

# PROJECT

Automatic Generation and Creation of Texts (Q&A)

JANUARY 2025

Prepared For

- Mme LAZIB

Prepared By

- Anis BENINI
- Mohand OURZIK
- Amazigh OULTACHE



# Sommaire

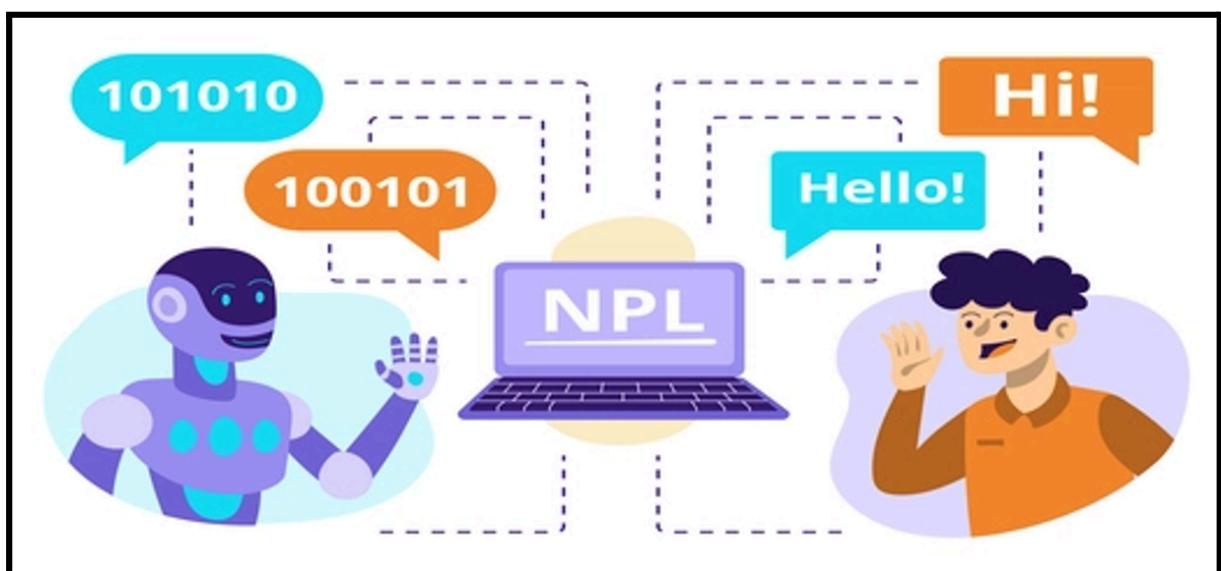
<b>I. Introduction.....</b>	<b>4</b>
<b>II. État de l'art.....</b>	<b>5</b>
2.1. Génération Automatique de Textes en NLP.....	5
2.2 Présentation des Modèles de Langage Utilisés.....	5
Motivation du Projet.....	5
GPT2 Large.....	6
LLaMa 3.2 1B.....	7
2.3. Le Fine-tuning dans le Traitement du Langage Naturel (NLP).....	8
Processus et Méthodologie.....	8
Apport du Fine-tuning pour la Tâche de Questions-Réponses.....	8
<b>III. Méthodologie.....</b>	<b>8</b>
1. Préparation et Traitement des Données.....	8
A) Préparation des données.....	8
- Collecte et Constitution du Corpus.....	8
- Origine et Structure du Dataset.....	8
B) Justification du Choix du Dataset.....	9
C) Prétraitement.....	9
1. Chargement et Nettoyage des Données.....	9
2. Formatage des Prompts.....	10
3. Analyse et Filtrage des Données.....	12
4. Conversion en Format Exploitable.....	13
2. Implémentation et Adaptation des Modèles.....	14
2.1. Processus de Fine-tuning.....	14
2.1.1. Paramétrage de l'Entraînement.....	14
→ Hyperparamètres pour LLaMA 3.2 1B.....	14
→ Hyperparamètres pour GPT-2 Large.....	16
→ Stratégie d'Optimisation.....	17
→ QLoRA pour une Meilleure Gestion des Ressources.....	18
2.1.2. Préparation et Configuration du Tokenizer.....	18
2.1.3. Quantification du Modèle.....	20
2.1.4. Chargement et Configuration des Modèles.....	20
2.1.5. Configuration et Lancement du Fine-Tuning.....	21
3. Suivi de l'Entraînement et Ajustement du Taux d'Apprentissage.....	23
→ Suivi de l'entraînement.....	23
→ Planification du Taux d'Apprentissage.....	25
<b>IV. Évaluation des Performances.....</b>	<b>27</b>
1. Métriques d'Évaluation.....	27
2. Exemples de Générations du Modèle.....	28
<b>V. Conclusion.....</b>	<b>30</b>

## I. Introduction :

Le traitement automatique du langage naturel (**NLP**, Natural Language Processing) est un domaine crucial de l'intelligence artificielle qui vise à permettre aux machines de comprendre, générer et interagir avec le langage humain. Les applications du **NLP** sont omniprésentes, allant de la traduction automatique à l'analyse des sentiments, en passant par le résumé automatique et les systèmes de questions-réponses (**QA**). Ces systèmes, qui permettent de fournir des réponses précises à des questions posées par des utilisateurs, jouent un rôle essentiel dans des contextes variés tels que les assistants virtuels, les chatbots et les applications éducatives.

L'émergence de modèles pré-entraînés comme **GPT2** et **Llama** a considérablement modifié le paysage du **NLP**. Ces modèles, dotés d'une architecture transformer, se distinguent par leur capacité à générer des textes de haute qualité en s'appuyant sur des quantités massives de données. Cependant, leur performance brute ne suffit pas toujours pour des applications précises. C'est pourquoi le fine-tuning, qui consiste à ajuster un modèle pré-entraîné sur un corpus spécifique et une tâche ciblée, s'avère essentiel. Dans le cadre de ce projet, nous nous intéressons particulièrement à l'adaptation de deux modèles **GPT2 large** et **Llama 3.2 1B** pour la réalisation de tâches de questions-réponses. Cette approche vise à combiner la puissance générative des modèles avec la précision nécessaire pour répondre efficacement à des interrogations posées en langage naturel.

Ce projet cherche à adapter et comparer deux modèles de langage **GPT2 large** et **Llama 3.2 1B** pour améliorer leurs performances en questions-réponses. Il s'agit de relever le défi d'affiner des modèles généraux afin de garantir des réponses précises et pertinentes, tout en identifiant les forces et faiblesses de chaque approche dans un contexte d'utilisation ciblée.



## **II. État de l'art**

### **2.1. Génération Automatique de Textes en NLP**

Historiquement, la génération de texte a évolué depuis les méthodes basées sur des règles et des templates, puis les modèles statistiques, jusqu'aux modèles basés sur l'architecture **Transformer**, comme GPT et ses variantes. Ces derniers, entraînés sur d'immenses corpus, offrent une compréhension contextuelle avancée et permettent de générer des contenus cohérents et fluides, ouvrant ainsi la voie à diverses applications, notamment la rédaction automatique d'articles et la création de dialogues interactifs.

### **2.2 Présentation des Modèles de Langage Utilisés**

#### **→ Motivation du Projet**

- Pourquoi comparer **LlaMa 3.2 1B** vs **GPT2 Large** pour un dataset **QA** ???



Dans le cadre de ce projet, l'objectif est de Fine-Tuner et d'évaluer deux modèles pour des tâches de Question-Réponses. Le choix de comparer **LlaMa 3.2 1B** et **GPT2 Large** repose sur plusieurs raisons :

#### **→ Diversité architecturale et évolutive :**

**GPT2 Large**, avec ses **36 couches**, une taille cachée de **1280** et **20** têtes d'attention pour un total de **774 millions** de paramètres, représente une génération plus ancienne des modèles Transformer. À l'inverse, **LlaMa 3.2 1B**, conçu pour environ **1 milliard** de paramètres, illustre les avancées récentes en termes de gestion de contexte (par exemple, la capacité à traiter un contexte bien plus étendu) et d'optimisation des ressources.

La comparaison entre ces deux approches permet d'évaluer comment l'évolution architecturale impacte la performance dans la compréhension et la génération de réponses précises sur un dataset QA.

#### **→ Pertinence pour la tâche QA :**

La capacité à comprendre et répondre correctement à des questions repose sur une bonne gestion du contexte et une compréhension fine des nuances linguistiques. En comparant ces deux modèles, on peut observer les différences en termes de qualité des réponses générées, leur cohérence et leur pertinence par rapport aux questions posées.

