Transformers et LLMs

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).

⇒ **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).

⇒ **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).

⇒ **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

- Papier original: [Vaswani et al., 2017]
- 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

- Papier original: [Vaswani et al., 2017]
- 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

- Papier original: [Vaswani et al., 2017]
- 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

- Papier original: [Vaswani et al., 2017]
- 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]

# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token 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]", "[PAD]", "[PAD]", "[PAD]"]
# Converting to token ids
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Tokenization et embedding

Tokenization et embedding

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

où pos est la position du input token $(0, 1, 2, \dots)$, i se réfère à la dimension de l'encoding $(0 \leq i \leq 255)$, et $d = 512$.

Tokenization et embedding

Figure taken from [Alammar, 2018]. Exemple de "positional encoding" légèrement différent de celui présenté au slide précédent: les fonctions $\cos$ et $\sin$ ne s'alternent pas en fonction de l'indice i...

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

$$Q = W^Q(X) \qquad K = W^K(X') \qquad V = W^V(X')$$

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

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

- Lorsqu'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

- Lorsqu'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

$$Q = W^Q(X) \qquad K = W^K(X) \qquad V = W^V(X)$$

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

Figure taken from [Fleuret, 2022].

Multiple Attention Heads

Figure taken from [Alammar, 2018].

Exemple

- **Tâche: Détection de motif**
- On donne une suite de 20 lettres parmi A, B, C et D:

  Par exemple [B, A, C, A, D, D, A, B, C, B, C, A, B, A, D, D, A, B, C, B].
- On veut prédire le symbole juste après le dernier A (B ici).
- Un *Recurrent Neural Network (RNN)* ou *Multi Layer Perceptron (MLP)* doit parcourir la séquence entière et mémoriser où se trouve le dernier A.
- Un *mini transformer avec attention* peut apprendre très vite à pointer le dernier A, puis regarder le symbole qui suit.

Exemple

- **Tâche: Détection de motif**
- On donne une suite de 20 lettres parmi A, B, C et D:

  Par exemple [B, A, C, A, D, D, A, B, C, B, C, A, B, A, D, D, A, B, C, B].
- On veut prédire le symbole juste après le dernier A (B ici).
- Un *Recurrent Neural Network (RNN)* ou *Multi Layer Perceptron (MLP)* doit parcourir la séquence entière et mémoriser où se trouve le dernier A.
- Un *mini transformer avec attention* peut apprendre très vite à pointer le dernier A, puis regarder le symbole qui suit.

Exemple

- **Tâche: Détection de motif**
- On donne une suite de 20 lettres parmi A, B, C et D:

  Par exemple [B, A, C, A, D, D, A, B, C, B, C, A, B, A, D, D, A, B, C, B].
- On veut prédire le symbole juste après le dernier A (B ici).
- Un *Recurrent Neural Network (RNN)* ou *Multi Layer Perceptron (MLP)* doit parcourir la séquence entière et mémoriser où se trouve le dernier A.
- Un *mini transformer avec attention* peut apprendre très vite à pointer le dernier A, puis regarder le symbole qui suit.

Exemple

- **Tâche: Détection de motif**
- On donne une suite de 20 lettres parmi A, B, C et D:

  Par exemple [B, A, C, A, D, D, A, B, C, B, C, A, B, A, D, D, A, B, C, B].
- On veut prédire le symbole juste après le dernier A (B ici).
- Un *Recurrent Neural Network (RNN)* ou *Multi Layer Perceptron (MLP)* doit parcourir la séquence entière et mémoriser où se trouve le dernier A.
- Un *mini transformer avec attention* peut apprendre très vite à pointer le dernier A, puis regarder le symbole qui suit.

Exemple

- **Tâche: Détection de motif**
- On donne une suite de 20 lettres parmi A, B, C et D:

  Par exemple [B, A, C, A, D, D, A, B, C, B, C, A, B, A, D, D, A, B, C, B].
- On veut prédire le symbole juste après le dernier A (B ici).
- Un *Recurrent Neural Network (RNN)* ou *Multi Layer Perceptron (MLP)* doit parcourir la séquence entière et mémoriser où se trouve le dernier A.
- Un *mini transformer avec attention* peut apprendre très vite à pointer le dernier A, puis regarder le symbole qui suit.

