- Les **transformers** sont les modèles révolutionnaires qui ont donné lieu à toute la famille des **large language models (LLMs)**.
- Ils ont par la suite été généralisés dans d'autres domaines que le NLP: vision, tabular, etc.

Architecture encodeur-décodeur



- Le dernier état de l'encodeur (flèche horizontale rouge) est le *context vector*.
- C'est un *embedding* qui encode tous les inputs et permet le décodage.

Architecture encodeur-décodeur



- Le dernier état de l'encodeur (flèche horizontale rouge) est le *context vector*.
- C'est un *embedding* qui encode tous les inputs et permet le décodage.

Architecture encodeur-décodeur

- Les réseaux récurrents (RNNs) souffrent de la *disparition du gradient (vanishing gradient)*.
- Les RNNs ne permettent pas la *parallélisation*: la forward pass doit être calculée de manière séquentielle, car les états sucessifs du réseaux sont donnés par l'historique des inputs.
- Les RNNs peinent à capturer les dépendances de long termes entre les inputs (long-term dependencies).

$\Rightarrow$ **Solution:** architecture *feedforward* (no vanishing/exploding gradients, parallelizable) couplée à un mécanisme d'*attention* (short and long term dependencies).

Architecture encodeur-décodeur

- Les réseaux récurrents (RNNs) souffrent de la *disparition du gradient (vanishing gradient)*.
- Les RNNs ne permettent pas la *parallélisation*: la forward pass doit être calculée de manière séquentielle, car les états sucessifs du réseaux sont donnés par l'historique des inputs.
- Les RNNs peinent à capturer les dépendances de long termes entre les inputs (long-term dependencies).

$\Rightarrow$ **Solution:** architecture *feedforward* (no vanishing/exploding gradients, parallelizable) couplée à un mécanisme d'*attention* (short and long term dependencies).

Architecture encodeur-décodeur

- Les réseaux récurrents (RNNs) souffrent de la *disparition du gradient (vanishing gradient)*.
- Les RNNs ne permettent pas la *parallélisation*: la forward pass doit être calculée de manière séquentielle, car les états sucessifs du réseaux sont donnés par l'historique des inputs.
- Les RNNs peinent à capturer les dépendances de long termes entre les inputs (long-term dependencies).

$\Rightarrow$ **Solution:** architecture *feedforward* (no vanishing/exploding gradients, parallelizable) couplée à un mécanisme d'*attention* (short and long term dependencies).

Architecture encodeur-décodeur

- Les réseaux récurrents (RNNs) souffrent de la *disparition du gradient (vanishing gradient)*.
- Les RNNs ne permettent pas la *parallélisation*: la forward pass doit être calculée de manière séquentielle, car les états sucessifs du réseaux sont donnés par l'historique des inputs.
- Les RNNs peinent à capturer les dépendances de long termes entre les inputs (long-term dependencies).

$\Rightarrow$ **Solution:** architecture *feedforward* (no vanishing/exploding gradients, parallelizable) couplée à un mécanisme d'*attention* (short and long term dependencies).

Transformer

- Architecture **encodeur-décodeur**.
- Réseau de neurones **feedforward** (no vanishing/exploding gradients, parallelizable)
- Mécanismes de **self-attention** et d'**attention** dans l'encodeur et le décodeur (short and long term dependencies).

Transformer

- Architecture **encodeur-décodeur**.
- Réseau de neurones **feedforward** (no vanishing/exploding gradients, parallelizable)
- Mécanismes de **self-attention** et d'**attention** dans l'encodeur et le décodeur (short and long term dependencies).

Transformer

- Architecture **encodeur-décodeur**.
- Réseau de neurones **feedforward** (no vanishing/exploding gradients, parallelizable)
- Mécanismes de **self-attention** et d'**attention** dans l'encodeur et le décodeur (short and long term dependencies).

Transformer

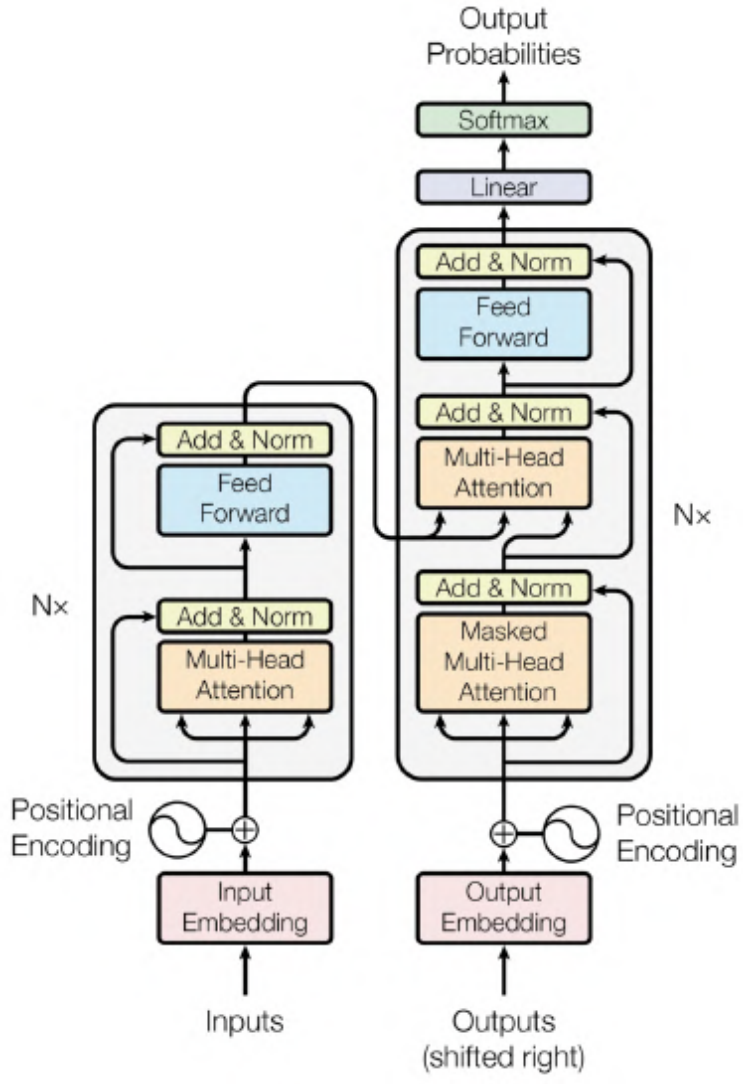


Figure taken from [Vaswani et al., 2017].

Transformer

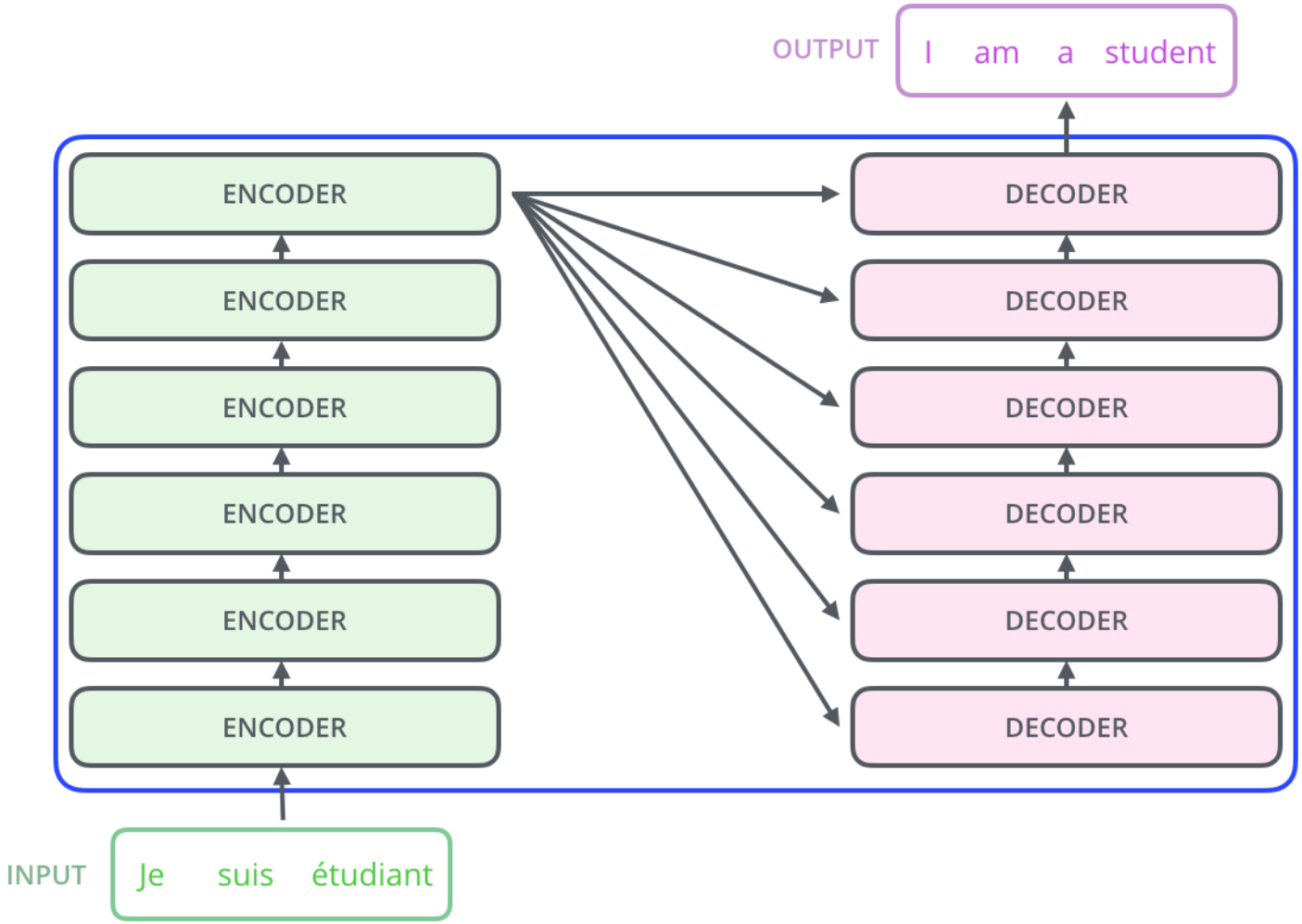


Figure taken from [Alammar, 2018].

Transformer

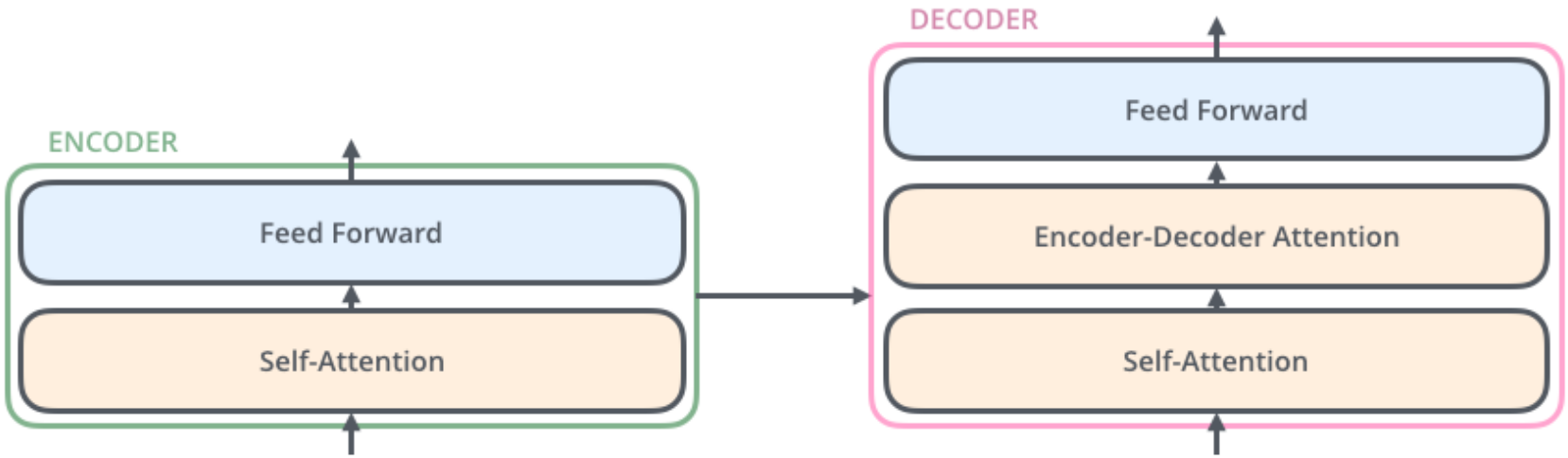


Figure taken from [Alammar, 2018].