❖ **GPT2 Large :**

**a) Architecture et Caractéristiques**

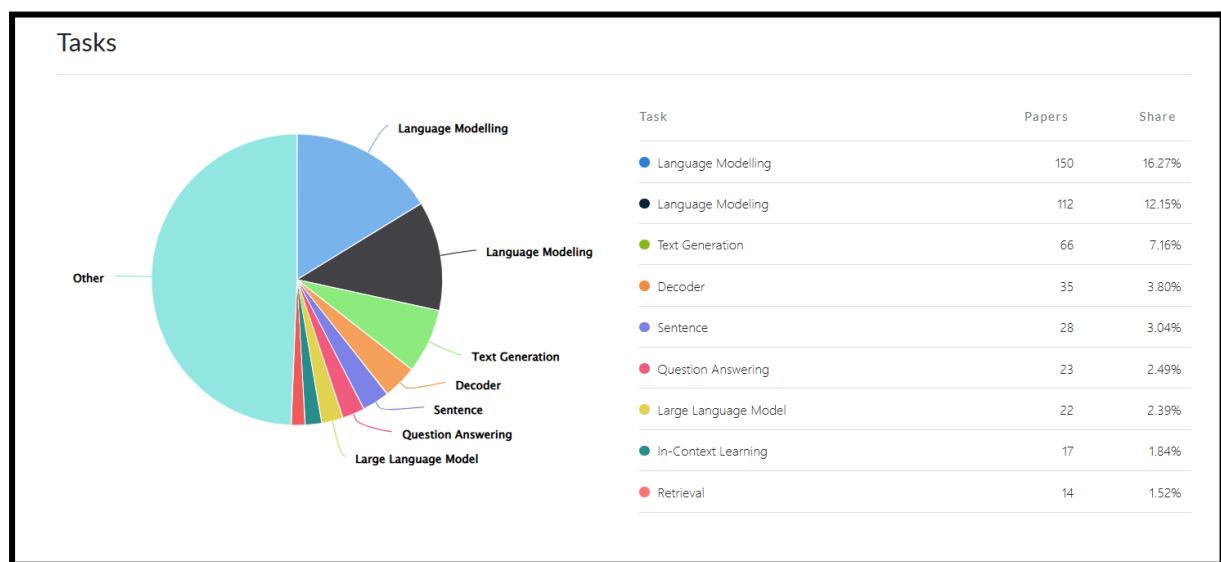
GPT2 Large représente une version avancée du modèle GPT2, fondée sur l'architecture Transformer. Ce modèle intègre de multiples couches et compte des millions de paramètres, ce qui lui confère une remarquable capacité à générer des textes à la fois cohérents et riches en contexte.

**b) Performances en Génération de Texte**

En s'appuyant sur un entraînement réalisé sur d'énormes volumes de données textuelles, GPT2 Large se distingue par la fluidité et la diversité de ses productions. Sa capacité à adapter le style et le contenu en fonction du contexte le rend particulièrement adapté aux tâches nécessitant créativité, finesse et adaptabilité.

**c) Utilisation en Fine-tuning**

L'architecture pré-entraînée de GPT2 Large facilite sa spécialisation pour des tâches précises. Grâce à la technique du fine-tuning, le modèle peut être ajusté sur des jeux de données ciblés, optimisant ainsi ses performances pour des applications spécifiques telles que la génération de réponses en contexte de questions-réponses.



- Some Details about this model :

→ 36-layer, 1280-hidden, 20-heads, 774M parameters.

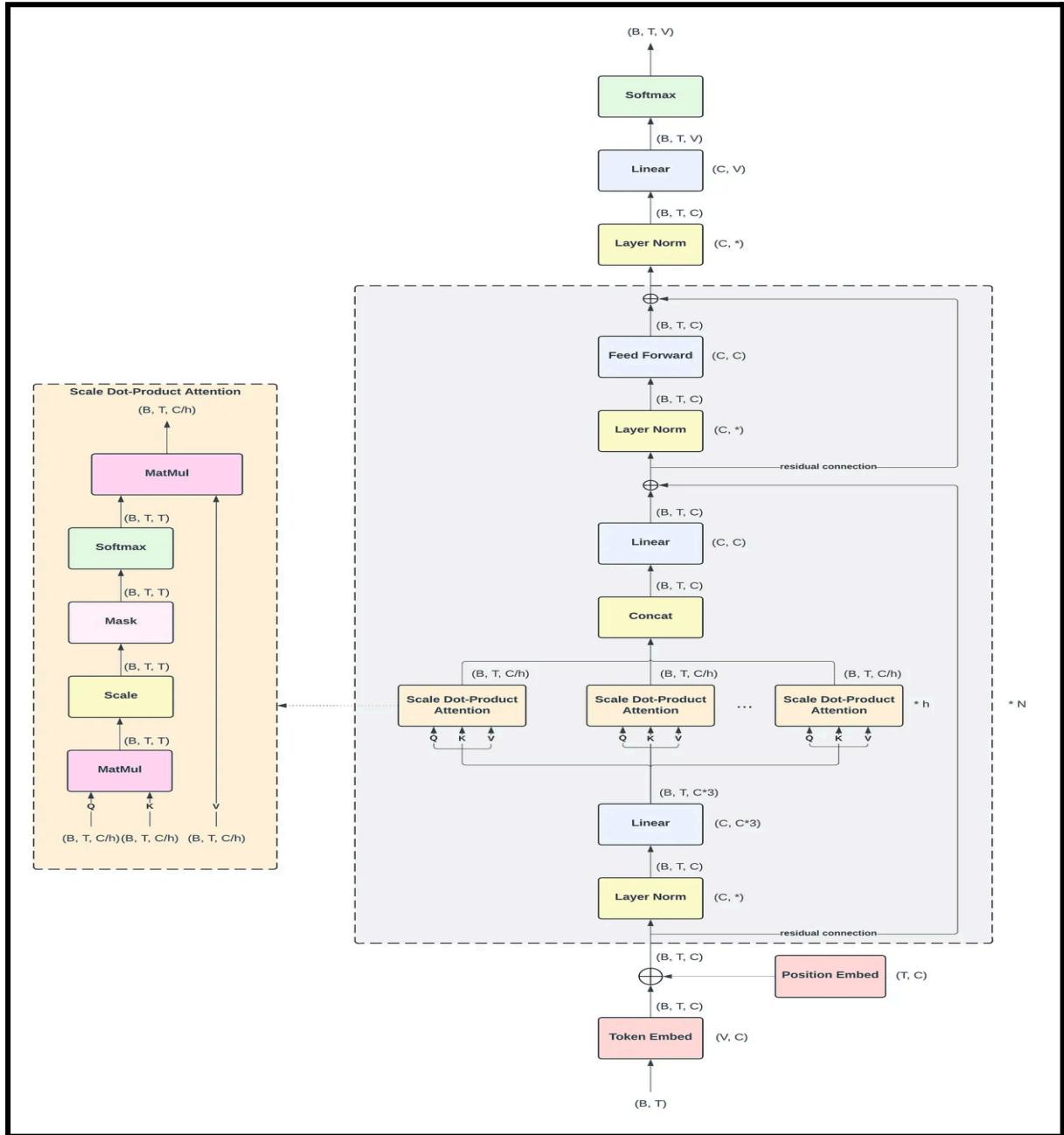


Figure1: Gpt2 Architecture diagram

❖ **Llama 3.2 1B :**

a) **Architecture et Caractéristiques**

Llama 3.2 1B repose sur l'architecture **Transformer** et adopte un modèle entièrement basé sur un décodeur, similaire à GPT. Avec un milliard de paramètres, il est conçu pour offrir un compromis idéal entre puissance de calcul et efficacité, permettant une meilleure gestion des ressources tout en garantissant des performances solides en compréhension et génération de texte.

b) **Performances en Génération de Texte**

Comme **GPT2 Large**, Llama 3.2 1B utilise un modèle de type décodeur uniquement, optimisé pour la génération de texte auto-régressive. Ce mécanisme permet au modèle de prédire chaque mot successif en s'appuyant sur le contexte précédent, assurant ainsi une production fluide et cohérente des réponses. Grâce à cette architecture, Llama 3.2 1B excelle dans les tâches nécessitant une génération de texte dynamique et contextuelle, ce qui en fait un choix pertinent pour un système de questions-réponses interactif. Sa légèreté lui confère également un avantage en rapidité d'exécution, le rendant particulièrement adapté aux applications nécessitant des réponses quasi-instantanées.

c) **Utilisation en Fine-tuning**

Comme GPT2 Large, Llama 3.2 1B peut être affiné via un fine-tuning ciblé pour améliorer ses performances sur des tâches spécifiques. Son architecture plus compacte permet un entraînement plus rapide et moins coûteux en ressources, sans compromettre la qualité des réponses générées.

→ Pour **Llama 3.2 1B**, bien que les détails architecturaux exacts puissent varier selon l'implémentation, une configuration typique pour un modèle d'environ 1 milliard de paramètres (**1.24B** params) serait la suivante :

- **Nombre de couches** : environ 24
- **Taille cachée** : autour de 1024
- **Nombre de têtes d'attention** : environ 16.

	Training Data	Params	Input Modalities	Output Modalities	Context Length	GQA	Shared Embedding	Token Count	Knowledge CutOff
<b>LLama 3.2 1B</b>	A new mix of publicly available online data.	1,24 B	MultiLingual Text	MultiLingual Text & Code	128K	Yes	Yes	Up to 9T tokens	December 2023
<b>GPT 2 Large</b>	WebText dataset (~40GB of internet text )	774M	Monolingual (English) text	Monolingual (English) text	1024 tokens	No	No	~10B tokens	October 2019

### **2.3. Le Fine-tuning dans le Traitement du Langage Naturel (NLP)**

Le fine-tuning s'est imposé comme une étape incontournable dans l'exploitation des modèles de langage pré-entraînés, permettant leur adaptation à des tâches spécifiques. Il s'agit d'affiner un modèle ayant déjà acquis une connaissance générale du langage en le réentraînement sur un corpus spécialisé correspondant à une application ciblée.