Exemple

- **Modèle 1: MLP**
- Embedding dim $= 32$ et output dim $= 4$ (nb de lettres)
- 2 fully connected layers: $20 \times 32 \rightarrow 64$ et $64 \rightarrow 4$
- Nb de paramètres $= 41'509$
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 56.4% sur le test set (500 samples)

Exemple

- **Modèle 1: MLP**
- Embedding dim $= 32$ et output dim $= 4$ (nb de lettres)
- 2 fully connected layers: $20 \times 32 \rightarrow 64$ et $64 \rightarrow 4$
- Nb de paramètres $= 41'509$
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 56.4% sur le test set (500 samples)

Exemple

- **Modèle 1: MLP**
- Embedding dim $= 32$ et output dim $= 4$ (nb de lettres)
- 2 fully connected layers: $20 \times 32 \to 64$ et $64 \to 4$
- Nb de paramètres $= 41'509$
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 56.4% sur le test set (500 samples)

Exemple

- **Modèle 1: MLP**
- Embedding dim $= 32$ et output dim $= 4$ (nb de lettres)
- 2 fully connected layers: $20 \times 32 \rightarrow 64$ et $64 \rightarrow 4$
- Nb de paramètres $= 41'509$
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 56.4% sur le test set (500 samples)

Exemple

- **Modèle 1: MLP**
- Embedding dim $= 32$ et output dim $= 4$ (nb de lettres)
- 2 fully connected layers: $20 \times 32 \rightarrow 64$ et $64 \rightarrow 4$
- Nb de paramètres $= 41'509$
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 56.4% sur le test set (500 samples)

Exemple

- **Modèle 1: MLP**
- Embedding dim $= 32$ et output dim $= 4$ (nb de lettres)
- 2 fully connected layers: $20 \times 32 \rightarrow 64$ et $64 \rightarrow 4$
- Nb de paramètres $= 41'509$
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 56.4% sur le test set (500 samples)

Exemple

```python
class SimpleMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.embed = nn.Embedding(VOCAB_SIZE, EMBED_DIM)
        self.fc1 = nn.Linear(SEQ_LEN * EMBED_DIM, HIDDEN_DIM)
        self.fc2 = nn.Linear(HIDDEN_DIM, VOCAB_SIZE)

    def forward(self, x):
        emb = self.embed(x)  # (batch, seq, emb)
        flat = emb.view(x.size(0), -1)
        h = F.relu(self.fc1(flat))
        return self.fc2(h)
```

Exemple

- **Modèle 2: Mini Tranformer**
- Embedding dim $= 32$, nb têtes d'attention $= 1$, nb de transformer blocks $= 3$
- Nb de paramètres $= 39'077$ (à peu près comme le MLP)
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 95.6% sur le test set (500 samples)
- Résultats drastiquement meilleurs que le MLP, mais entraînement plus lent.

Exemple

- **Modèle 2: Mini Tranformer**
- Embedding dim $= 32$, nb têtes d'attention $= 1$, nb de transformer blocks $= 3$
- Nb de paramètres $= 39'077$ (à peu près comme le MLP)
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 95.6% sur le test set (500 samples)
- Résultats drastiquement meilleurs que le MLP, mais entraînement plus lent.

Exemple

- **Modèle 2: Mini Tranformer**
- Embedding dim $= 32$, nb têtes d'attention $= 1$, nb de transformer blocks = 3
- Nb de paramètres $= 39'077$ (à peu près comme le MLP)
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 95.6% sur le test set (500 samples)
- Résultats drastiquement meilleurs que le MLP, mais entraînement plus lent.

Exemple

- **Modèle 2: Mini Tranformer**
- Embedding dim $= 32$, nb têtes d'attention $= 1$, nb de transformer blocks = 3
- Nb de paramètres $= 39'077$ (à peu près comme le MLP)
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 95.6% sur le test set (500 samples)
- Résultats drastiquement meilleurs que le MLP, mais entraînement plus lent.

Exemple

- **Modèle 2: Mini Tranformer**
- Embedding dim $= 32$, nb têtes d'attention $= 1$, nb de transformer blocks = 3
- Nb de paramètres $= 39'077$ (à peu près comme le MLP)
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 95.6% sur le test set (500 samples)
- Résultats drastiquement meilleurs que le MLP, mais entraînement plus lent.