Transformer

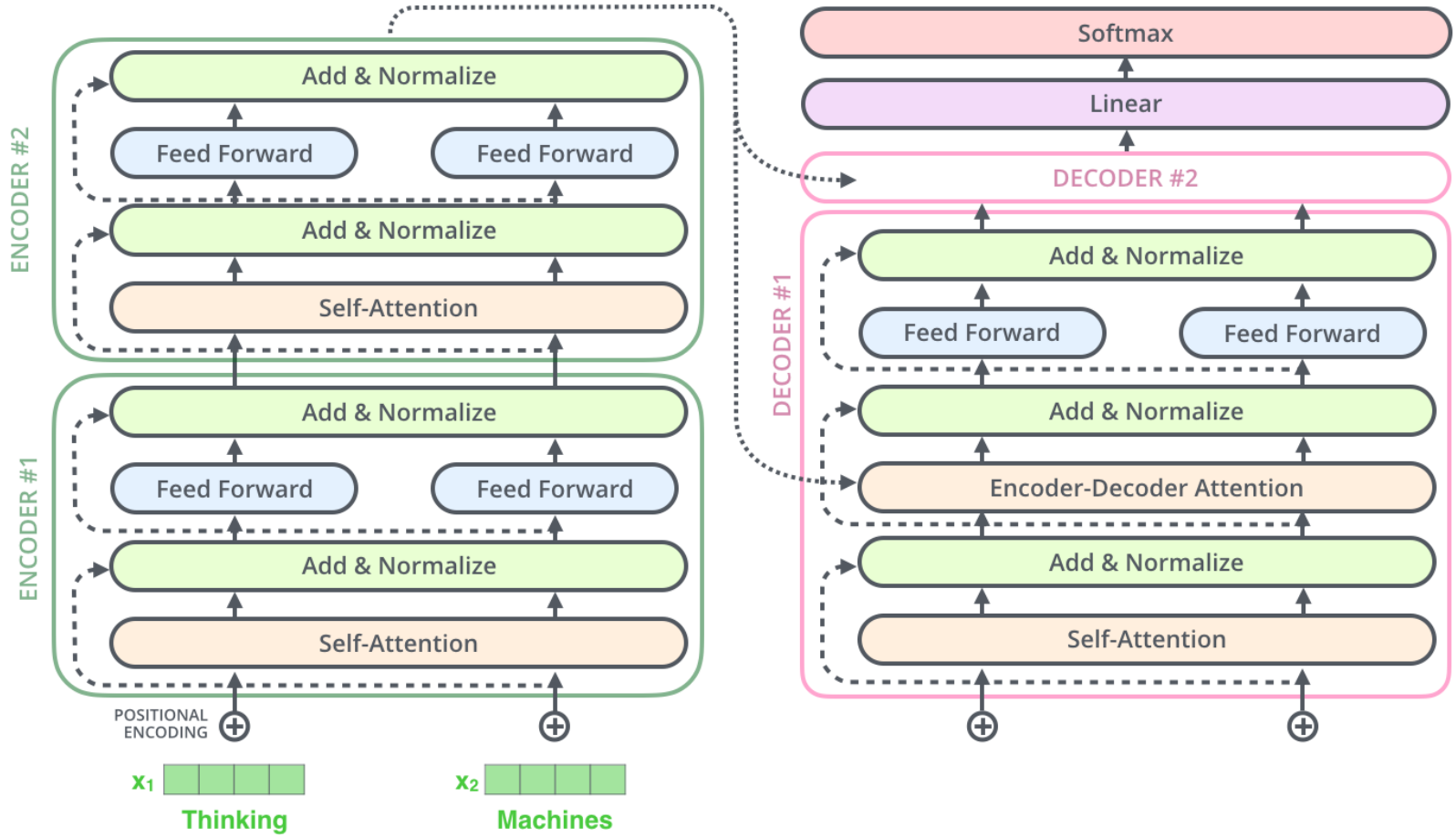


Figure taken from [Alammar, 2018].

Tokenization et embedding

- Le text passé en input est "tokenisé", i.e., convertit en une séquence de "input tokens".
- Chaque input token est associé à un entier appelé "token id", qui est à un indice dans un vocabulaire d'environ 40K mots.
- Chaque token id est ensuite "embeddé" en un vecteur de taille 512 grâce à un embedding statique sous forme de "lookup matrice" de dimension $30K \times 512$.
- À chaque "input embedding" est ajouté un vecteur de "positional encoding" qui capture l'information de la position de cet input dans le texte de départ.

Tokenization et embedding

- Le text passé en input est "tokenisé", i.e., convertit en une séquence de "input tokens".
- Chaque input token est associé à un entier appelé "token id", qui est à un indice dans un vocabulaire d'environ 40K mots.
- Chaque token id est ensuite "embeddé" en un vecteur de taille 512 grâce à un embedding statique sous forme de "lookup matrice" de dimension $30K \times 512$.
- À chaque "input embedding" est ajouté un vecteur de "positional encoding" qui capture l'information de la position de cet input dans le texte de départ.

Tokenization et embedding

- Le text passé en input est "tokenisé", i.e., convertit en une séquence de "input tokens".
- Chaque input token est associé à un entier appelé "token id", qui est à un indice dans un vocabulaire d'environ 40K mots.
- Chaque token id est ensuite "embeddé" en un vecteur de taille 512 grâce à un embedding statique sous forme de "lookup matrice" de dimension $30K \times 512$.
- À chaque "input embedding" est ajouté un vecteur de "positional encoding" qui capture l'information de la position de cet input dans le texte de départ.

Tokenization et embedding

- Le text passé en input est "tokenisé", i.e., convertit en une séquence de "input tokens".
- Chaque input token est associé à un entier appelé "token id", qui est à un indice dans un vocabulaire d'environ 40K mots.
- Chaque token id est ensuite "embeddé" en un vecteur de taille 512 grâce à un embedding statique sous forme de "lookup matrice" de dimension $30K \times 512$.
- À chaque "input embedding" est ajouté un vecteur de "positional encoding" qui capture l'information de la position de cet input dans le texte de départ.

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence

Let's learn deep learning!

# Tokenized Sentence

["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens

["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding

["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs

[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence

Let's learn deep learning!