#### **➤ Processus et Méthodologie :**

Le fine-tuning repose sur un réajustement des poids du modèle, réalisé à travers un nouvel entraînement sur un jeu de données annoté propre au domaine étudié. Ce processus vise à optimiser ses capacités de génération et de compréhension du texte. L'ajustement des HyperParamètres clés, tels que le taux d'apprentissage, le nombre d'époques et la taille des batchs permet d'optimiser les performances et d'assurer une convergence efficace du modèle.

#### **➤ Apport du Fine-tuning pour la Tâche de Questions-Réponses :**

Appliquer le fine-tuning à un modèle dans un cadre de questions-réponses améliore significativement la précision, la cohérence et la pertinence des réponses générées. En s'entraînant sur des exemples spécifiques, le modèle apprend à mieux structurer ses réponses, à capter les nuances du langage et à mieux gérer les ambiguïtés contextuelles. De plus, ce processus permet de corriger certains biais présents dans le modèle pré-entraîné, garantissant ainsi une production plus fiable et adaptée aux besoins de l'utilisateur.

## **III. Méthodologie**

### **1. Préparation et Traitement des Données**

#### **A) Préparation des données :**

##### **- Collecte et Constitution du Corpus :**

Dans le cadre de notre projet, nous avons sélectionné le **Glaive Python Code QA Dataset**, un corpus de 140 000 paires de questions-réponses orientées vers la programmation en Python. Ce dataset est conçu pour soutenir le développement d'assistants intelligents capables de répondre avec précision aux requêtes des utilisateurs sur des problématiques de code.

##### **- Origine et Structure du Dataset :**

Le **Glaive Code Assistant dataset**, publié sur Hugging Face, est un ensemble de données structuré au format questions-réponses. Il couvre un large éventail de thématiques en programmation Python, allant des concepts fondamentaux (types de données, structures de contrôle) aux notions avancées (programmation orientée objet, manipulation des exceptions, optimisation du code).

Les caractéristiques principales du dataset sont les suivantes :

- **140 000** entrées comprenant des questions réelles posées par des utilisateurs sur des problématiques de code.

- **60 %** du dataset dédié à Python, le reste étant composé d'autres langages et concepts de programmation.

- Réponses claires et concises, souvent accompagnées de snippets de code, ce qui le rend particulièrement adapté à l'entraînement d'un modèle de questions-réponses spécialisé en développement informatique.

## B) Justification du Choix du Dataset

Ce dataset a été retenu pour plusieurs raisons :

1. **Qualité et diversité des données** : Il regroupe des problématiques variées, permettant d'entraîner un modèle robuste et polyvalent.
2. **Format optimisé pour le fine-tuning** : La structuration en paires questions-réponses facilite l'adaptation des modèles pré-entraînés, notamment **GPT2 Large** et **Llama 3.2 1B**, à la tâche spécifique de génération de réponses en programmation.
3. **Données réalistes** : Les questions étant issues de problématiques concrètes rencontrées par des développeurs, le modèle entraîné sera mieux préparé à traiter des requêtes authentiques.
4. **Licence libre (CC0 1.0)** : L'absence de restrictions liées aux droits d'auteur permet une exploitation sans contrainte pour la recherche et le développement.

Ce corpus constitue ainsi une base solide pour le fine-tuning de modèles de langage, leur permettant d'apprendre à formuler des réponses précises et pertinentes à des requêtes techniques, dans le cadre d'un système intelligent de questions-réponses dédié au développement en Python.

## C) Prétraitement :

Le prétraitement des données est une étape cruciale permettant de garantir la qualité et la pertinence des exemples utilisés pour le fine-tuning du modèle. Cette phase a impliqué plusieurs transformations sur le dataset initial afin de le rendre exploitable par notre modèle de questions-réponses.

### 1. Chargement et Nettoyage des Données :

Les données ont été chargées à partir d'un fichier CSV contenant deux colonnes principales : **QUESTION & ANSWER**. Une préparation initiale a été réalisée pour :

- Suppression des valeurs manquantes : les lignes contenant des valeurs nulles ont été éliminées afin d'éviter les entrées incomplètes.
- Conversion en dictionnaire : transformation du DataFrame en dictionnaire pour faciliter le formatage des données.

→ Voici le code utilisé :

```
dataset_path = path_dir + '/train.csv'
df = pd.read_csv(dataset_path, encoding= 'utf-8')
df = df.dropna(axis=0)
data = {
    'question': list(df['question']),
    'answer': list(df['answer']),
}

df = pd.DataFrame(data= data, columns=[ 'question', 'answer'])
# convert the DataFrame to dict()
examples = df.to_dict()
```

→ Voici le résultat obtenu :

	question	answer
0	How can I output bold text in Bash? I have a B...	Yes, you can format the output text in Bash to...
1	How can I install Python 3 on an AWS EC2 insta...	To install Python 3 on an AWS EC2 instance, yo...
2	How can I format the elapsed time from seconds...	You can achieve the desired time format using ...
3	I am trying to create a matrix of random numbe...	Your current implementation is actually quite ...
4	I am learning Python and have noticed extensiv...	The use of 'self' in Python is quite different...
...	...	...
136103	Can you provide a Ruby program that reads an a...	Sure! Here's a Ruby program that accomplishes ...
136104	What is the code to develop a program that tak...	You can use the following code:\n\n```python\n...
136105	How can I update my code to verify if the vari...	You can use the `isinstance()` function in Pyt...
136106	Can you provide an application code that autom...	Sure! Here's a code snippet that demonstrates ...
136107	I am trying to write a function in Java that c...	Certainly! Here's a Java function that checks ...
136108 rows x 2 columns		

## 2. Formatage des Prompts :

Le dataset a ensuite été reformulé pour respecter un format adapté au Fine-Tuning du modèle de langage.

Chaque exemple a été structuré comme suit :

```
### Question: {question}
### Answer: {answer}
```

→ Voici le code utilisé :

```
def formatting_prompts_func(example):
    output_texts = []
    for i in range(len(example['question'])):
        text = f"### Question: {example['question'][i]}\n### Answer: {example['answer'][i]}"
        output_texts.append(text)
    return output_texts

finetuning_dataset_question_answer = formatting_prompts_func(examples)
```

Ce formatage explicite permet au modèle d'identifier clairement la structure des paires questions-réponses et d'améliorer la qualité des générations, ensuite on a modifier Le DataFrame pour qu'il soit constitué d'une seule colonne, appelée "text", qui contient les textes formatés (contenant à la fois la question et la réponse) avec le code suivant :

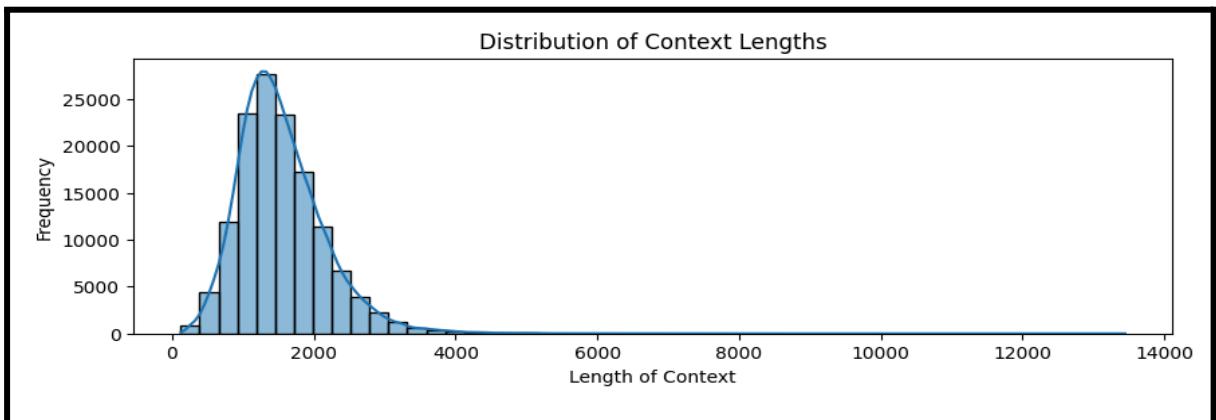
```
filtered_dataset_df = pd.DataFrame(finetuning_dataset_question_answer, columns=['text'])
filtered_dataset_df
```

→ Voici le résultat obtenu :

	text
0	### Question: How can I output bold text in Ba...
1	### Question: How can I install Python 3 on an...
2	### Question: How can I format the elapsed tim...
3	### Question: I am trying to create a matrix o...
4	### Question: I am learning Python and have no...
...	...
136103	### Question: Can you provide a Ruby program t...
136104	### Question: What is the code to develop a pr...
136105	### Question: How can I update my code to veri...
136106	### Question: Can you provide an application c...
136107	### Question: I am trying to write a function ...
136108 rows × 1 columns	