Exemple

- **Modèle 2: Mini Tranformer**
- Embedding dim $= 32$, nb têtes d'attention $= 1$, nb de transformer blocks = 3
- Nb de paramètres $= 39'077$ (à peu près comme le MLP)
- Entraînement: 300 époques sur le train set (1500 samples)
- Résultats: Précision de 95.6% sur le test set (500 samples)
- Résultats drastiquement meilleurs que le MLP, mais entraînement plus lent.

Exemple

```python
class MiniTransformer(nn.Module):
    def __init__(self, vocab_size=VOCAB_SIZE, seq_len=SEQ_LEN, embed_dim=EMBED_DIM, heads=1): # 1 heads only
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.pos_embed = nn.Embedding(seq_len, embed_dim)
        encoder_layer = nn.TransformerEncoderLayer(embed_dim,
                                                   heads,
                                                   dim_feedforward=HIDDEN_DIM*2,  # augmented dim
                                                   batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=3)         # 3 blocks!
        self.fc = nn.Linear(embed_dim, vocab_size)

    def forward(self, x):
        pos_ids = torch.arange(x.shape[1], device=x.device).unsqueeze(0)
        emb = self.embed(x) + self.pos_embed(pos_ids)
        out = self.encoder(emb)
        pooled = out[:, -1, :]
        return self.fc(pooled)
```

Exemple: visualisation des poids d'attention

Exemple: visualisation des poids d'attention

Exemple: visualisation des poids d'attention

Exemple: visualisation des poids d'attention

Exemple: visualisation des poids d'attention

Exemple: visualisation des poids d'attention

Architectures

- Le premier Transformer était une architecture encodeur-décodeur. En pratique, on utilise d'autres architectures également...
- Architecture **encoder-only**, composée de *blocs d'encodeurs* empilés les uns sur les autres.
- Architecture **encoder-decoder**, composée de *blocs d'encodeurs* empilés suivis de *blocs de décodeurs* empilés.
- Architecture **decoder-only**, architecture composée de *blocs de décodeurs* empilés.

Architectures

- Le premier Transformer était une architecture encodeur-décodeur. En pratique, on utilise d'autres architectures également...
- Architecture **encoder-only**, composée de *blocs d'encodeurs* empilés les uns sur les autres.
- Architecture **encoder-decoder**, composée de *blocs d'encodeurs* empilés suivis de *blocs de décodeurs* empilés.
- Architecture **decoder-only**, architecture composée de *blocs de décodeurs* empilés.

Architectures

- Le premier Transformer était une architecture encodeur-décodeur. En pratique, on utilise d'autres architectures également...
- Architecture **encoder-only**, composée de *blocs d'encodeurs* empilés les uns sur les autres.
- Architecture **encoder-decoder**, composée de *blocs d'encodeurs* empilés suivis de *blocs de décodeurs* empilés.
- Architecture **decoder-only**, architecture composée de *blocs de décodeurs* empilés.

Architectures

- Le premier Transformer était une architecture encodeur-décodeur. En pratique, on utilise d'autres architectures également...
- Architecture **encoder-only**, composée de *blocs d'encodeurs* empilés les uns sur les autres.
- Architecture **encoder-decoder**, composée de *blocs d'encodeurs* empilés suivis de *blocs de décodeurs* empilés.
- Architecture **decoder-only**, architecture composée de *blocs de décodeurs* empilés.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

- **Modèles d'encodage** ou **Embedders**, composés de *blocks d'encodeurs* (BERT, RoBERTa, ChemBERTa, etc.).
- Apprennent des représentations contextuelles des tokens ou des séquences complètes.
- Pour des tâches de compréhension (et non de génération).
  - Sequence classification: sentiment analysis, spam detection.
  - Token classification: named entity recognition (NER), POS tagging
  - Semantic similarity / retrieval: recherche sémantique, embeddings.
  - Property prediction: prédiction de propriétés moléculaires.

Encoder-only

Figure taken from [Alammar, 2018].

Encoder-only: entraînement

- Le modèle est entraîné en mode **supervisé** classique.
- L'encodeur produit une représentation contextuelle de la séquence d'entrée, transmise à un **MLP de sortie**: tête de classification ou de régression.
- Le modèle calcule la loss entre les prédictions et les labels.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** simplicité d'entraînement et convergence rapide.