# Tokenized Sentence

["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens

["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding

["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs

[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

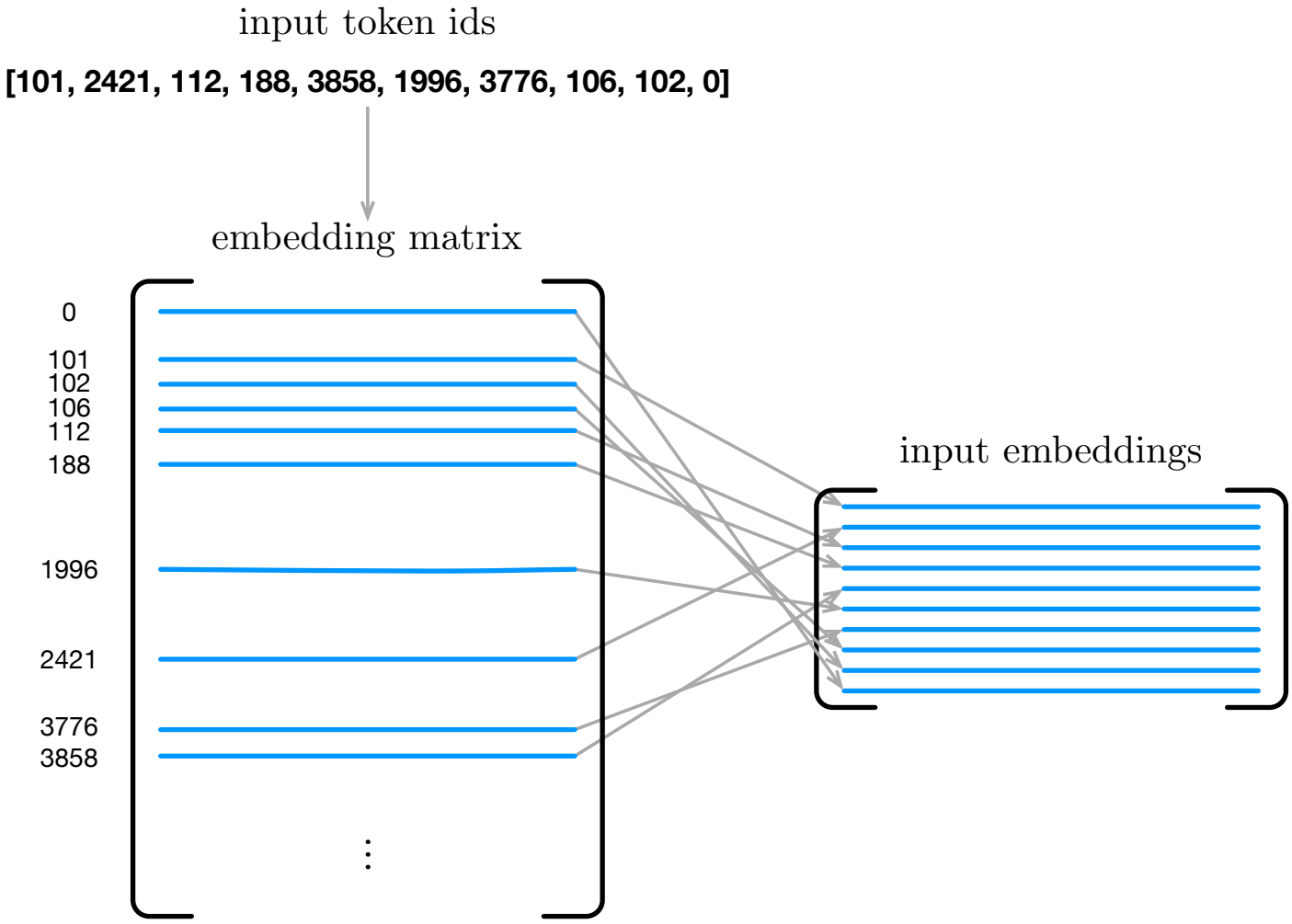
Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

```
# Original Sentence
Let's learn deep learning!

# Tokenized Sentence
["Let", "'", "s", "learn", "deep", "learning", "!"]

# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]

# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]

# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

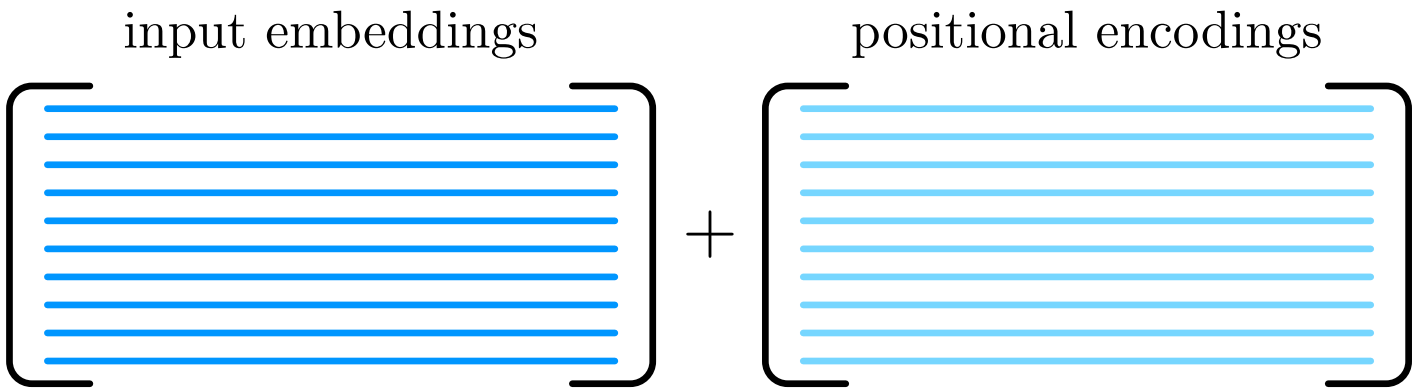
Tokenization et embedding



Tokenization et embedding

$$PE(pos, 2i) = \sin\left(pos/10000^{\frac{2i}{d}}\right)$$
$$PE(pos, 2i+1) = \cos\left(pos/10000^{\frac{2i}{d}}\right)$$

où pos est la position du input token $(0, 1, 2, \dots)$, i est la dimension de l'encoding $(0 \leq i \leq 511)$, et $d = 512$.



Attention

- On distingue 2 types d'attention:
  - l'attention classique
  - la self-attention

Attention

- On distingue 2 types d'attention:
  - **l'attention classique**
  - la self-attention

Attention

- On distingue 2 types d'attention:
  - **l'attention classique**
  - **la self-attention**

Attention

- Le **mécanisme d'attention** crée des embeddings de mots (tokens) qui prennent en compte d'autres mots de la phrase.
- L'attention joue le rôle de *mémoire*.
- Exemple: il serait bon que l'embedding du mot "bateau" dans la phrase "le bateau bleu est amarré dans le port d'Amsterdam" mette de l'attention sur les mots "bleu" et "amarré", puisque ces derniers apportent des précisions sur le bateau.
- En pratique, l'embedding d'un mot correspondra (en gros) à une combinaison pondérée des embeddings des autres mots.

Attention

- Le **mécanisme d'attention** crée des embeddings de mots (tokens) qui prennent en compte d'autres mots de la phrase.
- L'attention joue le rôle de *mémoire*.
- Exemple: il serait bon que l'embedding du mot "bateau" dans la phrase "le bateau bleu est amarré dans le port d'Amsterdam" mette de l'attention sur les mots "bleu" et "amarré", puisque ces derniers apportent des précisions sur le bateau.
- En pratique, l'embedding d'un mot correspondra (en gros) à une combinaison pondérée des embeddings des autres mots.

Attention

- Le **mécanisme d'attention** crée des embeddings de mots (tokens) qui prennent en compte d'autres mots de la phrase.
- L'attention joue le rôle de *mémoire*.