### **3. Analyse et Filtrage des Données :**

Une analyse statistique a été réalisée pour observer la distribution des longueurs de texte, notamment à l'aide d'un **histogramme des longueurs de contexte**. La figure ci-dessous nous montre le résultat obtenu :



Cette analyse a permis de fixer un **seuil maximal de 1024 tokens** afin d'éliminer les exemples trop longs, susceptibles de dépasser la capacité du modèle et d'entraîner des inefficacités dans l'apprentissage. On a utilisé le code suivant :

```
context_length = 1024
filtered_dataset = []
for text in finetuning_dataset_question_answer:
    if len(text) < context_length:
        filtered_dataset.append(text)

len(filtered_dataset)
```

Le filtrage final a conduit à un dataset optimisé, conservant uniquement les entrées respectant cette contrainte de longueur. Voici le Dataset obtenu :

	text
0	### Question: How can I make my Python program...
1	### Question: How can I parse a comma-separate...
2	### Question: In Python, is there an easy meth...
3	### Question: In Python, I know there is a lot...
4	### Question: How can I move the y-axis ticks ...
...	...
24894	### Question: How can I write a SQL query to c...
24895	### Question: I want to modify my code so that...
24896	### Question: How can I sort an array in ascen...
24897	### Question: Can you provide a Ruby program t...
24898	### Question: How can I update my code to veri...
24899 rows × 1 columns	

#### **4. Conversion en Format Exploitable :**

Après nettoyage et filtrage, le dataset a été transformé en un format compatible avec l'entraînement du modèle :

- Conversion en dataset **Hugging Face** pour une gestion optimisée des données avec le code suivant :



```
finetuning_dataset_loaded = datasets.Dataset.from_pandas(filtered_dataset_df)
finetuning_dataset_loaded
```

- **Division en ensembles d'entraînement et de test** (**80 %** pour l'entraînement, **20 %** pour le test), avec une répartition aléatoire assurant une diversité dans les exemples présentés au modèle. Voici le code utilisé :



```
dataset = finetuning_dataset_loaded.train_test_split(test_size=0.20, shuffle=True,
seed=42)
train_dataset = dataset['train']
test_dataset = dataset['test']
train_dataset, test_dataset
```

→ voici le résultat qu'on a obtenu :

```
(Dataset({
    features: ['text'],
    num_rows: 19919
}),
Dataset({
    features: ['text'],
    num_rows: 4980
}))
```

- ★ Grâce à ce pipeline de prétraitement, nous avons obtenu un corpus propre, bien structuré et adapté au fine-tuning, garantissant ainsi une meilleure performance du modèle en phase d'apprentissage.

## **2. Implémentation et Adaptation des Modèles :**

### **2.1. Processus de Fine-tuning**

#### **2.1.1. Paramétrage de l'Entraînement :**

L'entraînement des modèles **GPT-2 Large** et **LLaMA 3.2 1B** a nécessité un ajustement précis des hyperparamètres afin d'optimiser les performances tout en tenant compte des contraintes matérielles.

#### → **Hyperparamètres pour LLaMA 3.2 1B :**

Les paramètres clés ont été définis comme suit :

- **Nombre d'époques : 2**

- Un nombre réduit d'époques permet de limiter le sur-apprentissage, en particulier sur des datasets spécialisés comme Glaive Python Code QA, où une trop grande exposition aux données d'entraînement pourrait nuire à la généralisation.
- Lors d'essais préliminaires, nous avons observé que la perte (loss) diminuait significativement en deux époques sans entraîner une saturation du modèle.

- **Taille du Batch : 4**

- La taille du Batch a été choisie en fonction des contraintes matérielles (GPU disponible) tout en maintenant un entraînement stable.
- Des tests avec des batchs plus grands ont conduit à une consommation excessive de mémoire, tandis que des batchs plus petits ralentissaient la convergence.

- **Gradient Accumulation Steps : 1**

- Cette valeur permet d'ajuster efficacement la taille effective du batch sans dépasser la mémoire disponible.
- Comme nous utilisons une taille de batch réduite, nous n'avons pas eu besoin d'accumuler plusieurs mises à jour de gradients avant d'effectuer une rétropropagation.

- **Taux d'apprentissage (learning rate) : 3e-4**

- Choisir un taux d'apprentissage (Learning Rate) optimal est crucial pour assurer une bonne convergence.
- Après plusieurs tests, nous avons constaté qu'un **learning rate** trop élevé (**>5e-4**) entraînait une instabilité et des oscillations, tandis qu'un **learning rate** trop faible (**<1e-4**) ralentissait considérablement l'apprentissage.

- **Type de scheduler : Cosine**
    - Le scheduler en **cosinus** est utilisé pour réduire progressivement le learning rate, ce qui permet une convergence plus douce et réduit les risques d'oscillation en fin d'entraînement.
    - Contrairement à un scheduler linéaire, la décroissance **cosinus** ajuste mieux la diminution du taux d'apprentissage, ce qui est particulièrement utile sur des modèles nécessitant un fine-tuning précis.
  - **WarmUp Ratio : 3%**
    - Une phase de **WarmUp** permet d'augmenter progressivement le learning rate au début de l'entraînement, évitant ainsi des mises à jour de poids trop brutales qui pourraient perturber l'optimisation.
    - Ce ratio de **3%** a été choisi après expérimentation : un **WarmUp** plus long ralentissait inutilement l'entraînement, tandis qu'un warmup plus court entraînait une instabilité initiale.
  - **Weight Decay : 0.001**
    - La régularisation via le weight decay prévient le sur-apprentissage en réduisant la magnitude des poids, ce qui améliore la capacité du modèle à généraliser.
    - Une valeur de **0.001** a été retenue après des tests où des valeurs plus élevées réduisaient la capacité du modèle à mémoriser les relations complexes des données, tandis que des valeurs plus faibles laissaient le modèle sur-apprendre.
- **Hyperparamètres pour GPT-2 Large :**
- **Nombre d'époques : 5**
    - Contrairement au fine-tuning de **LLaMA 3.2 1B**, nous avons choisi **5** époques pour **GPT-2** afin de permettre un apprentissage plus approfondi sur les structures de Questions-Réponses.
    - Des tests préliminaires ont montré qu'un entraînement plus court (< **3** époques) ne permettait pas au modèle d'acquérir suffisamment de connaissances, tandis qu'un entraînement trop long (> **5** époques) entraînait un sur-apprentissage.
  - **Taille du batch : 8**
    - Une taille de batch plus grande que celle utilisée pour **LLaMA 3.2 1B** a été adoptée, car **GPT-2** est un modèle plus léger et peut gérer un plus grand nombre d'exemples en parallèle sans surcharge mémoire.
    - Cette taille améliore la stabilité de l'entraînement en réduisant les variations du gradient.
  - **Gradient Accumulation Steps : 1**
    - Cette valeur est maintenue à **1** car la taille du batch est suffisante pour un bon apprentissage.

- L'accumulation de gradient n'était pas nécessaire pour éviter les dépassemens mémoire.
- **Taux d'apprentissage (learning rate) : 3e-4**
  - Le choix d'un learning rate de **3e-4** permet une convergence rapide tout en minimisant les risques d'oscillations.
  - Des essais avec un learning rate plus élevé (**>5e-4**) ont entraîné des instabilités, tandis qu'un taux plus faible (**<1e-4**) ralentissait excessivement l'apprentissage.
- **Type de scheduler : Cosine**
  - Comme pour **LLaMA 3.2 1B**, un scheduler cosinus a été sélectionné pour réduire progressivement le learning rate et éviter une convergence trop brutale qui pourrait nuire à la qualité des générations textuelles.
- **WarmUp Ratio : 3%**
  - Ce ratio permet une montée progressive du learning rate au début de l'entraînement pour éviter des ajustements trop abrupts des poids du modèle.
  - Un WarmUp plus court réduisait la stabilité initiale, tandis qu'un WarmUp plus long ralentissait l'optimisation.
- **Weight Decay : 0.001**
  - Comme pour **LLaMA 3.2 1B**, un weight decay de **0.001** est utilisé pour éviter le sur-apprentissage, tout en assurant une bonne généralisation.

#### → Stratégie d'Optimisation

Le choix de l'algorithme d'optimisation joue un rôle clé dans la convergence du modèle. Pour ce Fine-Tuning, nous avons opté pour **Paged AdamW 32-bit**.

**Paged AdamW 32-bit** est une version optimisée de l'optimiseur **AdamW**, qui utilise une gestion paginée de la mémoire pour réduire la consommation GPU tout en maintenant une précision en **32 bits**. Il est conçu pour entraîner efficacement de grands modèles de langage sur du hardware avec une mémoire limitée.

- Décomposition du terme :
  - **Paged** : Utilise une gestion paginée de la mémoire pour réduire l'empreinte mémoire sur le GPU, en stockant certaines parties des poids et gradients sur la mémoire CPU lorsque nécessaire.
  - **AdamW** : Une amélioration de l'optimiseur Adam, introduisant une meilleure régularisation via la pénalisation du poids (weight decay), ce qui améliore la généralisation du modèle.
  - **32-bit** : Indique que les calculs sont effectués en précision 32 bits (FP32), offrant un équilibre entre précision et efficacité mémoire.