Encoder-only: entraînement

- Le modèle est entraîné en mode **supervisé** classique.
- L'encodeur produit une représentation contextuelle de la séquence d'entrée, transmise à un **MLP de sortie**: tête de classification ou de régression.
- Le modèle calcule la loss entre les prédictions et les labels.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** simplicité d'entraînement et convergence rapide.

Encoder-only: entraînement

- Le modèle est entraîné en mode **supervisé** classique.
- L'encodeur produit une représentation contextuelle de la séquence d'entrée, transmise à un **MLP de sortie**: tête de classification ou de régression.
- Le modèle calcule la loss entre les prédictions et les labels.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** simplicité d'entraînement et convergence rapide.

Encoder-only: entraînement

- Le modèle est entraîné en mode **supervisé** classique.
- L'encodeur produit une représentation contextuelle de la séquence d'entrée, transmise à un **MLP de sortie**: tête de classification ou de régression.
- Le modèle calcule la loss entre les prédictions et les labels.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** simplicité d'entraînement et convergence rapide.

Encoder-only: entraînement

- Le modèle est entraîné en mode **supervisé** classique.
- L'encodeur produit une représentation contextuelle de la séquence d'entrée, transmise à un **MLP de sortie**: tête de classification ou de régression.
- Le modèle calcule la loss entre les prédictions et les labels.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** simplicité d'entraînement et convergence rapide.

Encoder-only: entraînement

Encoder-only: inférence

[“The”, “dog”, “runs”, “away”, “.”]

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

- **Modèles sequence-to-sequence**, composés de *blocs d'encodeurs et de décodeurs* (T5, BART, mBART, etc.).
- Apprennent une transformation de sequences en séquences.
- Pour des tâches de transformation de texte à texte.
  - Machine translation: traduction automatique entre langues.
  - Summarization: génération de résumés de documents.
  - Question answering: réponse générative à une question.
  - Paraphrase generation: reformulation de texte.

Encoder-decoder

Figure taken from [Alammar, 2018].

Encoder-decoder: entraînement

- Le modèle est entraîné en mode **teacher forcing**.
- La séquence cible (target sequence) est fournie au décodeur, décalée d'un token de début `[START]`, afin d'apprendre à prédire le token suivant à chaque étape.
- L'encodeur traite la séquence d'entrée, le décodeur reçoit les tokens cibles décalés, et le modèle calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et absence d'accumulation d'erreurs pendant l'apprentissage.

Encoder-decoder: entraînement

- Le modèle est entraîné en mode **teacher forcing**.
- La séquence cible (target sequence) est fournie au décodeur, décalée d'un token de début `[START]`, afin d'apprendre à prédire le token suivant à chaque étape.
- L'encodeur traite la séquence d'entrée, le décodeur reçoit les tokens cibles décalés, et le modèle calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et absence d'accumulation d'erreurs pendant l'apprentissage.

Encoder-decoder: entraînement

- Le modèle est entraîné en mode **teacher forcing**.
- La séquence cible (target sequence) est fournie au décodeur, décalée d'un token de début `[START]`, afin d'apprendre à prédire le token suivant à chaque étape.
- L'encodeur traite la séquence d'entrée, le décodeur reçoit les tokens cibles décalés, et le modèle calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et absence d'accumulation d'erreurs pendant l'apprentissage.

Encoder-decoder: entraînement

- Le modèle est entraîné en mode **teacher forcing**.
- La séquence cible (target sequence) est fournie au décodeur, décalée d'un token de début `[START]`, afin d'apprendre à prédire le token suivant à chaque étape.
- L'encodeur traite la séquence d'entrée, le décodeur reçoit les tokens cibles décalés, et le modèle calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et absence d'accumulation d'erreurs pendant l'apprentissage.

Encoder-decoder: entraînement

- Le modèle est entraîné en mode **teacher forcing**.
- La séquence cible (target sequence) est fournie au décodeur, décalée d'un token de début `[START]`, afin d'apprendre à prédire le token suivant à chaque étape.
- L'encodeur traite la séquence d'entrée, le décodeur reçoit les tokens cibles décalés, et le modèle calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et absence d'accumulation d'erreurs pendant l'apprentissage.