- Exemple: il serait bon que l'embedding du mot "bateau" dans la phrase "le bateau bleu est amarré dans le port d'Amsterdam" mette de l'attention sur les mots "bleu" et "amarré", puisque ces derniers apportent des précisions sur le bateau.
- En pratique, l'embedding d'un mot correspondra (en gros) à une combinaison pondérée des embeddings des autres mots.

Attention

- Le **mécanisme d'attention** crée des embeddings de mots (tokens) qui prennent en compte d'autres mots de la phrase.
- L'attention joue le rôle de *mémoire*.
- Exemple: il serait bon que l'embedding du mot "bateau" dans la phrase "le bateau bleu est amarré dans le port d'Amsterdam" mette de l'attention sur les mots "bleu" et "amarré", puisque ces derniers apportent des précisions sur le bateau.
- En pratique, l'embedding d'un mot correspondra (en gros) à une combinaison pondérée des embeddings des autres mots.

Attention

- Analogie avec un *système de recherche (retrieval system)* basé sur des *requêtes*, des *clés* et des *valeurs (queries, keys, and values)*.
- On a une base de données d'éléments représentés par des couples clé-valeur. Pour une requête donnée, on cherche les clés qui possèdent le plus de similarités avec cette dernière et on extrait les valeurs correspondantes.
- *A "query" matches different "keys" and retrieves the corresponding "values".*

Attention

- Analogie avec un *système de recherche (retrieval system)* basé sur des *requêtes*, des *clés* et des *valeurs (queries, keys, and values)*.
- On a une base de données d'éléments représentés par des couples clé-valeur. Pour une requête donnée, on cherche les clés qui possèdent le plus de similarités avec cette dernière et on extrait les valeurs correspondantes.
- *A "query" matches different "keys" and retrieves the corresponding "values".*

Attention

- Analogie avec un *système de recherche (retrieval system)* basé sur des *requêtes*, des *clés* et des *valeurs (queries, keys, and values)*.
- On a une base de données d'éléments représentés par des couples clé-valeur. Pour une requête donnée, on cherche les clés qui possèdent le plus de similarités avec cette dernière et on extrait les valeurs correspondantes.
- *A "query" matches different "keys" and retrieves the corresponding "values".*

Attention

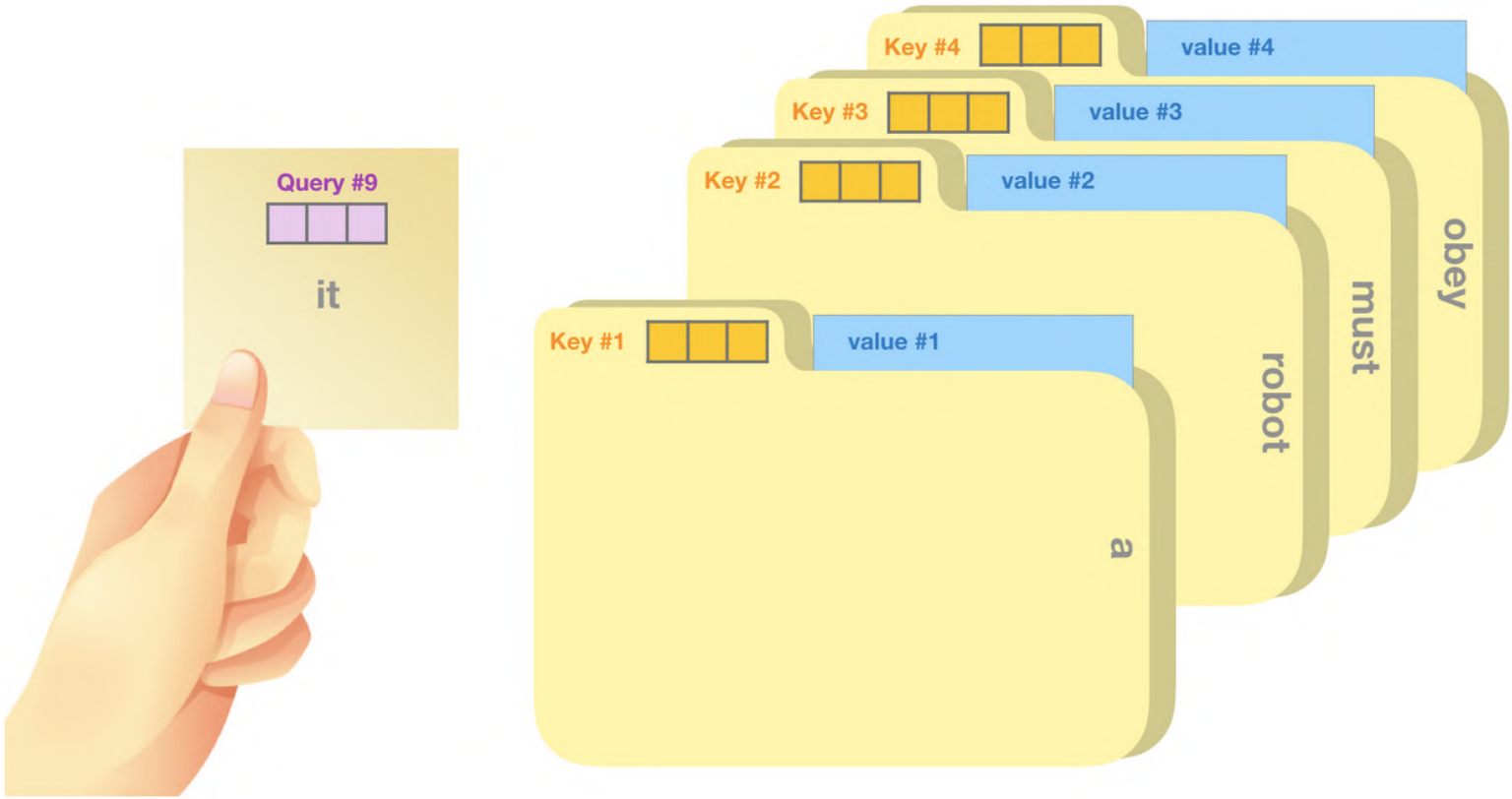


Figure taken from [Alammar, 2018].

Attention

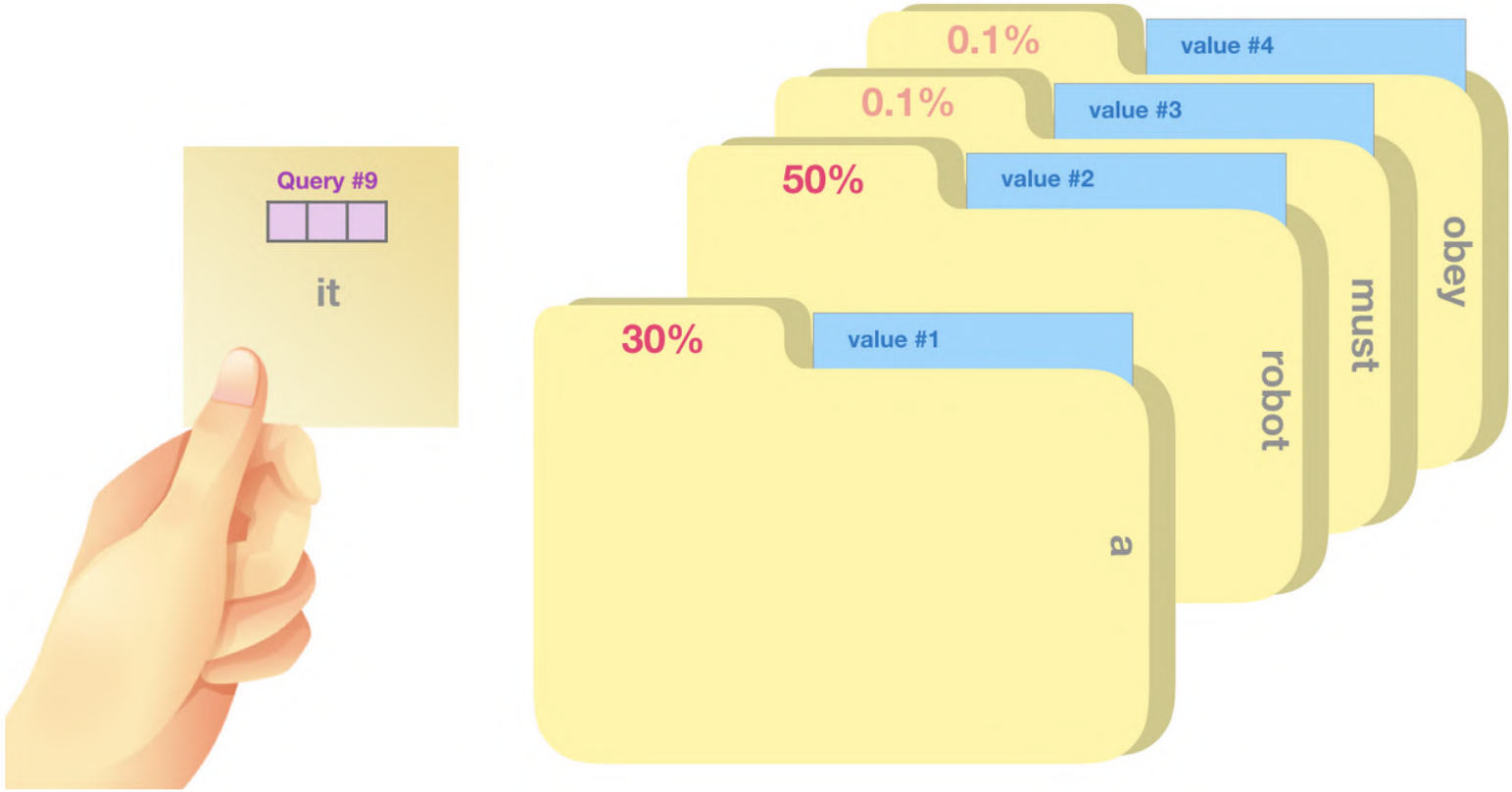


Figure taken from [Alammar, 2018].

Scaled Dot-Product Attention

- Pour la self-attention qui se déroule dans l'encodeur, les requêtes, les clés et les valeurs proviennent de la séquence des inputs.
