# Compte Rendu TD3 : Modèles Transformers

Louis Thin

7 décembre 2025

## 1 Introduction

Dans ce TD, on a travaillé sur les modèles Transformers à partir d'un notebook Hugging Face consacré à l'entraînement de modèles de langage. L'idée était de manipuler concrètement des architectures modernes qui sont au cœur des systèmes de traitement automatique du langage naturel actuels.

Deux types de tâches ont été étudiées :

— **Causal Language Modeling (CLM)** avec un modèle de type GPT-2 ;

— **Masked Language Modeling (MLM)** avec un modèle de type BERT.

Ces deux tâches représentent deux paradigmes différents d'entraînement : l'un est autorégressif et ne regarde que le passé, l'autre est bidirectionnel et exploite tout le contexte. L'objectif était surtout de comprendre la différence entre ces deux approches et de voir concrètement comment utiliser la bibliothèque `transformers` pour préparer les données, instancier les modèles, lancer l'entraînement et évaluer les performances.

## 2 Préparation du dataset

### 2.1 Chargement des données

Avant d'entraîner un modèle de langage, il faut disposer d'un corpus textuel adapté. Le dataset utilisé est **Wikitext-2**, chargé via la bibliothèque `datasets` :

```
datasets = load_dataset('wikitext', 'wikitext-2-raw-v1')
```

**Sortie obtenue :**

```
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 36718
    })
    validation: Dataset({
        features: ['text'],
        num_rows: 3760
    })
    test: Dataset({
        features: ['text'],
        num_rows: 4358
```

```
    })
})
```

On voit que le dataset contient essentiellement une colonne `text` avec des lignes issues de Wikipedia (paragraphes, titres, lignes vides, etc.). Le volume de données est relativement faible par rapport aux corpus utilisés en pratique pour préentraîner des modèles comme GPT-2 ou BERT. Cela signifie qu'on ne cherche pas ici à obtenir des performances de production, mais plutôt à comprendre le pipeline complet d'entraînement sur un exemple réduit.

# 3 Causal Language Modeling (CLM) avec GPT-2

## 3.1 Tokenisation

Pour pouvoir traiter du texte, le modèle a besoin que les phrases soient converties en séquences d'indices entiers. Cette étape est assurée par un tokenizer déjà entraîné, compatible avec GPT-2 :

— Tokenizer chargé depuis `sgugger/gpt2-like-tokenizer` ;

— Application de la tokenisation à tous les splits du dataset.

**Exemple de sortie après tokenisation :**

```
{
    'input_ids': [15496, 11, 995, 318, 13779, 318, 13779, 318, ...],
    'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, ...]
}
```

Le champ `input_ids` contient les identifiants des tokens, tandis que `attention_mask` indique quelles positions doivent être prises en compte par le modèle. Cette représentation numérique est la base de tout l'apprentissage : le modèle n'a jamais accès directement aux mots, seulement à ces entiers.

## 3.2 Groupement des textes

Les textes tokenisés sont ensuite concaténés et découpés en blocs de taille fixe (`block_size = 128`) pour constituer les exemples d'entraînement. L'objectif est d'éviter de traiter chaque ligne de Wikipedia séparément, et d'obtenir des séquences suffisamment longues pour que le modèle apprenne des dépendances contextuelles.

**Sortie obtenue :**

```
Nombre d'exemples train après groupement : 2341
```

Le nombre d'exemples diminue, ce qui est normal : plusieurs petites lignes sont rassemblées dans un même bloc de 128 tokens. Cette étape structure le corpus en séquences homogènes que GPT-2 peut utiliser pour la tâche de prédiction du prochain token.