Encoder-decoder: entraînement

Encoder-decoder: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- L'encodeur traite la séquence d'entrée une seule fois et produit ses représentations contextuelles.
- À chaque étape, le décodeur reçoit les représentations de l'entrée, les tokens de sortie déjà générés, et prédit le token suivant.
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling.

Encoder-decoder: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- L'encodeur traite la séquence d'entrée une seule fois et produit ses représentations contextuelles.
- À chaque étape, le décodeur reçoit les représentations de l'entrée, les tokens de sortie déjà générés, et prédit le token suivant.
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling.

Encoder-decoder: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- L'encodeur traite la séquence d'entrée une seule fois et produit ses représentations contextuelles.
- À chaque étape, le décodeur reçoit les représentations de l'entrée, les tokens de sortie déjà générés, et prédit le token suivant.
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling.

Encoder-decoder: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- L'encodeur traite la séquence d'entrée une seule fois et produit ses représentations contextuelles.
- À chaque étape, le décodeur reçoit les représentations de l'entrée, les tokens de sortie déjà générés, et prédit le token suivant.
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling.

Encoder-decoder: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- L'encodeur traite la séquence d'entrée une seule fois et produit ses représentations contextuelles.
- À chaque étape, le décodeur reçoit les représentations de l'entrée, les tokens de sortie déjà générés, et prédit le token suivant.
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling.

Encoder-decoder: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- L'encodeur traite la séquence d'entrée une seule fois et produit ses représentations contextuelles.
- À chaque étape, le décodeur reçoit les représentations de l'entrée, les tokens de sortie déjà générés, et prédit le token suivant.
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling.

Encoder-decoder: inférence

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

- **Modèles génératifs auto-régressifs**, composés de *blocs de décodeurs* uniquement (GPT, LLaMA, Mistral, etc.).
- Apprennent à générer des séquences token par token, en se basant sur les tokens précédents: mode autorégressif.
- Pour des tâches de génération libre.
  - Language modeling: prédiction du prochain mot dans une séquence.
  - Text generation: génération libre de texte, complétion de phrase.
  - Dialogue / chat: agents conversationnels, assistants.
  - Code generation: génération automatique de code.

Decoder-only

Figure taken from [Alammar, 2018].

Decoder-only: entraînement

- Le modèle est entraîné en mode **auto-régressif** (**causal language modeling**).
- La séquence d'entrée contient à la fois le contexte et les cibles à prédire, décalées d'un token de début `[START]`.
- Le modèle reçoit les tokens décalés, applique une **masked self-attention** pour ne voir que les tokens précédents, et calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et apprentissage direct du mode auto-régressif.

Decoder-only: entraînement

- Le modèle est entraîné en mode **auto-régressif** (**causal language modeling**).
- La séquence d'entrée contient à la fois le contexte et les cibles à prédire, décalées d'un token de début `[START]`.
- Le modèle reçoit les tokens décalés, applique une **masked self-attention** pour ne voir que les tokens précédents, et calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et apprentissage direct du mode auto-régressif.

Decoder-only: entraînement

- Le modèle est entraîné en mode **auto-régressif** (**causal language modeling**).
- La séquence d'entrée contient à la fois le contexte et les cibles à prédire, décalées d'un token de début `[START]`.
- Le modèle reçoit les tokens décalés, applique une **masked self-attention** pour ne voir que les tokens précédents, et calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et apprentissage direct du mode auto-régressif.

Decoder-only: entraînement

- Le modèle est entraîné en mode **auto-régressif** (**causal language modeling**).
- La séquence d'entrée contient à la fois le contexte et les cibles à prédire, décalées d'un token de début `[START]`.
- Le modèle reçoit les tokens décalés, applique une **masked self-attention** pour ne voir que les tokens précédents, et calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et apprentissage direct du mode auto-régressif.

Decoder-only: entraînement

- Le modèle est entraîné en mode **auto-régressif** (**causal language modeling**).
- La séquence d'entrée contient à la fois le contexte et les cibles à prédire, décalées d'un token de début `[START]`.
- Le modèle reçoit les tokens décalés, applique une **masked self-attention** pour ne voir que les tokens précédents, et calcule la loss entre les prédictions et les tokens cibles non décalés.
- Les paramètres sont mis à jour par **backpropagation**.
- **Avantages:** entraînement parallèle sur toute la séquence et apprentissage direct du mode auto-régressif.