- Pour la self-attention qui se déroule dans le décodeur, les requêtes, les clés et les valeurs proviennent de la séquence des mots décodés jusqu'à maintenant.
- Pour l'attention qui se déroule dans le décodeur, les requêtes proviennent des outputs de l'encodeur, et les clés et les valeurs proviennent de la séquence des mots décodés jusqu'à maintenant.

Scaled Dot-Product Attention

- Pour la self-attention qui se déroule dans l'encodeur, les requêtes, les clés et les valeurs proviennent de la séquence des inputs.
- Pour la self-attention qui se déroule dans le décodeur, les requêtes, les clés et les valeurs proviennent de la séquence des mots décodés jusqu'à maintenant.
- Pour l'attention qui se déroule dans le décodeur, les requêtes proviennent des outputs de l'encodeur, et les clés et les valeurs proviennent de la séquence des mots décodés jusqu'à maintenant.

Scaled Dot-Product Attention

- Pour la self-attention qui se déroule dans l'encodeur, les requêtes, les clés et les valeurs proviennent de la séquence des inputs.
- Pour la self-attention qui se déroule dans le décodeur, les requêtes, les clés et les valeurs proviennent de la séquence des mots décodés jusqu'à maintenant.
- Pour l'attention qui se déroule dans le décodeur, les requêtes proviennent des outputs de l'encodeur, et les clés et les valeurs proviennent de la séquence des mots décodés jusqu'à maintenant.

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

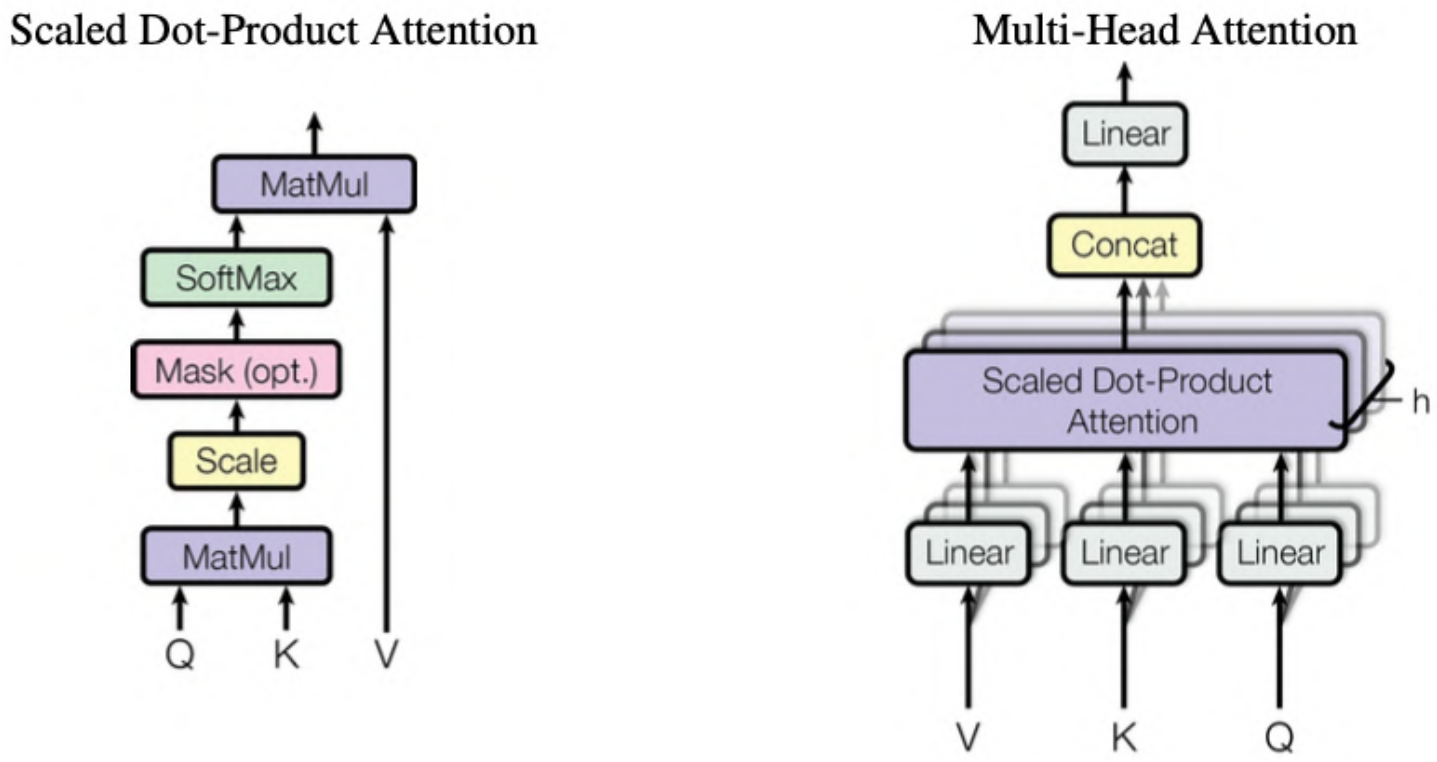
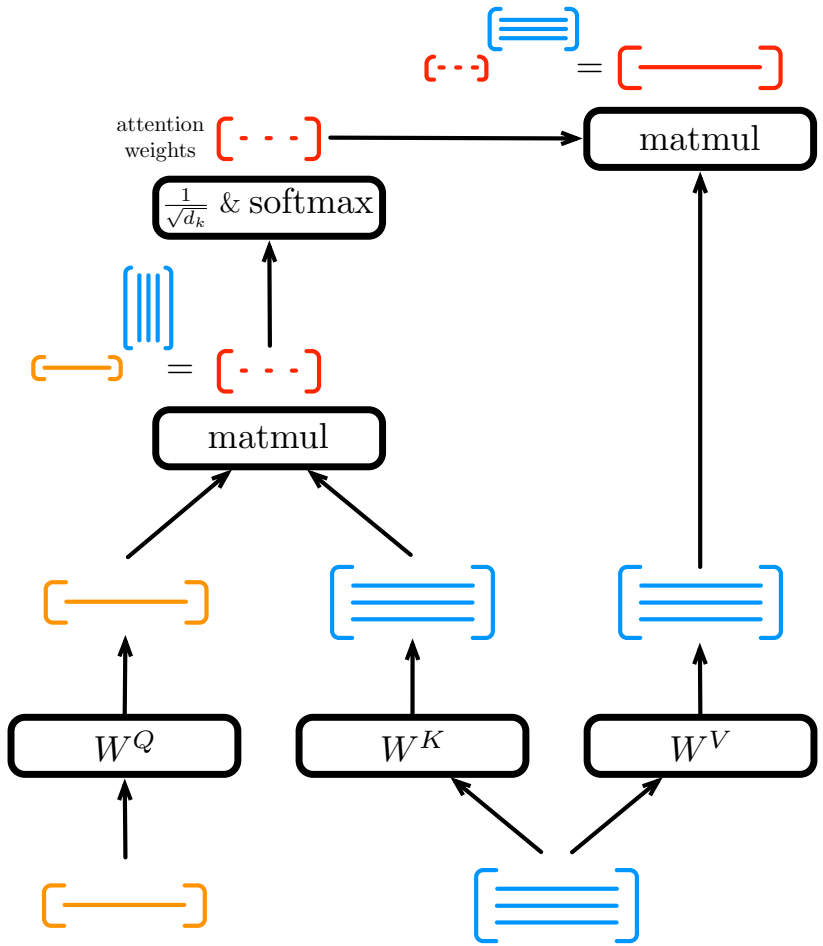


Figure taken from [Vaswani et al., 2017].

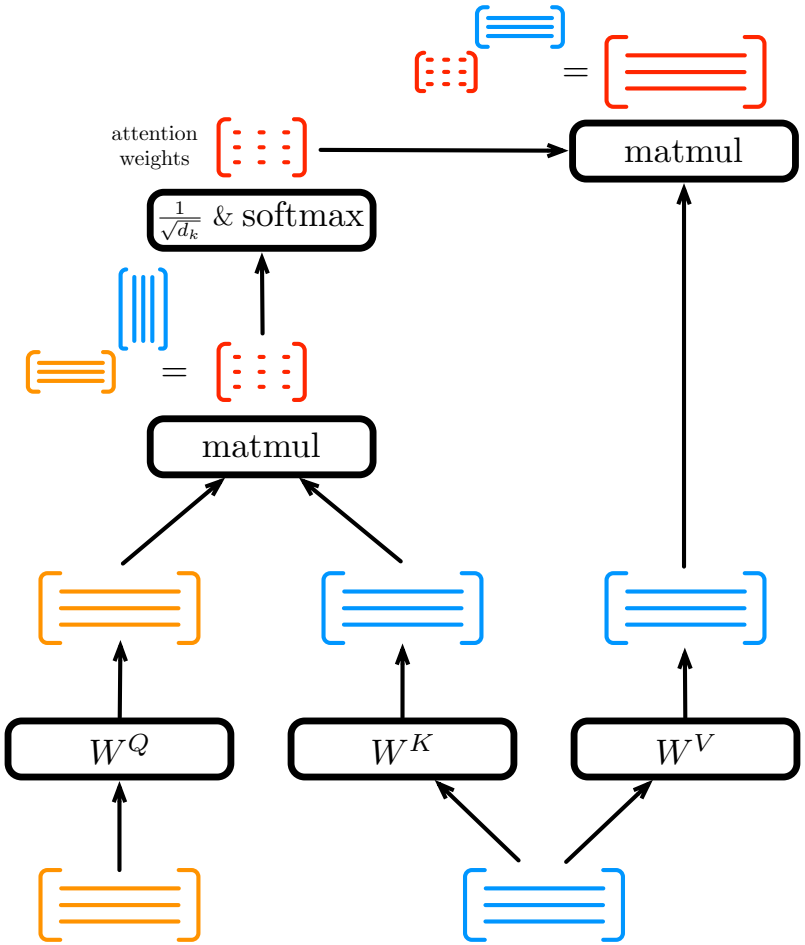
Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



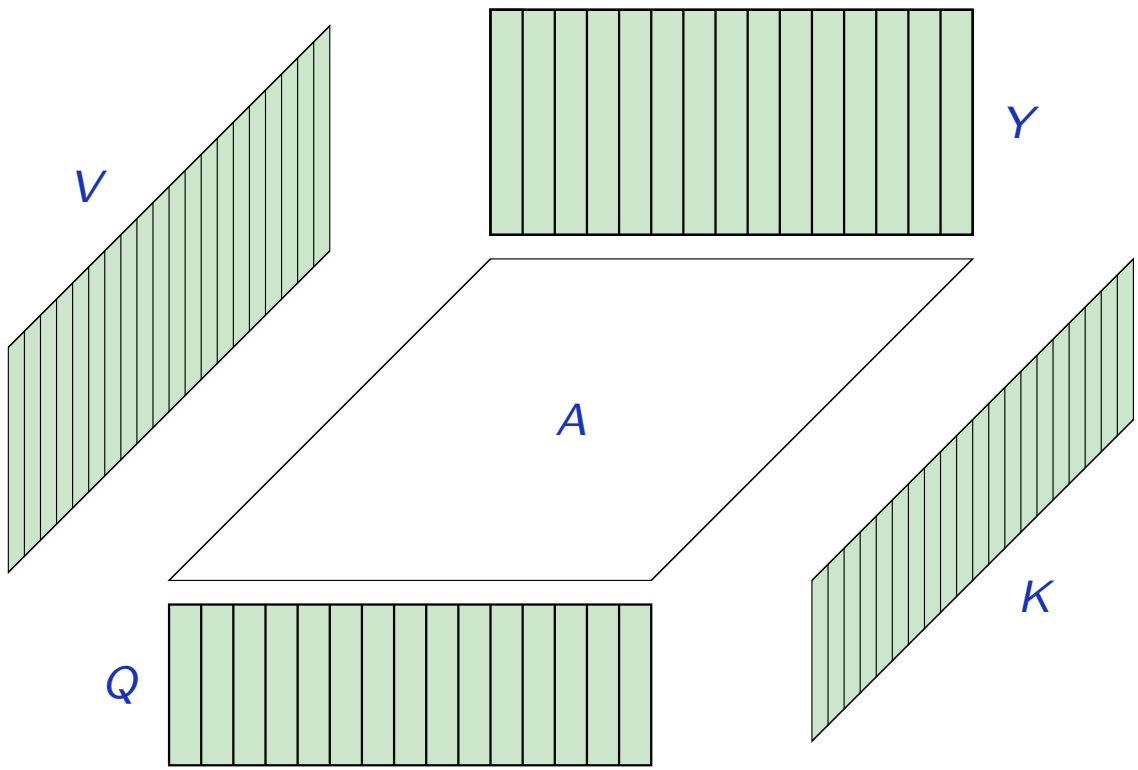
Scaled Dot-Product Attention

- On peut appliquer le processus à plusieurs *queries* en parallèle.



Scaled Dot-Product Attention

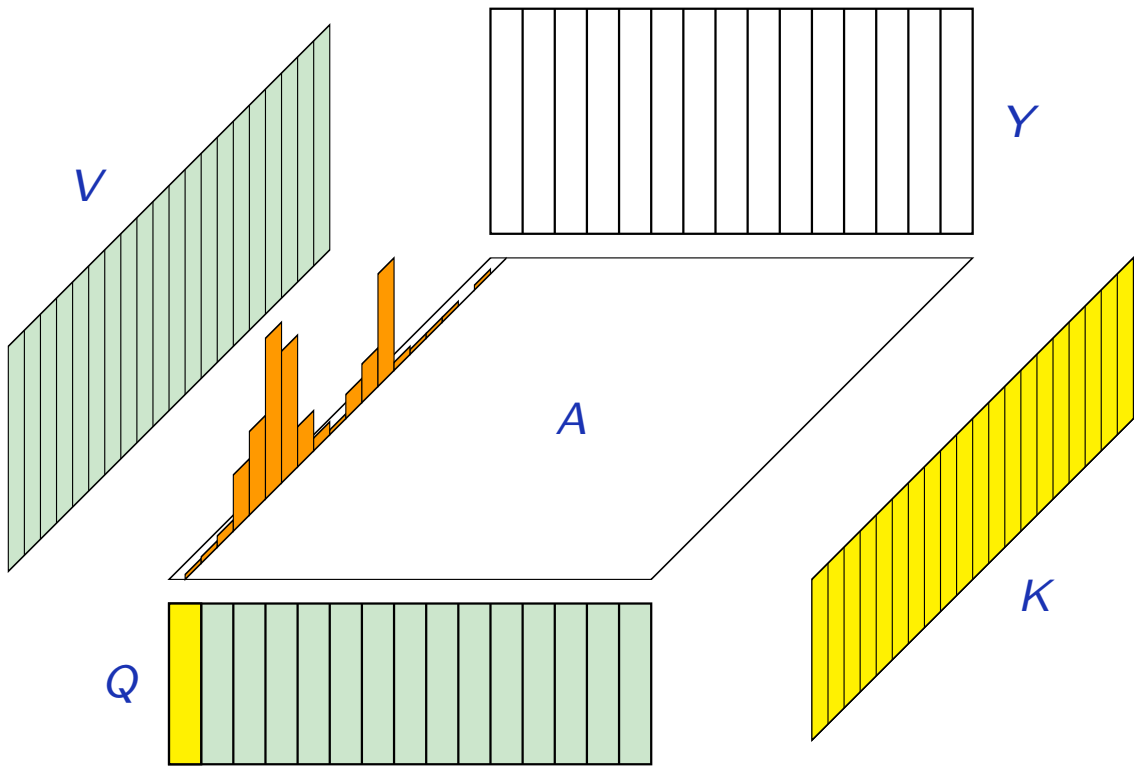
$$A_i = \text{softmax}\left(\frac{KQ_i}{\sqrt{D}}\right) \qquad Y_i = V^\top A_i$$



Figures taken from [Fleuret, 2022].

Scaled Dot-Product Attention

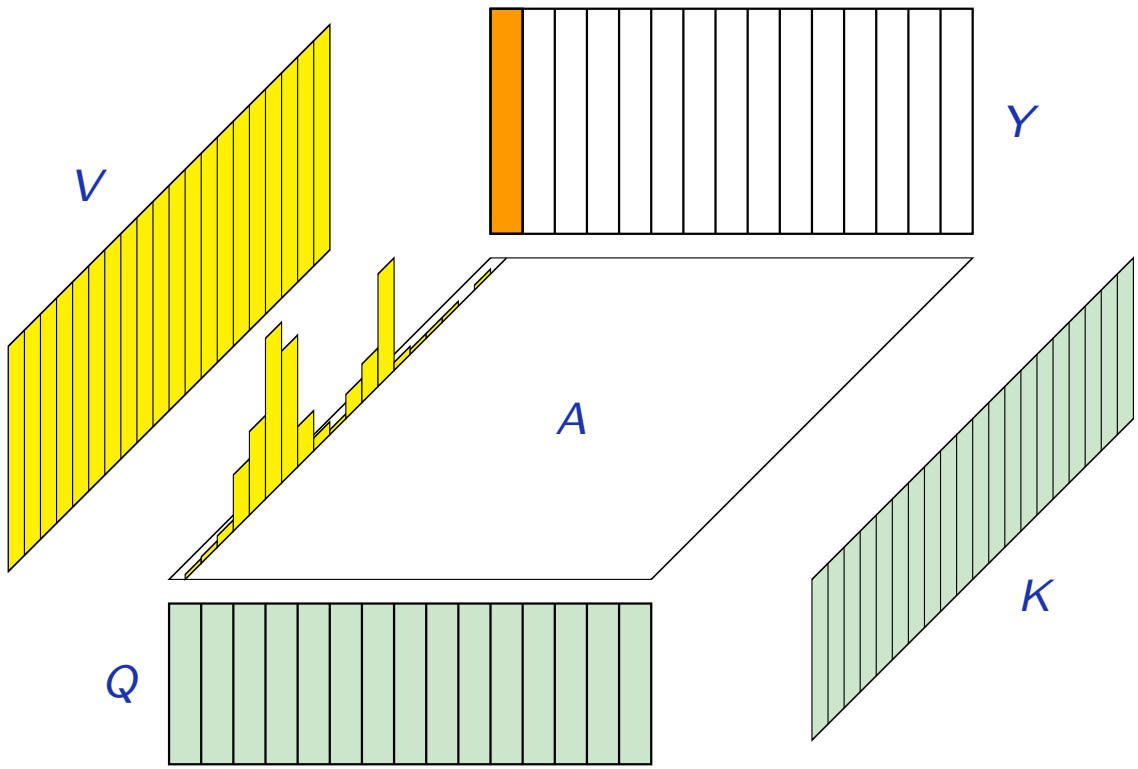
$$A_i = \text{softmax}\left(\frac{KQ_i}{\sqrt{D}}\right)$$



Figures taken from [Fleuret, 2022].

Scaled Dot-Product Attention

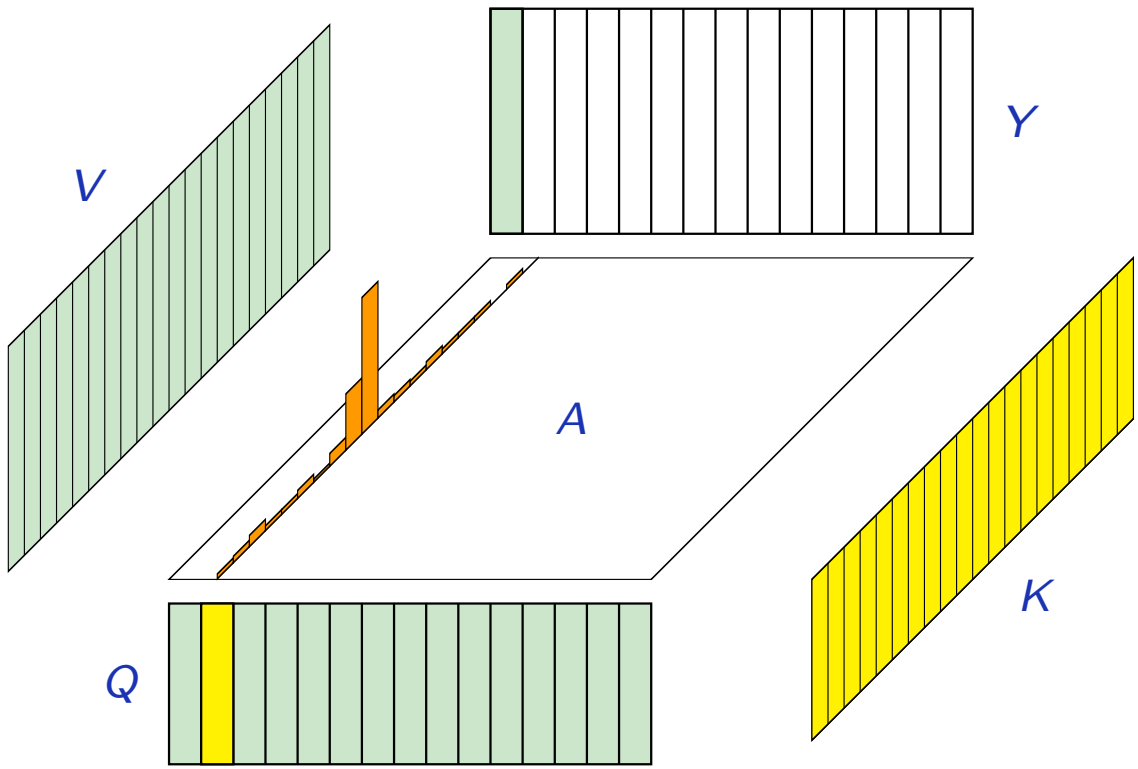
$$Y_i = V^\top A_i$$



Figures taken from [Fleuret, 2022].

Scaled Dot-Product Attention

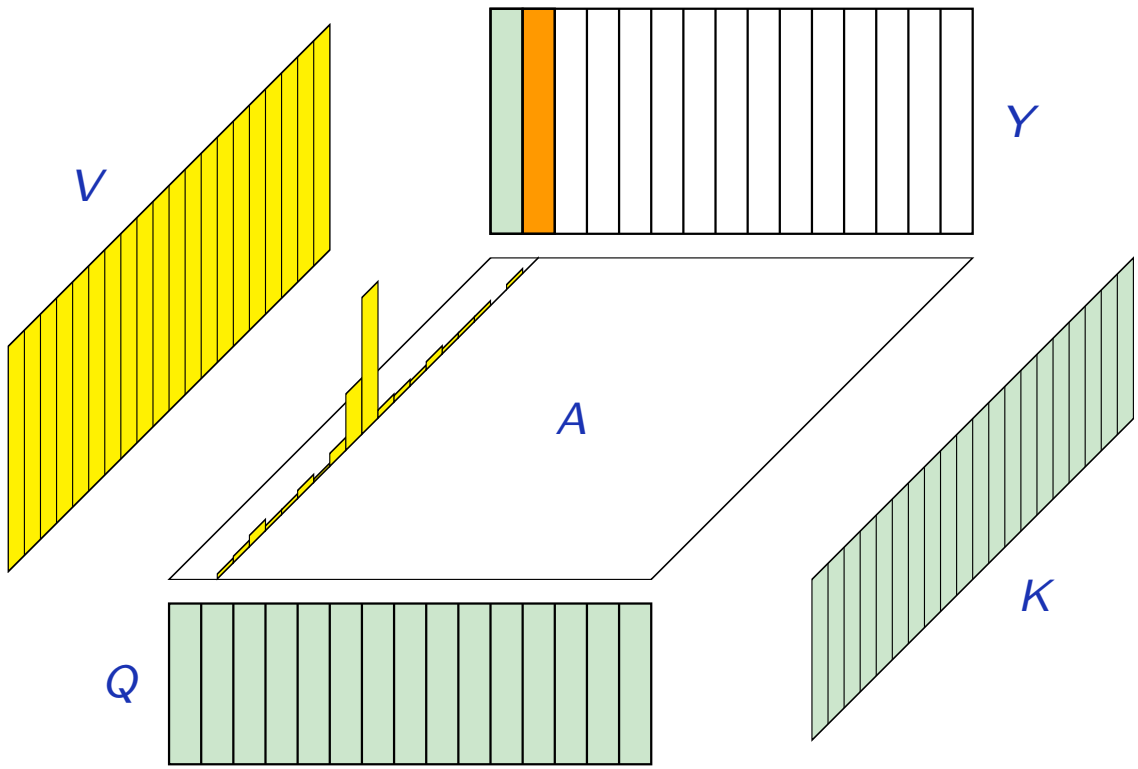
$$A_i = \text{softmax}\left(\frac{KQ_i}{\sqrt{D}}\right)$$



Figures taken from [Fleuret, 2022].

Scaled Dot-Product Attention

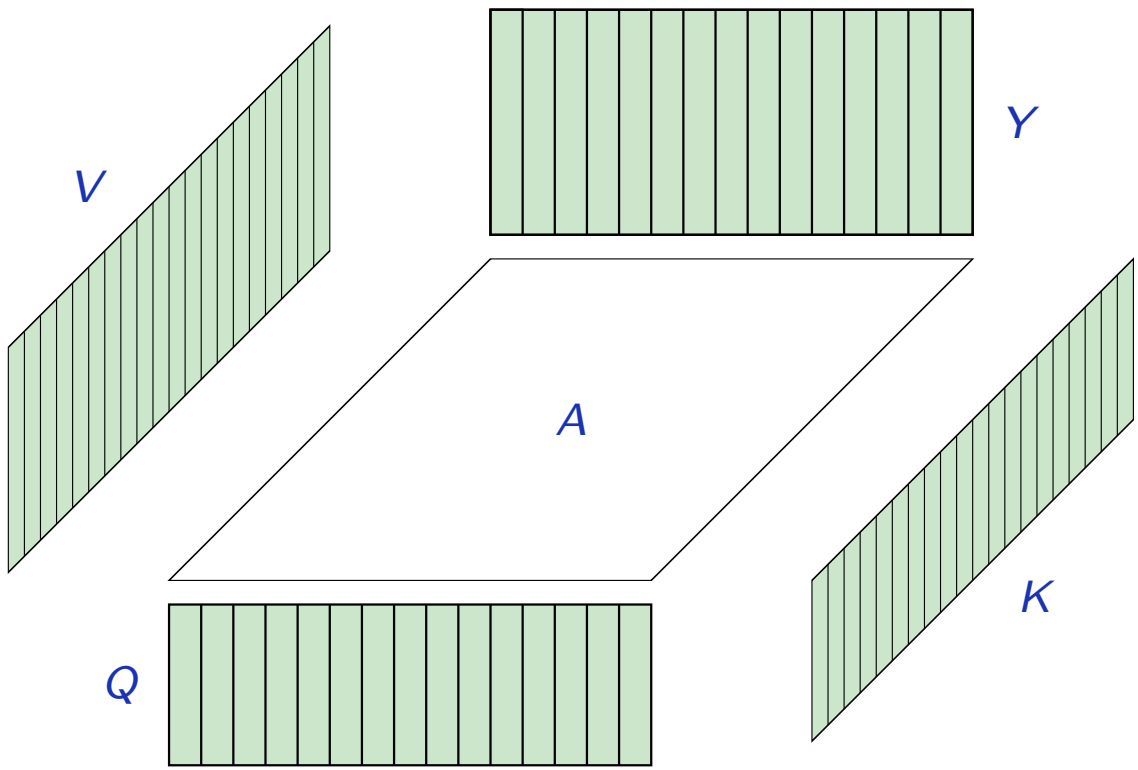
$$Y_i = V^\top A_i$$



Figures taken from [Fleuret, 2022].