, qui présente plusieurs avantages :

- Optimisation efficace de la mémoire, permettant l'entraînement sur des modèles volumineux.
- Gestion optimisée des mises à jour des poids pour améliorer la stabilité.
- Adapté aux contraintes de fine-tuning sur des architectures complexes comme GPT-2 et LLaMA.

#### → **QLoRA pour une Meilleure Gestion des Ressources :**

Le fine-tuning a été effectué avec **QLoRA (Quantized Low-Rank Adaptation)**, une méthode qui réduit la consommation mémoire tout en adaptant efficacement un modèle.

- **Paramètres clés :**

- **LoRA Rank (16)** : Définit la taille des matrices de bas-rang pour l'adaptation.
- **LoRA Alpha (32)** : Amplifie les mises à jour des poids pour compenser la réduction de dimension.
- **LoRA Dropout (20%)** : Prévient le sur-apprentissage en régularisant les activations.

- **Couches ciblées pour chaque modèle :**

- **LLaMA 3.2 1B** : `['q_proj', 'k_proj', 'v_proj', 'o_proj']`  
→ Ajustement des projections d'attention pour une meilleure spécialisation.
- **GPT-2 Large** : `['c_attn', 'c_fc']`  
→ Fine-tuning limité aux modules d'attention et de feedforward pour un équilibre entre adaptation et efficacité.

#### **2.1.2. Préparation et Configuration du Tokenizer :**

Nous avons chargé le tokenizer de **LLaMA 3.2 1B** pour préparer les données d'entrée du modèle. Les paramètres suivants ont été définis :

- **Longueur maximale : 2048 tokens** (correspondant à la capacité du modèle).
  - **Padding** : Utilisation du token **EOS** (`pad_token = eos_token`) pour compléter les séquences plus courtes.
  - **Tronquage** : Les séquences trop longues sont coupées à droite (`truncation_side = 'right'`).
  - **Alignement du padding** : Défini sur **droite** pour optimiser le traitement batché.
- Cela garantit une gestion efficace des entrées, essentielle pour un fine-tuning stable et performant.

→ Voici le code qu'on a utilisé :

```
# load the tokenizer
max_sequence_length = 2048 # == llama 3.2 1B max_sequence length

tokenizer = AutoTokenizer.from_pretrained(
    'meta-llama/Llama-3.2-1B',
    trust_remote_code = True
)
## pad the sequence if it is < max_sequence_length
tokenizer.pad_token = tokenizer.eos_token

tokenizer.padding_side = 'right'

tokenizer.model_max_length = max_sequence_length

tokenizer.truncation_side = 'right'
```

### 2.1.3. Quantification du Modèle

Nous avons activé la quantification en **4 bits** pour réduire l'utilisation mémoire et accélérer l'inférence. La configuration utilise **NF4** (Normalized Float 4), un format optimisé pour les modèles de langage, et active la double quantification pour améliorer la précision. Si la quantification en **4 bits** est désactivée, le modèle est chargé en **8 bits** tout en conservant un calcul en **float16**.

→ Le code ci-dessous illustre cela :

```
# The Quantization
quant_4_bit = True
if quant_4_bit:
    quant_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_compute_dtype=torch.float16,
        bnb_4bit_quant_type="nf4",
    )
else:
    quant_config = BitsAndBytesConfig(
        load_in_8bit=True,
        bnb_4bit_compute_dtype=torch.float16,
```

## 2.1.4. Chargement et Configuration des Modèles

Nous avons chargé les modèles **LLaMA 3.2 1B** et **GPT-2 Large** avec la quantification configurée pour optimiser l'utilisation mémoire. Le paramètre `device_map='auto'` permet une répartition automatique du modèle sur les ressources disponibles (CPU/GPU). De plus, le token de padding a été défini comme le token EOS du tokenizer pour assurer une gestion cohérente des séquences générées. Enfin, l'empreinte mémoire de chaque modèle est affichée pour évaluer leur consommation en fonction de la quantification appliquée. Voici le code qui nous permet cela :

```
## set the base model
base_model =
AutoModelForCausalLM.from_pretrained(
    path_dir + 'llama3.2-1B',
    # 'meta-llama/Llama-3.2-1B',
    quantization_config = quant_config,
    device_map = 'auto',
    # local_files_only = True,
)

base_model.generation_config.pad_token_id =
tokenizer.eos_token_id
print(f"{{(base_model.get_memory_footprint() / 1e6):.2f} MB")
```

**LLaMA 3.2-1B**

```
model = AutoModelForCausalLM.from_pretrained(
    'openai-community/gpt2-large',
    quantization_config = quant_config,
    device_map = 'auto',
    # local_files_only = True,
)

model.generation_config.pad_token_id =
tokenizer.eos_token_id
print(f"{{(model.get_memory_footprint() / 1e6):.2f} MB")
```

**GPT-2-Large**

## 2.1.5. Configuration et Lancement du Fine-Tuning :

Nous avons défini la configuration **LoRA** (Low-Rank Adaptation) pour optimiser l'adaptation du modèle tout en réduisant la consommation mémoire.

Les hyperparamètres clés incluent LoRA Rank, Alpha et Dropout, ainsi que les couches ciblées pour l'adaptation avec le code suivant :

```
lora_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_rank,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=target_modules,
)
```

Ensuite, nous avons configuré les paramètres d'entraînement via **SFTConfig**, en précisant le nombre d'époques, la taille des batchs, l'optimiseur (**Paged\_AdamW**), le scheduler d'apprentissage (**cosine**), ainsi que les stratégies de sauvegarde et de journalisation sur **TensorBoard** avec le code suivant :

```
train_config = SFTConfig(  
    output_dir = output_dir,  
    num_train_epochs=epochs,  
    per_device_train_batch_size=batch_size,  
    per_device_eval_batch_size=batch_size,  
    eval_strategy='no',  
    gradient_accumulation_steps=gradient_accumulation_steps,  
    optim = optimizer,  
    save_steps=20,  
    logging_steps=20,  
    logging_dir= path_dir + '/logs',  
    save_total_limit = 10,  
    learning_rate=learning_rate,  
    weight_decay=weight_decay,  
    fp16 = True,  
    bf16 = False, ## set to True if you're not using the A100 GPU  
    max_grad_norm=0.3,  
    max_steps=-1,  
    warmup_ratio=warmup_ratio,  
    group_by_length=True,  
    max_seq_length=max_sequence_length,  
    lr_scheduler_type=lr_scheduler_type,  
    dataset_text_field="text",  
    save_strategy='steps',  
    report_to='tensorboard',  
)
```

Nous avons instancié le **SFTTrainer**, qui orchestre le Fine-Tuning en intégrant le modèle de base, les données d'entraînement, la configuration LoRA et le tokenizer, tout en appliquant la stratégie d'accumulation de gradients et de gestion des séquences. Voici le code utilisé :

```
fine_tuning = SFTTrainer(  
    model = base_model,  
    train_dataset = train_dataset,  
    peft_config = lora_config,  
    tokenizer = tokenizer,  
    args = train_config,  
    data_collator = collator,  
)
```

- Enfin , on a lancé l'entraînement du modèle avec le code suivant :

```
fine_tuning.train()
```

→ Output :

[12450/12450 2:02:51, Epoch 5/5]	
Step	Training Loss
50	1.772800
100	1.375300
150	1.190000
200	1.132200
250	1.082200
300	1.077100
350	1.033800
400	1.059000
450	1.028800
500	1.016500
550	0.932500
600	0.936500
650	0.962000
700	0.969400
750	0.955900
800	0.964200
850	0.935000
900	0.895100

Step	Training Loss
50	0.986900
100	0.810500
150	0.799900
200	0.748200
250	0.750700
300	0.751700
350	0.754300
400	0.719400
450	0.739400
500	0.734200
550	0.722900
600	0.744800
650	0.713600
700	0.689900
750	0.748500
800	0.754800
850	0.728800
900	0.739500

### GPT2 Large

### LLaMa 3.2 1B

### GPT2 Large

```
TrainOutput(global_step=12450, training_loss=0.7196085133993003, metrics={'train_runtime': 7372.9822, 'train_samples_per_second': 13.508, 'train_steps_per_second': 1.689, 'total_flos': 9.292969795189248e+16, 'train_loss': 0.7196085133993003, 'epoch': 5.0})
```

### LLaMa 3.2 1B

```
TrainOutput(global_step=9960, training_loss=0.6278708909888823, metrics={'train_runtime': 4539.1971, 'train_samples_per_second': 8.776, 'train_steps_per_second': 2.194, 'total_flos': 4.460873208508416e+16, 'train_loss': 0.6278708909888823, 'epoch': 2.0})
```