Decoder-only: entraînement

Decoder-only: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- À chaque étape, il reçoit les tokens déjà générés et prédit le token suivant.
- La **masked self-attention** garantit que chaque token ne voit que les tokens précédents (et non futurs).
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling, top-k, nucleus sampling.

Decoder-only: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- À chaque étape, il reçoit les tokens déjà générés et prédit le token suivant.
- La **masked self-attention** garantit que chaque token ne voit que les tokens précédents (et non futurs).
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling, top-k, nucleus sampling.

Decoder-only: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- À chaque étape, il reçoit les tokens déjà générés et prédit le token suivant.
- La **masked self-attention** garantit que chaque token ne voit que les tokens précédents (et non futurs).
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling, top-k, nucleus sampling.

Decoder-only: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- À chaque étape, il reçoit les tokens déjà générés et prédit le token suivant.
- La **masked self-attention** garantit que chaque token ne voit que les tokens précédents (et non futurs).
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling, top-k, nucleus sampling.

Decoder-only: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- À chaque étape, il reçoit les tokens déjà générés et prédit le token suivant.
- La **masked self-attention** garantit que chaque token ne voit que les tokens précédents (et non futurs).
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling, top-k, nucleus sampling.

Decoder-only: inférence

- **Mode auto-régressif:** le modèle génère une séquence de texte *pas à pas*, un token à la fois.
- À chaque étape, il reçoit les tokens déjà générés et prédit le token suivant.
- La **masked self-attention** garantit que chaque token ne voit que les tokens précédents (et non futurs).
- Le nouveau token est ajouté aux tokens de sortie déjà générés, qui sont réinjectés dans le décodeur.
- Le processus continue jusqu'à la production du token `[END]`.
- **Stratégies de décodage:** greedy decoding, beam search, sampling, top-k, nucleus sampling.

Decoder-only: inférence

Training tasks: pre-training

(1) Causal Language Modeling (Causal LM)

- Tâche principale pour entraîner les LLMs *génératifs* de type *decoder-only*, tels que GPT.
- **Objectif:** "Next-token prediction": étant donnée une suite de tokens $\boldsymbol{x}_{<t} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1})$, il faut prédire le token suivant $\boldsymbol{x}_t$. On n'utilise que le *contexte uni-directionnel* du passé: *mode autorégressif*.
- Formellement, le modèle apprend la distribution de langage:

$$P(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \prod_{t=0}^{n} P(\boldsymbol{x}_t \mid \boldsymbol{x}_{<t})$$

- **Exemple:** `'The cat sat on the [MASK]'` $\longrightarrow$ `'mat'`
- **Loss:** cross-entropy loss

$$\mathcal{L} = -\sum_{t} \log P(\boldsymbol{x}_t \mid \boldsymbol{x}_{<t})$$

Training tasks: pre-training

(1) Causal Language Modeling (Causal LM)

- Tâche principale pour entraîner les LLMs *génératifs* de type *decoder-only*, tels que GPT.
- **Objectif:** "Next-token prediction": étant donnée une suite de tokens $x_{<t} = (x_0, x_1, \ldots, x_{t-1})$, il faut prédire le token suivant x_t. On n'utilise que le *contexte uni-directionnel* du passé: *mode autorégressif*.
- Formellement, le modèle apprend la distribution de langage:

$$P(x_0, x_1, \ldots, x_n) = \prod_{t=0}^{n} P(x_t \mid x_{<t})$$

- **Exemple:** `'The cat sat on the [MASK]'` $\longrightarrow$ `'mat'`
- **Loss:** cross-entropy loss

$$\mathcal{L} = -\sum_t \log P(x_t \mid x_{<t})$$

Training tasks: pre-training

(1) Causal Language Modeling (Causal LM)

- Tâche principale pour entraîner les LLMs *génératifs* de type *decoder-only*, tels que GPT.
- **Objectif:** "Next-token prediction": étant donnée une suite de tokens $\boldsymbol{x_{<t}} = (\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_{t-1}})$, il faut prédire le token suivant $\boldsymbol{x_t}$. On n'utilise que le *contexte uni-directionnel* du passé: *mode autorégressif*.
- Formellement, le modèle apprend la distribution de langage:

$$P(\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) = \prod_{t=0}^{n} P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