Scaled Dot-Product Attention

$$A_i = \text{softmax}\left(\frac{KQ_i}{\sqrt{D}}\right) \qquad Y_i = V^\top A_i$$



Figures taken from [Fleuret, 2022].

Scaled Dot-Product Attention

$$Q = W^Q(X) \quad \text{et} \quad K = W^K(X'), \quad V = W^V(X')$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

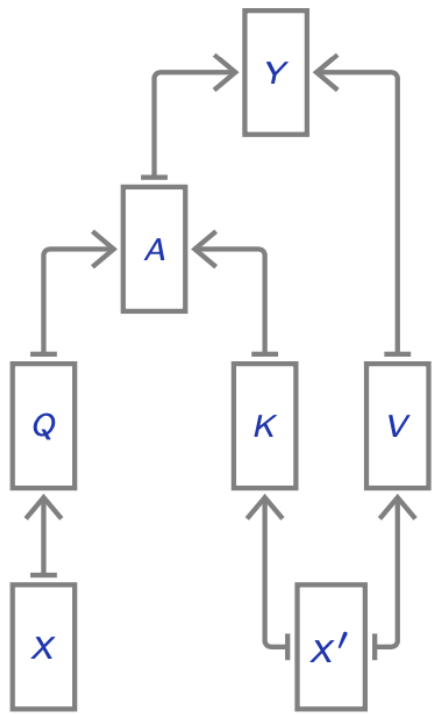
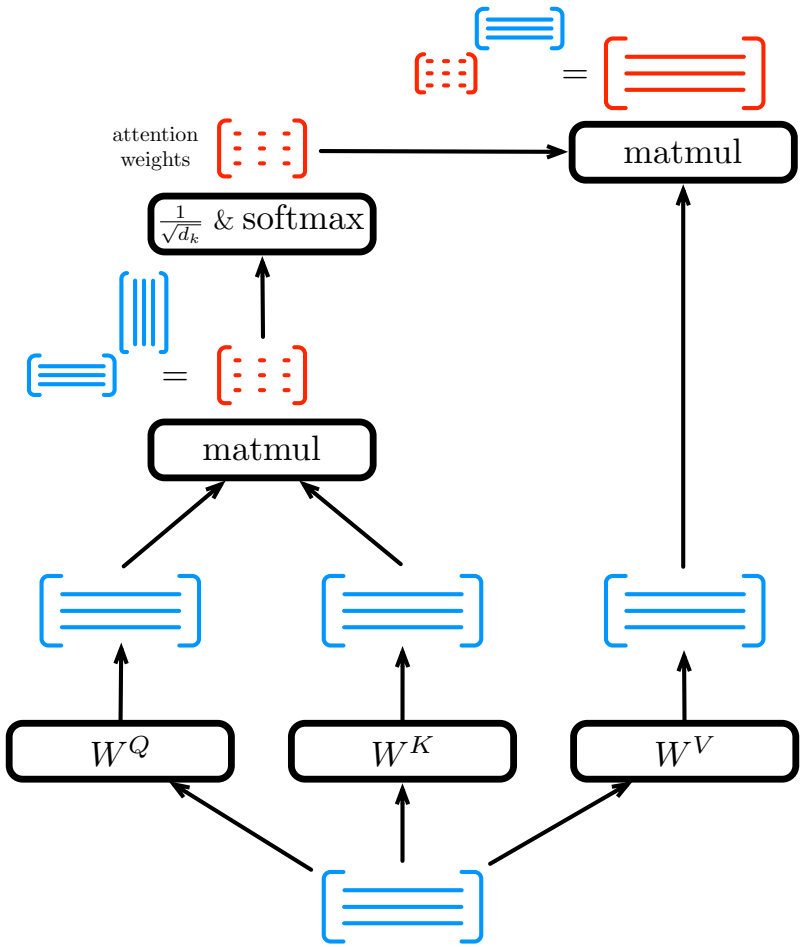


Figure taken from [Fleuret, 2022].

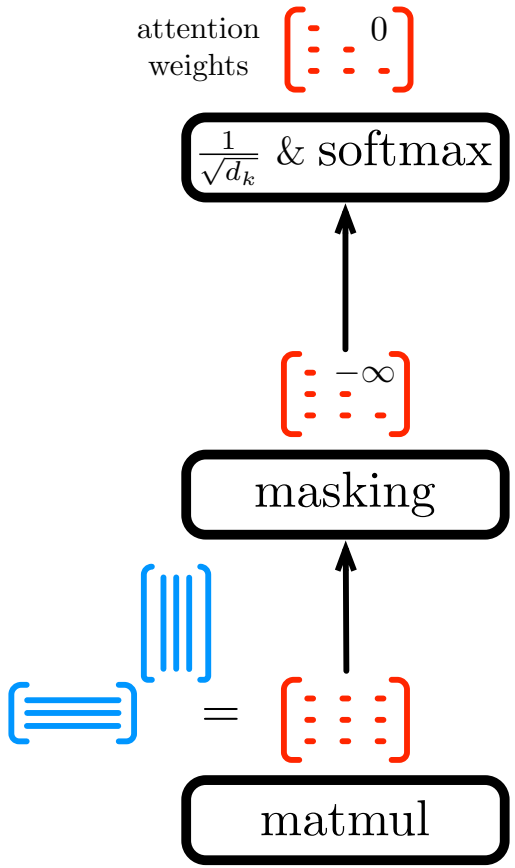
Self-Attention

- Lorsque les *queries* proviennent de la même sequence que les *keys* et les *values*, on parle de **self-attention**.



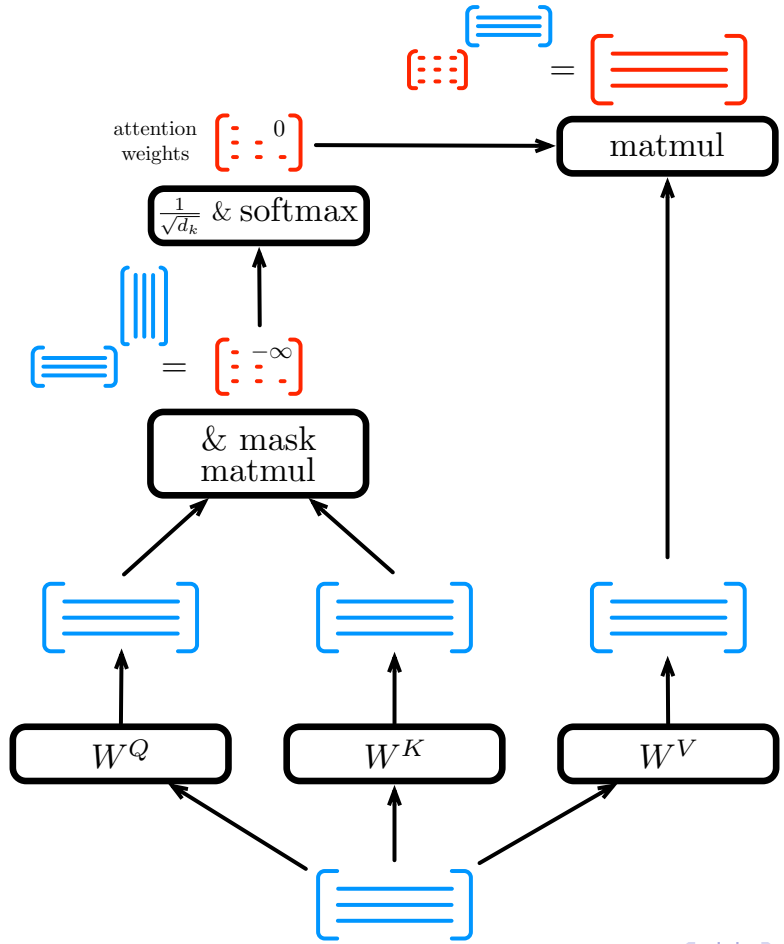
Masked Self-Attention

- Lorsque on interdit aux *queries* de porter de l'attention sur les éléments suivants de la sequence, on parle de **masked self-attention**.



Masked Self-Attention

- Lorsque on interdit aux *queries* de porter de l'attention sur les éléments suivants de la sequence, on parle de **masked self-attention**.



Self-Attention and Masked Self-Attention

$$Q = W^Q(X) \quad , \quad K = W^K(X) \text{ et } V = W^V(X)$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

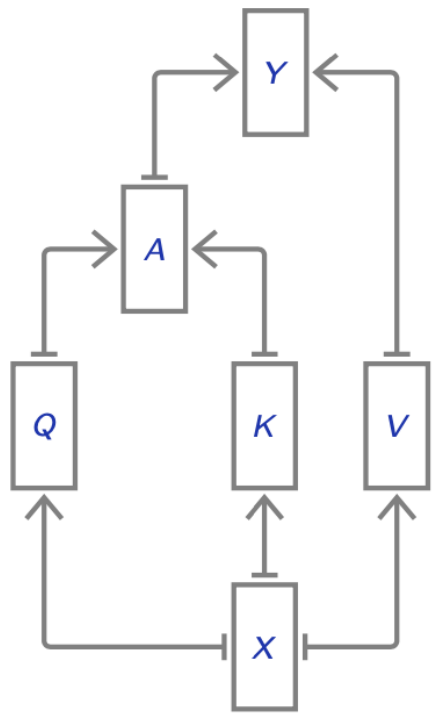


Figure taken from [Fleuret, 2022].

Multiple Attention Heads

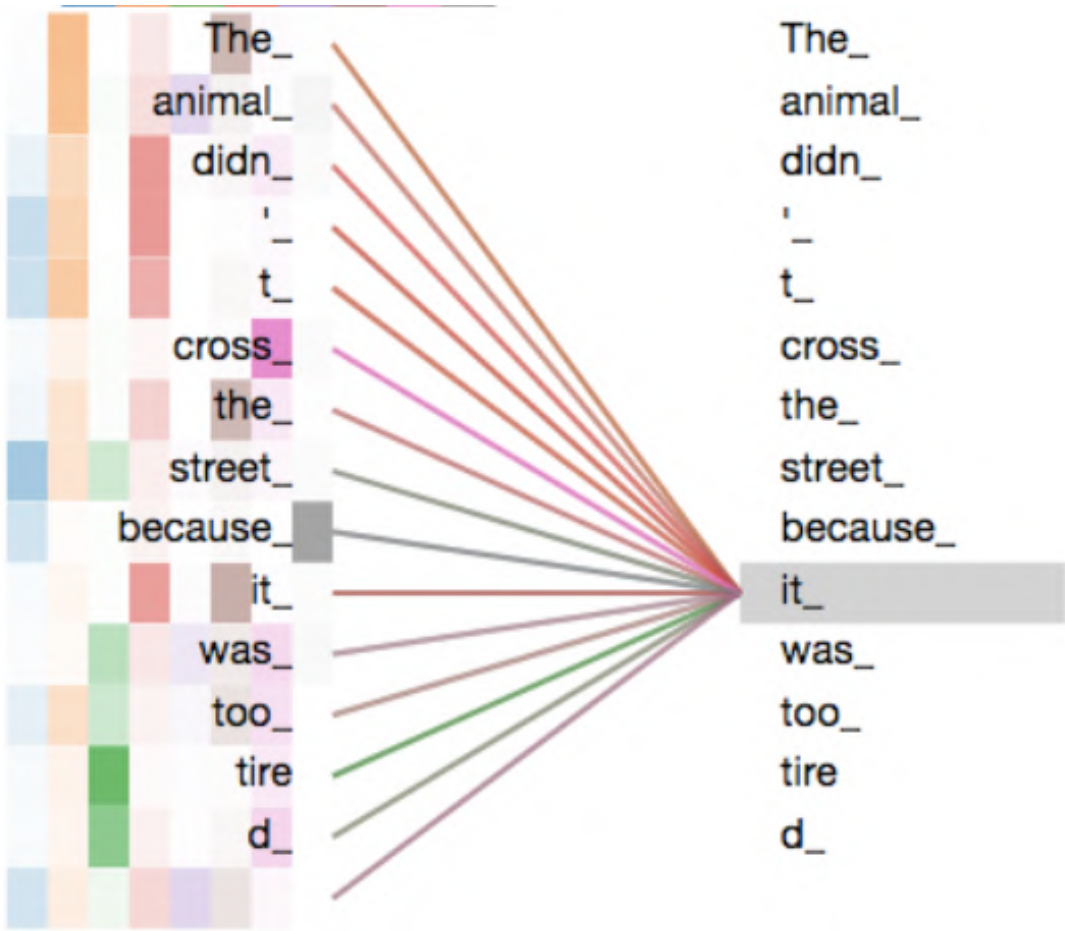


Figure taken from [Alammar, 2018].

Encodeur-décodeur

- L'**encodeur** utilise de la **self-attention** pour "embedder" les inputs.
- Le **décodeur** procède en 2 étapes:

1. **Masked self-attention** pour "embedder" les mots décodés jusqu'à maintenant.
2. **Attention** pour générer le nouveau mot à décoder: la query est l'embedding du dernier mot décodé et les keys et les values sont les embeddings des inputs.

Encodeur-décodeur

- L'**encodeur** utilise de la **self-attention** pour "embedder" les inputs.
- Le **décodeur** procède en 2 étapes:

1. **Masked self-attention** pour "embedder" les mots décodés jusqu'à maintenant.
2. **Attention** pour générer le nouveau mot à décoder: la query est l'embedding du dernier mot décodé et les keys et les values sont les embeddings des inputs.

Encodeur-décodeur

- L'**encodeur** utilise de la **self-attention** pour "embedder" les inputs.
- Le **décodeur** procède en 2 étapes:

1. **Masked self-attention** pour "embedder" les mots décodés jusqu'à maintenant.
2. **Attention** pour générer le nouveau mot à décoder: la query est l'embedding du dernier mot décodé et les keys et les values sont les embeddings des inputs.

Encodeur-décodeur

- L'**encodeur** utilise de la **self-attention** pour "embedder" les inputs.