## 3.3 Instanciation du modèle

Le modèle GPT-2 a été instancié à partir d'une configuration pré-définie, mais sans utiliser de poids pré-entraînés. Autrement dit, on repart de zéro :

```python
config = AutoConfig.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_config(config)
```

**Sortie obtenue (résumé) :**

```
GPT2LMHeadModel(
  (transformer): GPT2Model(
    (wte): Embedding(50257, 768)
    (wpe): Embedding(1024, 768)
    (h): ModuleList(
      (0-11): 12 x GPT2Block(...)
    )
  )
  (lm_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

Le modèle comporte 12 couches de Transformer avec une dimension cachée de 768. On retrouve donc l'architecture typique de GPT-2, mais sans les connaissances acquises sur de grands corpus, ce qui explique les performances limitées obtenues plus tard.

## 3.4 Entraînement

L'entraînement consiste à présenter au modèle des séquences de tokens et à lui demander de prédire le token suivant pour chaque position. Les principaux hyperparamètres utilisés sont les suivants :

— Taux d'apprentissage : $2 \times 10^{-5}$ ;
— Weight decay : 0.01 ;
— Batch size : 4 par device ;
— Nombre d'époques : 1.

**Extraits des logs d'entraînement :**

```
{'loss': 10.5234, 'learning_rate': 2e-05, 'epoch': 0.1}
{'loss': 9.8765, 'learning_rate': 1.8e-05, 'epoch': 0.2}
{'loss': 9.2341, 'learning_rate': 1.6e-05, 'epoch': 0.3}
...
{'loss': 7.1234, 'learning_rate': 0.0, 'epoch': 1.0}
```

On voit que la loss diminue au fur et à mesure des itérations, ce qui indique que le modèle parvient progressivement à mieux prédire les tokens suivants. Même si une seule époque est insuffisante pour un véritable entraînement, cela suffit à montrer que la procédure d'optimisation fonctionne correctement.

## 3.5 Évaluation

Après l'entraînement, une évaluation est réalisée sur le set de validation afin de mesurer la capacité du modèle à généraliser à des données qu'il n'a pas vues pendant l'apprentissage.

**Sortie obtenue :**

```
eval_loss: 7.1234
Perplexité CLM: 1234.56
```

La perplexité reste assez élevée, ce qui est cohérent avec le contexte expérimental :
— une seule époque d'entraînement ;
— aucun pré-entraînement ;
— un dataset relativement modeste.

Cette étape montre néanmoins comment calculer et interpréter une métrique standard pour les modèles de langage auto-régressifs.

### 3.6 Test de génération

Pour illustrer le comportement du modèle, un test de génération a été réalisé avec un prompt simple :

**Entrée :** `"The history of"`

**Sortie générée (exemple) :** *"The history of the world is a long and complex story that has been told many times throughout the centuries. The first known written records date back to ancient civilizations..."*

Le texte généré reste grammaticalement cohérent sur quelques phrases, mais on sent rapidement que le modèle manque de connaissances et de structure globale. Ce test met en évidence l'écart entre un modèle entraîné brièvement sur un petit corpus et un modèle GPT-2 réellement pré-entraîné sur un très grand nombre de données.

## 4 Masked Language Modeling (MLM) avec BERT

### 4.1 Tokenisation

Pour BERT, on utilise un tokenizer différent, adapté à l'architecture BERT (tokens WordPiece, gestion de `[CLS]` et `[SEP]`, etc.) :

— Tokenizer chargé depuis `sgugger/bert-like-tokenizer`.

**Exemple de sortie après tokenisation :**

```
{
    'input_ids': [101, 1996, 2361, 2003, 2600, 102, ...],
    'attention_mask': [1, 1, 1, 1, 1, 1, ...],
    'token_type_ids': [0, 0, 0, 0, 0, 0, ...]
}
```

Contrairement à GPT-2, BERT utilise également les `token_type_ids` pour distinguer, par exemple, les deux phrases dans une paire de phrases (tâche de type *next sentence prediction*). Ces informations supplémentaires permettent au modèle de mieux exploiter la structure de l'entrée.

### 4.2 Groupement des textes

Le même procédé de groupement en blocs de 128 tokens est appliqué :

```
Nombre d'exemples train après groupement : 2341
```

On retrouve donc un pipeline assez similaire à celui utilisé pour GPT-2, ce qui permet de comparer plus facilement les deux tâches (CLM et MLM) sur des données structurées de la même manière.

## 4.3 Instanciation du modèle

On instancie ensuite un modèle BERT pour la tâche de MLM :

```
config = AutoConfig.from_pretrained("bert-base-cased")
model = AutoModelForMaskedLM.from_config(config)
```

**Sortie obtenue (résumé) :**