- **Exemple:** `'The cat sat on the [MASK]'` $\longrightarrow$ `'mat'`
- **Loss:** cross-entropy loss

$$\mathcal{L} = -\sum_{t} \log P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

Training tasks: pre-training

(1) Causal Language Modeling (Causal LM)

- Tâche principale pour entraîner les LLMs *génératifs* de type *decoder-only*, tels que GPT.
- **Objectif:** "Next-token prediction": étant donnée une suite de tokens $\boldsymbol{x_{<t}} = (\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_{t-1}})$, il faut prédire le token suivant $\boldsymbol{x_t}$. On n'utilise que le *contexte uni-directionnel* du passé: *mode autorégressif*.
- Formellement, le modèle apprend la distribution de langage:

$$P(\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) = \prod_{t=0}^{n} P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

- **Exemple:** `'The cat sat on the [MASK]'` $\longrightarrow$ `'mat'`
- **Loss:** cross-entropy loss

$$\mathcal{L} = -\sum_{t} \log P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

Training tasks: pre-training

(1) Causal Language Modeling (Causal LM)

- Tâche principale pour entraîner les LLMs *génératifs* de type *decoder-only*, tels que GPT.
- **Objectif:** "Next-token prediction": étant donnée une suite de tokens $\boldsymbol{x_{<t}} = (\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_{t-1}})$, il faut prédire le token suivant $\boldsymbol{x_t}$. On n'utilise que le *contexte uni-directionnel* du passé: *mode autorégressif*.
- Formellement, le modèle apprend la distribution de langage:

$$P(\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) = \prod_{t=0}^{n} P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

- **Exemple:** `'The cat sat on the [MASK]'` $\longrightarrow$ `'mat'`
- **Loss:** cross-entropy loss

$$\mathcal{L} = -\sum_{t} \log P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

Training tasks: pre-training

(1) Causal Language Modeling (Causal LM)

- Tâche principale pour entraîner les LLMs *génératifs* de type *decoder-only*, tels que GPT.
- **Objectif:** "Next-token prediction": étant donnée une suite de tokens $\boldsymbol{x_{<t}} = (\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_{t-1}})$, il faut prédire le token suivant $\boldsymbol{x_t}$. On n'utilise que le *contexte uni-directionnel* du passé: *mode autorégressif*.
- Formellement, le modèle apprend la distribution de langage:

$$P(\boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) = \prod_{t=0}^{n} P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

- **Exemple:** `'The cat sat on the [MASK]'` $\longrightarrow$ `'mat'`
- **Loss:** cross-entropy loss

$$\mathcal{L} = -\sum_{t} \log P(\boldsymbol{x_t} \mid \boldsymbol{x_{<t}})$$

Training tasks: pre-training

(2) Mask-Language Modeling (MLM)

- Pour entraîner les LLMs *non génératifs* de type *encoder-only* , tels que BERT, RoBERTa, etc.
- **Objectif:** étant donnée une suite de tokens, on masque des tokens au hasard (15%) et on demande de prédire ces tokens en utilisant le *contexte bi-directionnel* (passé et futur).
- **Exemple:** `'The [MASK] sat on the [MASK]'` $\longrightarrow$ `'cat'`, `'mat'`
- **Loss:** cross-entropy loss

Training tasks: pre-training

(2) Mask-Language Modeling (MLM)

- Pour entraîner les LLMs *non génératifs* de type *encoder-only* , tels que BERT, RoBERTa, etc.
- **Objectif:** étant donnée une suite de tokens, on masque des tokens au hasard (15%) et on demande de prédire ces tokens en utilisant le *contexte bi-directionnel* (passé et futur).
- **Exemple:** `'The [MASK] sat on the [MASK]'` $\longrightarrow$ `'cat'`, `'mat'`
- **Loss:** cross-entropy loss

Training tasks: pre-training

(2) Mask-Language Modeling (MLM)

- Pour entraîner les LLMs *non génératifs* de type *encoder-only* , tels que BERT, RoBERTa, etc.
- **Objectif:** étant donnée une suite de tokens, on masque des tokens au hasard (15%) et on demande de prédire ces tokens en utilisant le *contexte bi-directionnel* (passé et futur).
- **Exemple:** `'The [MASK] sat on the [MASK]'` $\longrightarrow$ `'cat'`, `'mat'`
- **Loss:** cross-entropy loss