- Le **décodeur** procède en 2 étapes:

1. **Masked self-attention** pour "embedder" les mots décodés jusqu'à maintenant.
2. **Attention** pour générer le nouveau mot à décoder: la query est l'embedding du dernier mot décodé et les keys et les values sont les embeddings des inputs.

Encodeur-décodeur

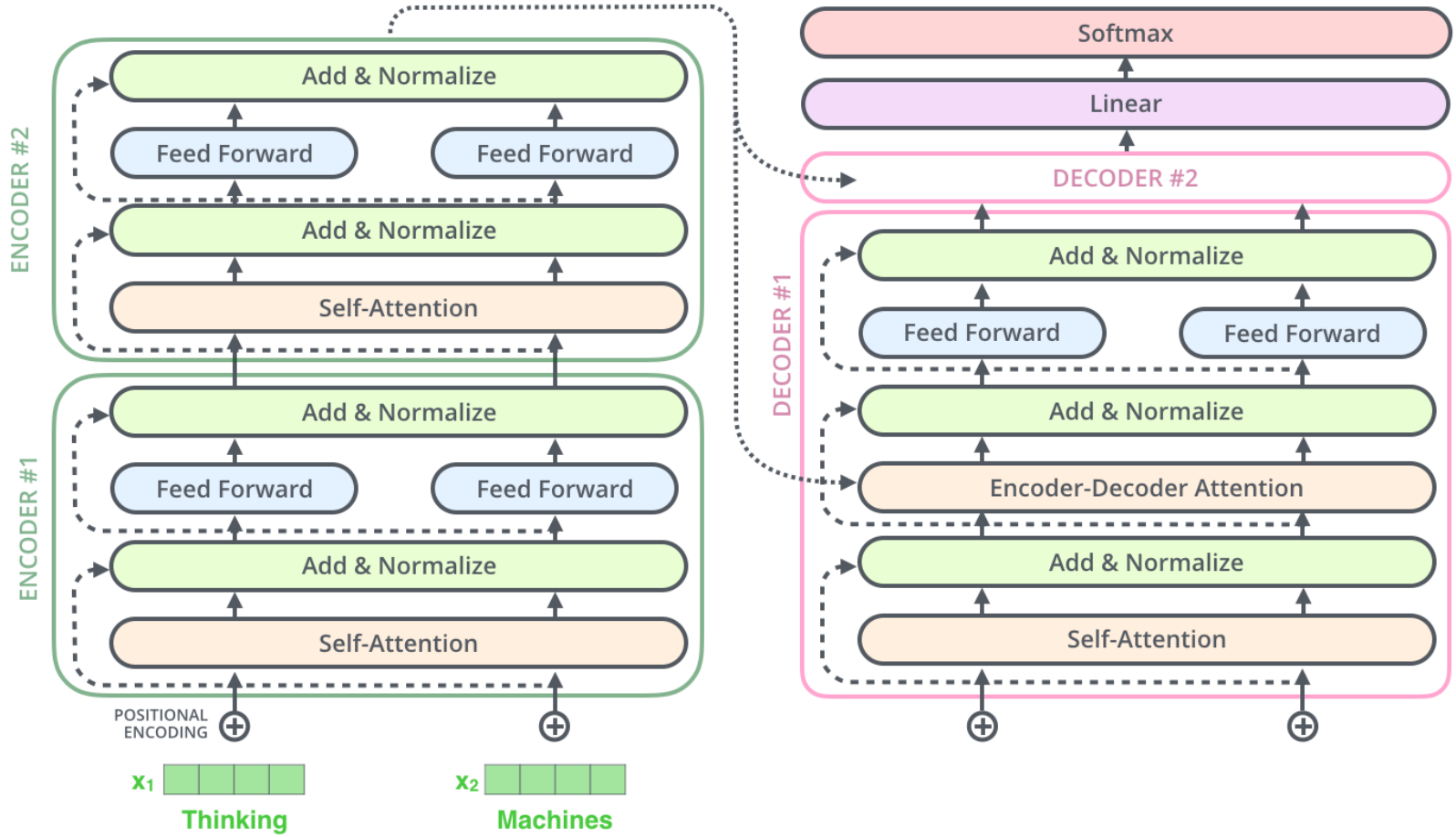


Figure taken from [Alammar, 2018].

Encodeur-décodeur: training

- Le Transformer est entraîné en mode *teacher-forcing*.
- La vraie phase d'output (target sequence) est passée au décodeur dans le but d'être apprise.
- Le décodeur traite la séquence d'input et la vraie phase d'output *en parallèle*, calcule la loss entre prédictions et réalités, et update ses paramètres par backpropagation.
- **Avantages:** parallèlisation et non-accumulation des erreurs lors du processus de décodage.

Encodeur-décodeur: training

- Le Transformer est entraîné en mode *teacher-forcing*.
- La vraie phase d'output (target sequence) est passée au décodeur dans le but d'être apprise.
- Le décodeur traite la séquence d'input et la vraie phase d'output *en parallèle*, calcule la loss entre prédictions et réalités, et update ses paramètres par backpropagation.
- **Avantages:** parallèlisation et non-accumulation des erreurs lors du processus de décodage.

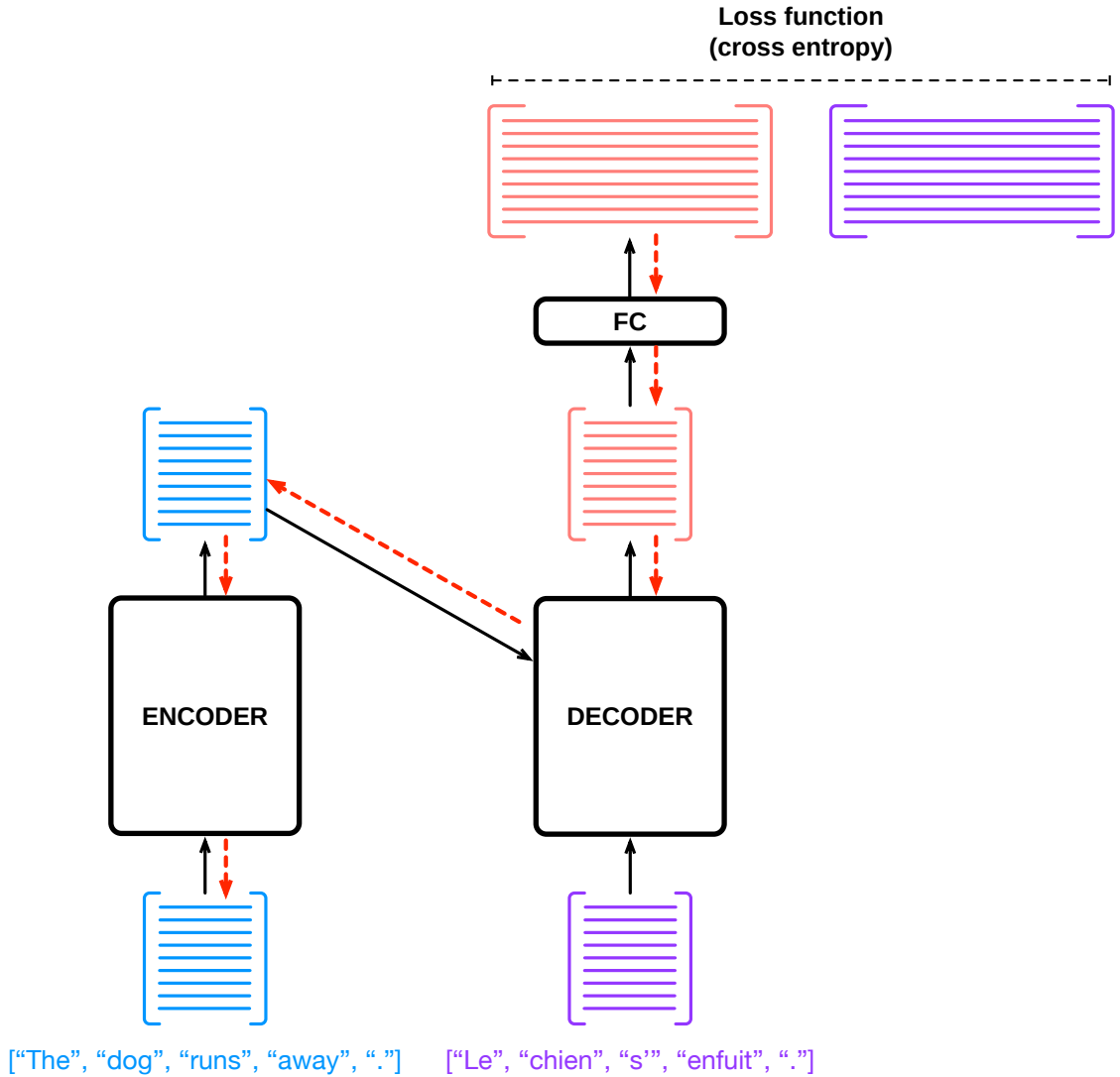
Encodeur-décodeur: training

- Le Transformer est entraîné en mode *teacher-forcing*.
- La vraie phase d'output (target sequence) est passée au décodeur dans le but d'être apprise.
- Le décodeur traite la séquence d'input et la vraie phase d'output *en parallèle*, calcule la loss entre prédictions et réalités, et update ses paramètres par backpropagation.
- **Avantages:** parallèlisation et non-accumulation des erreurs lors du processus de décodage.

Encodeur-décodeur: training

- Le Transformer est entraîné en mode *teacher-forcing*.
- La vraie phase d'output (target sequence) est passée au décodeur dans le but d'être apprise.
- Le décodeur traite la séquence d'input et la vraie phase d'output *en parallèle*, calcule la loss entre prédictions et réalités, et update ses paramètres par backpropagation.
- **Avantages:** parallèlisation et non-accumulation des erreurs lors du processus de décodage.

Encodeur-décodeur: training



Encodeur-décodeur: inférence

- Pour une séquence d'inputs donnée, le Transformer produit une séquence d'outputs *pas à pas*.
- À chaque étape, la séquence d'inputs et la séquence d'outputs décodées jusqu'à maintenant sont passées au décodeur.
- Le décodeur produit alors un nouveau mot qui est appondu à la séquence d'outputs décodées jusqu'à maintenant.