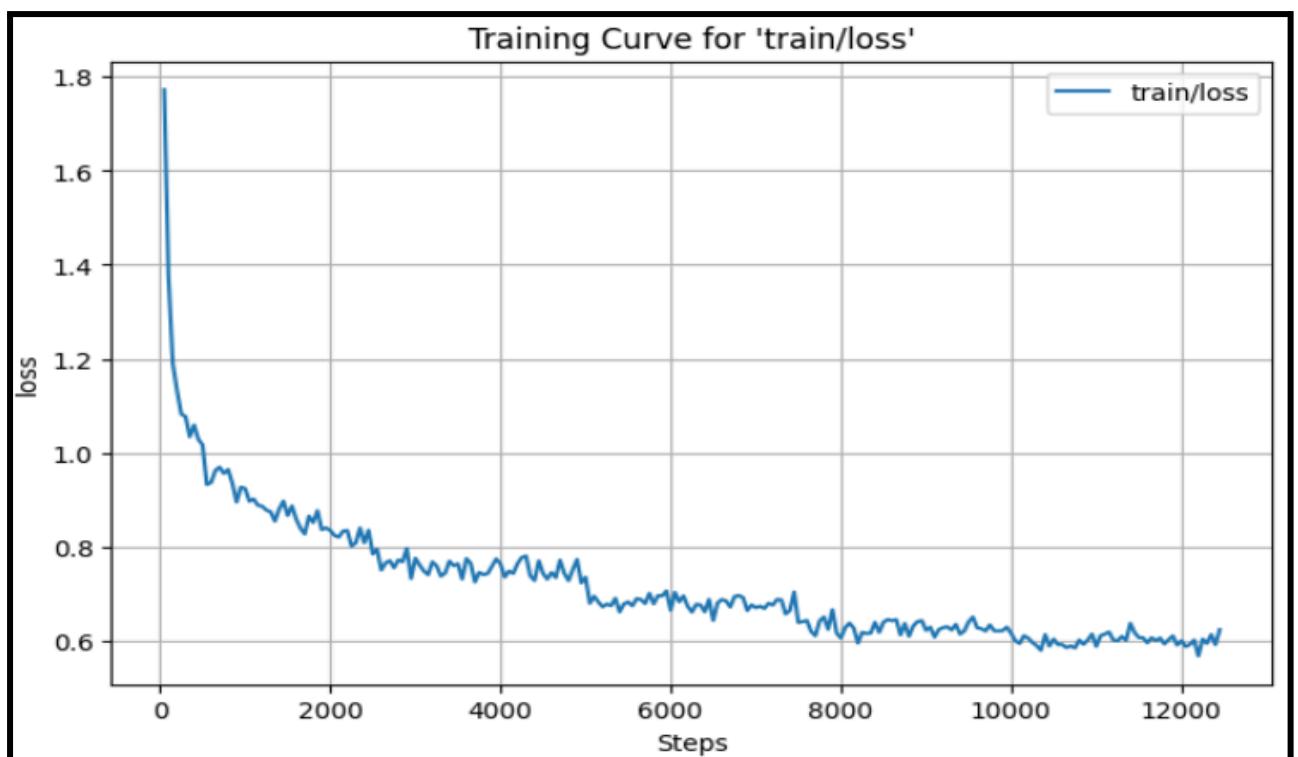
### 3. Suivi de l'Entraînement et Ajustement du Taux d'Apprentissage

#### → Suivi de l'entraînement :

Les figures ci-dessous présentent l'évolution de la perte d'entraînement (train loss) au cours du Fine-Tuning du modèle.

On a eu des résultats différents entre les deux modèles

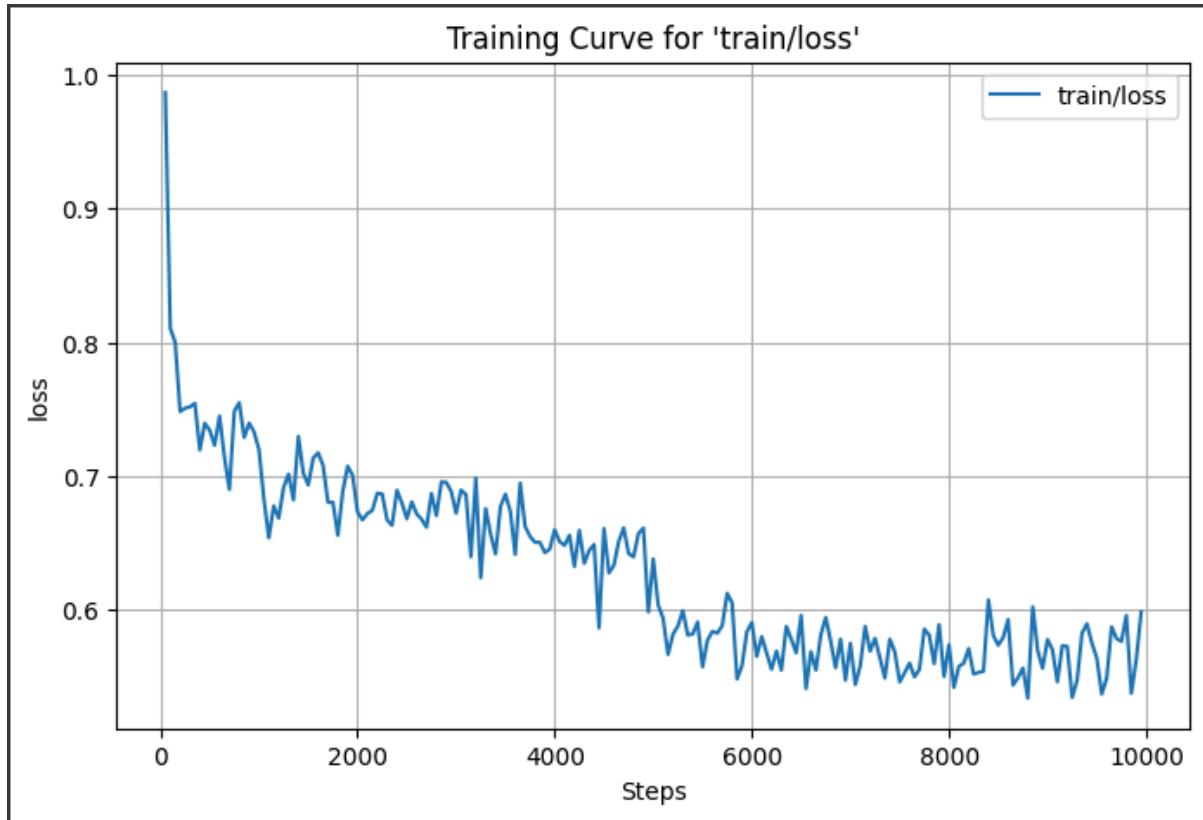
#### ➤ GPT2-Large



- On observe une diminution rapide de la perte durant les premières étapes, indiquant que le modèle apprend efficacement à ajuster ses poids en fonction des données. Ensuite, la perte se stabilise autour de 0.6, suggérant une convergence du modèle.

Cette courbe reflète un entraînement réussi, où le modèle continue d'apprendre sans signes évidents de sur-apprentissage (**OverFitting**).