Training tasks: pre-training

(2) Mask-Language Modeling (MLM)

- Pour entraîner les LLMs *non génératifs* de type *encoder-only* , tels que BERT, RoBERTa, etc.
- **Objectif:** étant donnée une suite de tokens, on masque des tokens au hasard (15%) et on demande de prédire ces tokens en utilisant le *contexte bi-directionnel* (passé et futur).
- **Exemple:** `'The [MASK] sat on the [MASK]'` $\longrightarrow$ `'cat'`, `'mat'`
- **Loss:** cross-entropy loss

Training tasks: pre-training

(2) Mask-Language Modeling (MLM)

- Pour entraîner les LLMs *non génératifs* de type *encoder-only* , tels que BERT, RoBERTa, etc.
- **Objectif:** étant donnée une suite de tokens, on masque des tokens au hasard (15%) et on demande de prédire ces tokens en utilisant le *contexte bi-directionnel* (passé et futur).
- **Exemple:** `'The [MASK] sat on the [MASK]'` $\longrightarrow$ `'cat'`, `'mat'`
- **Loss:** cross-entropy loss

Training tasks: fine-tuning

(3) Fine-tuning supervisé (supervised fine-tuning, SFT)

- Pour les modèles de type ChatGPT.
- **Objectif:** enseigner au modèle à suivre des instructions humaines et à générer des réponses utiles et polies.

  Données: paires de *prompts* et *réponses humaines*.

Training tasks: fine-tuning

(3) Fine-tuning supervisé (supervised fine-tuning, SFT)

- Pour les modèles de type ChatGPT.
- **Objectif:** enseigner au modèle à suivre des instructions humaines et à générer des réponses utiles et polies.

  Données: paires de *prompts* et *réponses humaines*.

Training tasks: fine-tuning

(3) **Fine-tuning supervisé (supervised fine-tuning, SFT)**

- Pour les modèles de type ChatGPT.
- **Objectif:** enseigner au modèle à suivre des instructions humaines et à générer des réponses utiles et polies.

  Données: paires de *prompts* et *réponses humaines*.

Training tasks: fine-tuning

(4) Reinforcement Learning from Human Feedback (RLHF)

- Pour les modèles de type ChatGPT.
- **Objectif:** Pour un prompt donné:
  - encoder un prompt en une séquence de tokens
  - calculer la réponse (inférence)
  - comparer la réponse à une réponse produite par un humain (reward model)

Training tasks: fine-tuning

(4) Reinforcement Learning from Human Feedback (RLHF)

- Pour les modèles de type ChatGPT.
- **Objectif:** Pour un prompt donné:
  - on collect multiple de responses possibles du modèle;
  - on les affiche à un humain (ou IA);
  - on met à jour le modèle pour que les réponses préférées soient plus probables.

Training tasks: fine-tuning

(4) Reinforcement Learning from Human Feedback (RLHF)

- Pour les modèles de type ChatGPT.
- **Objectif:** Pour un prompt donné:
  - on collect multiple de responses possibles du modèle;
  - on les classifie de manière humaine;
  - on fine-tune le modèle par renforcement (RL with PPO/DPO) à prédire les réponses préférées.

Training tasks: fine-tuning

(4) Reinforcement Learning from Human Feedback (RLHF)

- Pour les modèles de type ChatGPT.
- **Objectif:** Pour un prompt donné:
  - on collect multiple de responses possibles du modèle;
  - on les classifie de manière humaine;
  - on fine-tune le modèle par renforcement (RL with PPO/DPO) à prédire les réponses préférées.

Training tasks: fine-tuning

(4) Reinforcement Learning from Human Feedback (RLHF)

- Pour les modèles de type ChatGPT.
- **Objectif:** Pour un prompt donné:
  - on collect multiple de responses possibles du modèle;
  - on les classifie de manière humaine;
  - on fine-tune le modèle par renforcement (RL with PPO/DPO) à prédire les réponses préférées.

Training tasks: fine-tuning

(4) Reinforcement Learning from Human Feedback (RLHF)

- Pour les modèles de type ChatGPT.
- **Objectif:** Pour un prompt donné:
  - on collect multiple de responses possibles du modèle;
  - on les classifie de manière humaine;
  - on fine-tune le modèle par renforcement (RL with PPO/DPO) à prédire les réponses préférées.

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.