- La séquence d'inputs et cette nouvelle séquence d'outputs décodées jusqu'à maintenant sont re-passées au décodeur qui produit alors un nouveau mot.
- Etc.

Encodeur-décodeur: inférence

- Pour une séquence d'inputs donnée, le Transformer produit une séquence d'outputs *pas à pas*.
- À chaque étape, la séquence d'inputs et la séquence d'outputs décodées jusqu'à maintenant sont passées au décodeur.
- Le décodeur produit alors un nouveau mot qui est appondu à la séquence d'outputs décodées jusqu'à maintenant.
- La séquence d'inputs et cette nouvelle séquence d'outputs décodées jusqu'à maintenant sont re-passées au décodeur qui produit alors un nouveau mot.
- Etc.

Encodeur-décodeur: inférence

- Pour une séquence d'inputs donnée, le Transformer produit une séquence d'outputs *pas à pas*.
- À chaque étape, la séquence d'inputs et la séquence d'outputs décodées jusqu'à maintenant sont passées au décodeur.
- Le décodeur produit alors un nouveau mot qui est appondu à la séquence d'outputs décodées jusqu'à maintenant.
- La séquence d'inputs et cette nouvelle séquence d'outputs décodées jusqu'à maintenant sont re-passées au décodeur qui produit alors un nouveau mot.
- Etc.

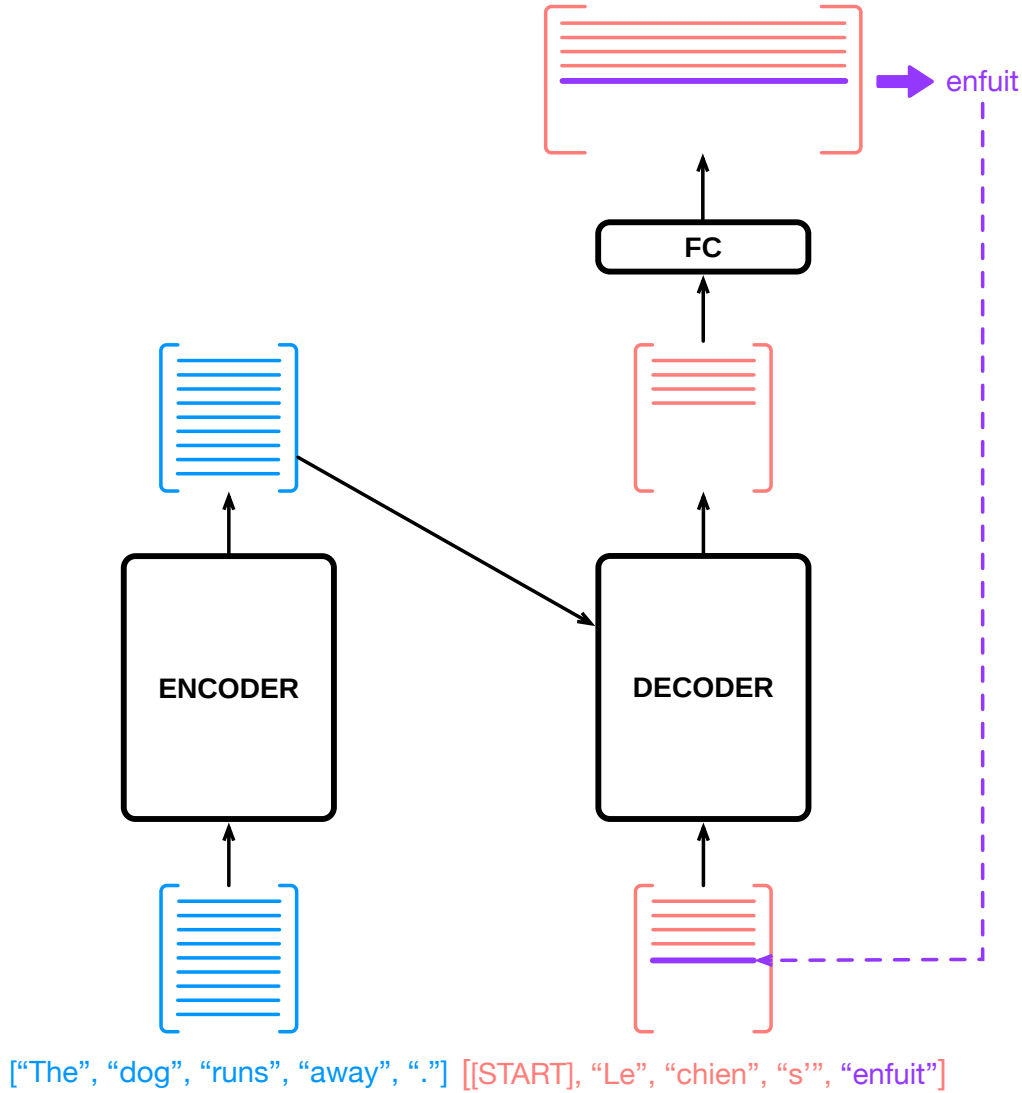
Encodeur-décodeur: inférence

- Pour une séquence d'inputs donnée, le Transformer produit une séquence d'outputs *pas à pas*.
- À chaque étape, la séquence d'inputs et la séquence d'outputs décodées jusqu'à maintenant sont passées au décodeur.
- Le décodeur produit alors un nouveau mot qui est appondu à la séquence d'outputs décodées jusqu'à maintenant.
- La séquence d'inputs et cette nouvelle séquence d'outputs décodées jusqu'à maintenant sont re-passées au décodeur qui produit alors un nouveau mot.
- Etc.

Encodeur-décodeur: inférence

- Pour une séquence d'inputs donnée, le Transformer produit une séquence d'outputs *pas à pas*.
- À chaque étape, la séquence d'inputs et la séquence d'outputs décodées jusqu'à maintenant sont passées au décodeur.
- Le décodeur produit alors un nouveau mot qui est appondu à la séquence d'outputs décodées jusqu'à maintenant.
- La séquence d'inputs et cette nouvelle séquence d'outputs décodées jusqu'à maintenant sont re-passées au décodeur qui produit alors un nouveau mot.
- Etc.

Encodeur-décodeur: inférence



Bibliographie


Alammar, J. (2018).
The illustrated transformer.


Fleuret, F. (2022).
Deep Learning Course.


Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017).
Attention is all you need.
In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.