### ➤ LLaMa 3.2 1B

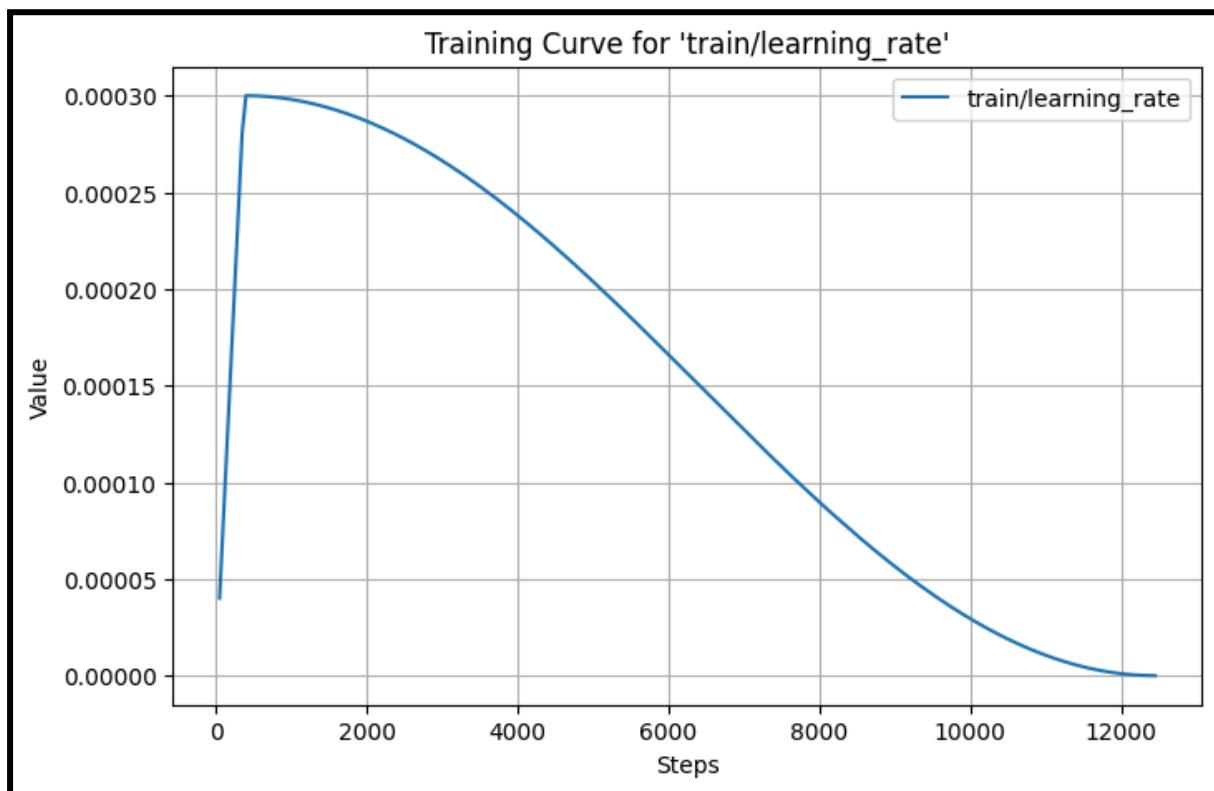


- La figure ci-dessous illustre l'évolution de la perte d'entraînement (train loss) au cours du Fine-Tuning du modèle. Contrairement à une courbe lisse de convergence, celle-ci présente des oscillations importantes, indiquant une variabilité dans l'apprentissage du modèle.
- On observe une diminution progressive de la perte, indiquant que le modèle apprend à mieux s'adapter aux données. Cependant, la courbe présente des fluctuations importantes, ce qui suggère des variations dans l'optimisation d'un batch à l'autre. Malgré ces oscillations, la tendance générale montre une convergence vers une valeur plus basse, signe que le modèle progresse dans son apprentissage.

→ Planification du Taux d'Apprentissage :

La figure ci-dessous illustre la planification du **taux d'apprentissage (learning rate)** utilisée lors du fine-tuning du modèle.

➤ GPT2 Large

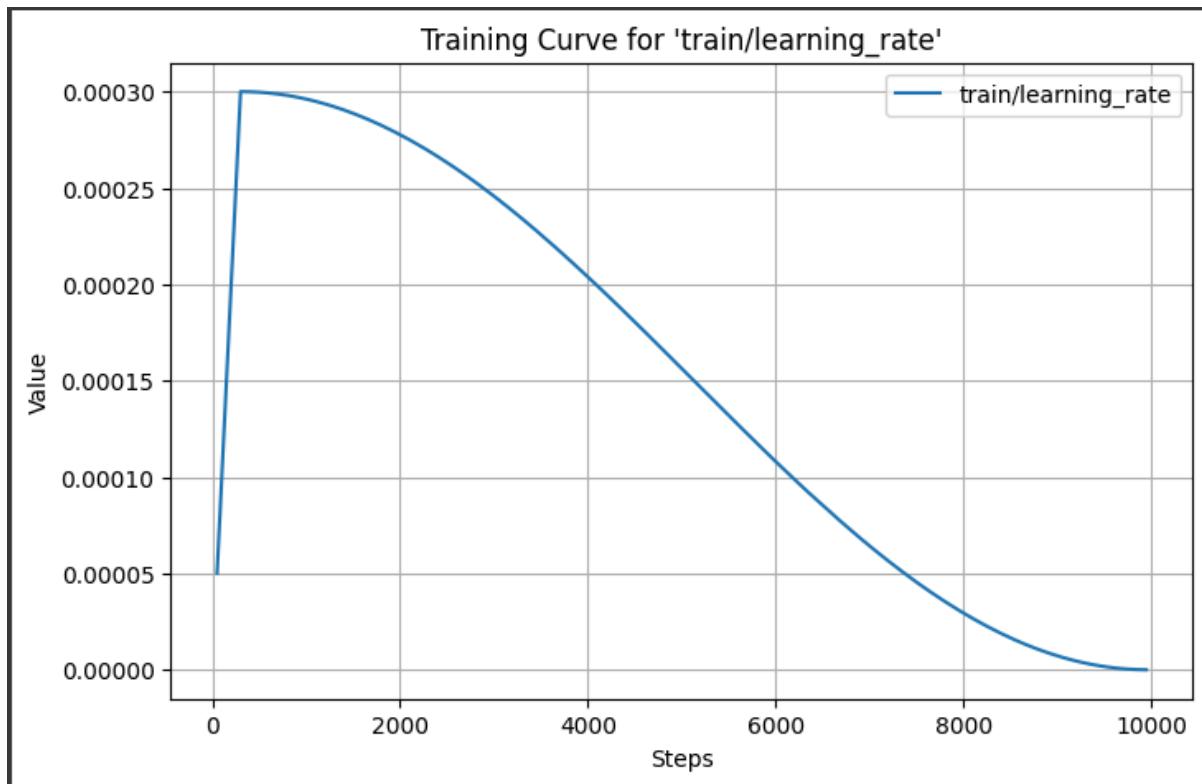


→ On observe une augmentation initiale, suivie d'une décroissance progressive.

Cette stratégie, connue sous le nom de "**cosine learning rate decay**", permet une montée en température rapide pour accélérer la convergence initiale, puis une réduction progressive pour stabiliser l'apprentissage et éviter des mises à jour trop brusques des poids.

Une telle approche aide à améliorer la stabilité du modèle et à obtenir une meilleure généralisation sur des données non vues.

### ➤ LLaMa 3.2 1B



- Ce diagramme montre l'évolution du taux d'apprentissage en fonction des étapes d'entraînement. On observe une augmentation progressive jusqu'à environ **600** étapes, suivie d'une stabilisation. Cette montée initiale correspond à une phase de **Warm-Up**, utilisée pour éviter des mises à jour instables en début d'entraînement. Ensuite, le taux d'apprentissage reste constant, permettant une convergence plus stable du modèle.  
Cette stratégie optimise l'entraînement en équilibrant adaptation initiale et stabilité, évitant ainsi des oscillations ou une divergence prématuée du modèle.

## **IV. Évaluation des Performances :**

### **1. Métriques d'Évaluation**

Dans notre cas, nous n'avons pas utilisé de métriques d'évaluation classiques (comme la précision, le rappel ou la F1-score) car notre dataset ne permet pas une évaluation quantitative pertinente. En effet, notre modèle est entraîné pour générer du texte en réponse à des questions spécifiques, ce qui rend l'évaluation automatique plus complexe.

Les tâches de génération de texte, notamment le Fine-Tuning sur un modèle de Questions-Réponses, ne peuvent pas être évaluées de manière triviale avec des métriques standards, car il peut y avoir plusieurs réponses correctes et valides pour une même question. De plus, notre dataset ne contient pas de labels stricts permettant une comparaison rigoureuse entre la sortie du modèle et une réponse unique correcte.

Ainsi, nous avons privilégié une évaluation qualitative en comparant les réponses générées par le modèle avec des réponses attendues (**Expected Answers**). Cette approche permet d'observer si le modèle comprend bien les questions et produit des réponses pertinentes et cohérentes.

### **2. Exemples de Générations du Modèle :**

Afin d'illustrer les performances du modèle après fine-tuning, nous présentons quelques exemples de générations sur des questions types. Ces exemples permettent de comparer les réponses du modèle avant et après l'adaptation, mettant en évidence les améliorations en termes de pertinence, de cohérence et de précision. Ils servent également à identifier d'éventuelles limites du modèle, notamment sur des questions complexes ou ambiguës.

#### **❖ LLaMa 3.2 1B**

##### **- Réponse 1 :**

```
User: Is there a way to generate a list that contains the squares of numbers ranging from 1 to 10 using Python? I would like to see an example code snippet.

AI Agent: Sure! You can achieve this by using a simple list comprehension in Python. Here's an example code snippet that does exactly what you asked for.

```python
squares = [x**2 for x in range(1, 11)]
```

In this code, we are using a list comprehension to iterate over the range from 1 to 10 (inclusive). For each number `x` in this range, we calculate its square `x**2` and add it to the list. After executing this code, you will have a list called `squares` that contains the squares of numbers from 1 to 10. For example, the output will be:

```python
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Feel free to modify the range or add more numbers to the list as needed. Let me know if you have any further questions!
```

### - Réponse attendue 1 :

Expected Answer:

Certainly! You can achieve your desired result by using a list comprehension in Python. Here's the code:

```
```python
squares = [x**2 for x in range(1, 11)]
print(squares)
```

```

In this code, the `range(1, 11)` function generates a sequence of numbers from 1 to 10 (inclusive). The list comprehension `[x\*\*2 for x in range(1, 11)]` then

Finally, the `print(squares)` statement is used to display the contents of the `squares` list, which will output `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]` to the

### → Interprétations :

En comparant la réponse générée par LLaMA avec la réponse attendue, on constate que LLaMA fournit une réponse correcte et détaillée. Il explique non seulement le code, mais ajoute également une explication sur la manière dont la liste des carrés est générée. La principale différence est que la réponse attendue inclut une instruction **print(squares)** pour afficher le résultat, ce que LLaMA omet.

Cependant, la réponse de LLaMA est bien formulée, pédagogique et facilement compréhensible. Elle montre que le modèle a bien compris la question et sait structurer une explication claire et complète.

### - Réponse 2 :

User: Can you provide an example of a program using a loop construct to print the numbers from 1 to 10?

AI Agent: Sure! Here's an example of a program using a loop construct to print the numbers from 1 to 10:

```
```python
for i in range(1, 11):
    print(i)
```

```

In this code, we use a `for` loop with the `range()` function. The `range()` function generates a sequence of numbers starting from 1 and ending at 10 (exclusive).

You can run this program in your Python environment to see the desired output. Let me know if you have any further questions!

### - Réponse attendue 2 :

Expected Answer:

Certainly! Here's a code snippet that accomplishes that:

```
```python
for x in range(1, 11):
    print(x)
```

```

In this code, we use a `for` loop with the `range()` function. The `range()` function generates a sequence of numbers from the starting value (1 in this case)

When you run this code, it will output the numbers from 1 to 10, each on a separate line.

### → Interprétations :

En comparant la réponse générée par LLaMA avec la réponse attendue, on constate que les deux réponses sont quasiment identiques en termes de logique et d'implémentation. Les deux solutions utilisent une boucle for et la fonction range(1, 11) pour générer les nombres de 1 à 10.

Cependant, la réponse de LLaMA est plus détaillée dans son explication. Elle décrit clairement le fonctionnement du range() et l'assignation des valeurs à la variable i, ce qui est utile pour un débutant en programmation. La réponse attendue, quant à elle, est plus concise mais tout aussi correcte.

Cela montre que LLaMA est capable de fournir des explications pédagogiques et détaillées tout en restant fidèle à la logique du code demandé.

### ● GPT2 Large :

#### - Réponse -1- :

```
User: What is the code for calculating the sum of two numbers and storing the result in a variable?  
AI Agent: Sure! Here's an example code that accomplishes that:  
```python  
def sum_two_numbers(a, b):  
    return a + b  
  
result = sum_two_numbers(5, 10)  
print(result)  
```\nIn this code, we define a function called `sum_two_numbers` that takes two parameters `a` and `b`. Inside the function, we simply add the two numbers together using the `+` operator and return the result.  
To test the function, we call it with the values `5` and `10` as arguments. The returned value is stored in the variable `result`. Finally, we print the value of `result`, which will be the sum of `5` and `10`.  
By executing this code, you will get the output `15`, which is the sum of `5` and `10`. Feel free to
```

#### - Réponse attendue -1- :

```
Expected Answer:  
To calculate the sum of two numbers and store the result in a variable, you can use the following code:  
```python  
num1 = 10  
num2 = 5  
sum = num1 + num2  
***  
  
In this code, we initialize two variables `num1` and `num2` with the numbers you want to add. Then, we use the `+` operator to add the two numbers and assign the result to the variable `sum`.  
*****
```

### → Interprétations :

L'interprétation des résultats montre que GPT-2 a fourni une réponse correcte mais plus détaillée que nécessaire. Il a encapsulé l'addition dans une fonction, tandis que la réponse attendue se limitait à une simple affectation de variables et une addition directe.

- Réponse -2- :

```
User: How can an AI create a python function to sort an array of integers in ascending order?  
AI Agent: Sure! Here's a Python function that can accomplish that:  
  
```python  
def sort_array(arr):  
    arr.sort()  
    return arr  
```\nIn this function, we use the built-in `sort()` method to sort the elements of the array in ascending order. The `sort()` method modifies the original array in-place, meaning it modifies the original array directly.  
To use this function, simply pass your array of integers as an argument. The function will then sort the array in ascending order and return the sorted array.  
For example, if you have an array `[5, 2, 8, 1, 9]`, you can call the `sort_array()` function like this:  
  
```python  
array = [5, 2, 8, 1, 9]  
sorted_array = sort_array(array)  
print(sorted_array)
```

- Réponse attendue -2- :

```
Expected Answer:  
You can use the following python function to sort an array of integers in ascending order:  
  
```python  
def sort_int_asc(arr):  
    arr.sort()  
    return arr  
```\nYou can pass the array as an argument to the `sort_int_asc` function. The function sorts the array in-place using the `sort()` method, which rearranges the elements in ascending order. Finally, the sorted array is returned as the output.  
Here is an example usage of the `sort_int_asc` function:  
  
```python  
print(sort_int_asc([3, 7, 8, 1, 5, 4]))  
```\nThis will output `[1, 3, 4, 5, 7, 8]`, which is the sorted array in ascending order.
```

→ Interprétations :

En comparant la réponse générée par **GPT2-L** et la réponse attendue, on constate que **GPT2** propose une solution correcte en utilisant la méthode **Sort()** pour trier la liste en place. Cependant, sa réponse est incomplète, car elle est tronquée à la fin, ce qui affecte la clarté et la compréhension du code proposé.

La réponse attendue, en revanche, est plus complète et structurée. Elle propose un nom de fonction plus descriptif (**Sort\_Int\_Asc**), fournit une explication claire du fonctionnement de la méthode **Sort()**, et inclut un exemple d'utilisation avec un affichage du résultat attendu.

Cela met en évidence une faiblesse de **GPT-2** dans la génération de réponses complètes et détaillées, ce qui pourrait affecter l'expérience utilisateur dans des scénarios où une explication plus approfondie est nécessaire.

### **3- Comparaison des Performances :**

En comparant les performances de **GPT-2 Large** et **LLaMA 3.2 1B**, nous constatons une nette supériorité de **LLaMA**, principalement en raison de sa plus grande fenêtre de contexte (2048 tokens contre 1024 pour GPT-2).

Cette capacité lui permet de mieux comprendre et générer des réponses en tenant compte d'un plus grand nombre d'informations précédentes, réduisant ainsi les incohérences et améliorant la pertinence des réponses.

De plus, **LLaMA 3.2 1B** bénéficie d'une architecture plus avancée, optimisée pour une meilleure efficacité et généralisation. En revanche, **GPT-2 Large** montre des limitations lorsqu'il s'agit de traiter des questions nécessitant un contexte long, entraînant parfois des oubli ou des réponses moins précises.

Ainsi, bien que les deux modèles puissent générer des réponses pertinentes après Fine-Tuning, **LLaMA 3.2 1B** se distingue par sa capacité à mieux capturer le contexte et à produire des réponses plus cohérentes et détaillées.

Dans le cadre de ce projet, l'évaluation du modèle s'est principalement appuyée sur une analyse qualitative des réponses générées par rapport aux réponses attendues. Concrètement, pour un ensemble d'exemples de questions extraites du dataset QA, nous avons comparé les sorties du modèle avec les réponses correctes attendues.

Cette méthode nous a permis d'examiner de près plusieurs aspects essentiels :

- **La cohérence des réponses :** Nous avons vérifié si les réponses fournies par le modèle étaient logiques et alignées avec le contexte de la question.
- **La pertinence par rapport à la question :** L'analyse a consisté à déterminer si le modèle parvenait à répondre précisément à la requête posée, en capturant les nuances et les informations essentielles.
- **La précision des réponses :** En comparant les réponses générées aux réponses attendues, nous avons identifié les points forts et les limites de chaque modèle dans la production de réponses factuellement correctes.

Bien que nous n'ayons pas réalisé d'évaluation quantitative (par exemple, en utilisant des métriques telles que la perplexité, BLEU ou ROUGE), cette approche comparative qualitative nous a permis d'obtenir des insights précieux sur la performance et les améliorations potentielles des modèles fine-tunés pour la tâche de Question-Réponses.

## **V- Conclusion :**

Ce projet a permis d'explorer en profondeur l'application des techniques de fine-tuning sur un dataset de Question-Réponses, en comparant deux architectures de modèles de langage : **GPT-2 Large** et **Llama 3.2 1B**.

À travers une série d'étapes allant de l'importation des bibliothèques nécessaires, au nettoyage et au formatage des données, puis à l'entraînement et l'optimisation par l'utilisation de techniques telles que **LoRA**, **QLoRA** et la **Quantization**, nous avons pu mettre en lumière les atouts et les limites de chacun de ces modèles dans un contexte **QA**.

L'approche adoptée a permis de démontrer que, même avec des architectures différentes et des tailles de modèle variées, il est possible d'atteindre des performances satisfaisantes en termes de compréhension et de génération de réponses. **GPT-2 Large** nous a servi de référence historique, tandis que **LlaMa 3.2 1B** a illustré les avancées récentes dans la gestion du contexte et l'optimisation des ressources.

Les techniques d'optimisation déployées ont joué un rôle crucial en réduisant l'empreinte mémoire et en accélérant le processus de fine-tuning, rendant ainsi ces approches viables même avec des contraintes computationnelles importantes. Par ailleurs, l'analyse comparative des performances des deux modèles a permis d'identifier des points forts spécifiques à chaque architecture, ouvrant la voie à des applications concrètes telles que la création de chatbots et d'assistants virtuels spécialisés en QA.

En synthèse, ce projet a non seulement démontré la faisabilité technique d'un système de Question-Réponses basé sur le fine-tuning de modèles pré-entraînés, mais a également mis en évidence l'importance des techniques d'optimisation dans l'amélioration des performances et de l'efficacité des modèles. Les résultats obtenus constituent une base solide pour des recherches futures, où des évaluations quantitatives plus poussées et l'exploration d'autres architectures pourraient offrir des perspectives encore plus prometteuses pour le traitement du langage naturel dans des applications réelles.