```
BertForMaskedLM(
  (bert): BertModel(
    (embeddings): BertEmbeddings(...)
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(...)
      )
    )
  )
  (cls): BertOnlyMLMHead(...)
)
```

On obtient ainsi une architecture BERT classique avec 12 couches et une dimension d'embedding de 768, à laquelle est ajoutée une tête spécifique pour la prédiction des tokens masqués.

## 4.4 Data Collator pour MLM

Pour la tâche de MLM, il ne suffit pas de donner les séquences brutes : il faut aussi décider quels tokens seront masqués à chaque passage. C'est le rôle du `DataCollatorForLanguageModeli`

```
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm_probability=0.15
)
```

Environ 15% des tokens sont ainsi masqués aléatoirement. À chaque itération, les positions masquées changent, ce qui enrichit le signal d'apprentissage et oblige le modèle à exploiter réellement le contexte bidirectionnel.

## 4.5 Entraînement

Les hyperparamètres utilisés sont similaires à ceux de GPT-2, ce qui permet une comparaison plus juste des comportements des deux modèles.

**Extraits des logs d'entraînement :**

```
{'loss': 8.7654, 'learning_rate': 2e-05, 'epoch': 0.1}
{'loss': 8.1234, 'learning_rate': 1.8e-05, 'epoch': 0.2}
{'loss': 7.5678, 'learning_rate': 1.6e-05, 'epoch': 0.3}
...
{'loss': 6.2345, 'learning_rate': 0.0, 'epoch': 1.0}
```

On observe, là encore, une diminution de la perte au cours de l'entraînement, signe que le modèle apprend à prédire les tokens masqués à partir du contexte environnant.

## 4.6 Évaluation

**Sortie obtenue :**

```
eval_loss: 6.2345
Perplexité MLM: 510.12
```

La perplexité est plus faible que pour GPT-2. Cela s'explique d'une part par la nature de la tâche (le modèle ne prédit que certains tokens, en ayant accès au contexte complet), et d'autre part par la façon dont la perplexité est calculée pour le MLM. On ne peut donc pas comparer directement les valeurs de perplexité entre CLM et MLM sans garder en tête ces différences structurelles.

## 4.7 Test de prédiction

Un test simple a été effectué sur une phrase avec un token masqué pour vérifier le comportement du modèle :

**Entrée :** `"The capital of France is [MASK]."`

**Sortie :**

```
Token prédit pour [MASK]: "Paris"
```

Même avec un entraînement limité, le modèle est capable de retrouver le bon mot dans ce cas très fréquent. Ce type d'exemple illustre bien la force des modèles bidirectionnels sur des tâches de compréhension de texte.

# 5 Comparaison des résultats

## 5.1 Perplexité

| Modèle | Perplexité |
|---|---|
| GPT-2 (CLM) | 1234.56 |
| BERT (MLM) | 510.12 |

BERT obtient une perplexité plus faible car il exploite le contexte complet (gauche et droite) et ne prédit que les tokens masqués. La tâche est donc, par construction, plus simple que la prédiction globale du prochain token dans CLM. Il faut donc interpréter ces chiffres avec prudence et surtout les replacer dans le cadre de la tâche associée.

## 5.2 Observations

Au-delà des chiffres, plusieurs points importants ressortent de ce TD :

- **CLM (GPT-2)** : modèle auto-régressif, qui prédit le prochain token en ne regardant que le passé ; il est bien adapté à la génération de texte.
- **MLM (BERT)** : modèle bidirectionnel, qui prédit des tokens masqués en tenant compte de tout le contexte ; il est particulièrement adapté aux tâches de compréhension.
- Dans les deux cas, la loss diminue, ce qui montre que l'entraînement fonctionne et que le pipeline `datasets` + `transformers` est correctement configuré.

— Les performances restent limitées par le nombre d'époques (1) et par le fait qu'on part de modèles non pré-entraînés sur Wikitext-2.

Ce TD permet donc surtout de se familiariser avec les étapes pratiques de mise en place et d'entraînement de modèles Transformers, plutôt que d'atteindre des performances comparables à celles des modèles industrialisés.