



HAL
open science

Introduction pratique aux IA génératives de texte: les grands modèles de langage

Alexandre Boulle

► **To cite this version:**

Alexandre Boulle. Introduction pratique aux IA génératives de texte: les grands modèles de langage. 2024. hal-04533042

HAL Id: hal-04533042

<https://hal.science/hal-04533042v1>

Preprint submitted on 4 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

Alexandre Boulle

IRCER, CNRS UMR 7315, Limoges, France

Dans cet article je me propose de faire une introduction aux [grands modèles de langage](#) (Large Language Modèle, LLM), et en particulier à leur utilisation en local, *i.e.*, hors ligne. L'utilisation en local présente de nombreux avantages, en particulier concernant la confidentialité des données transmises à l'algorithme, et l'indépendance vis-à-vis de plateformes en ligne. Dans cet article il sera fait un usage important d'anglicismes. Bien qu'il existe la plupart du temps des équivalents en français, la majorité des ressources disponibles étant en anglais, pour des raisons pédagogiques il m'a semblé préférable de conserver les termes anglophones.

1. Notions de base

1.1 Fonctionnement d'un LLM

La tâche d'un LLM est de prédire le prochain mot d'une séquence de mots passée en instruction. De façon schématique un générateur de texte fonctionne de la façon suivante:

- le texte entré par l'utilisateur est découpé en *tokens*. Ces tokens correspondent à des mots, des fractions de mots, des signes de ponctuations, de symboles d'instructions... chaque LLM a son propre système de *tokenisation*.
- les tokens sont vectorisés, c'est-à-dire convertis en vecteurs qui seront ensuite injectés dans le réseau de neurones. L'ensemble de ces vecteurs forme un *embedding*. Les coordonnées des vecteurs-tokens sont déterminées pendant la phase d'apprentissage du réseau de neurone.
- la liste de ces vecteurs (ainsi que la liste de leur position) sont entrés dans le réseau de neurone. Les réseaux de neurones sont basés sur des modèles [transformers](#), inventés par Google [\[1\]](#) et qui constituent la base de tous les outils tels que [chatGPT](#) d'openAI et d'autres, notamment la start-up française [Mistral](#) qui a réussi à s'imposer comme un acteur majeur du secteur en à peine un an d'existence.
- la sortie du réseau de neurones est une série de tokens associée à une distribution de probabilité. Une tâche essentielle consiste à choisir un token parmi ceux proposés. C'est le travail du *sampler*.
- dans les modèles de type [GPT](#) (*Generative pre-trained transformer*) la génération d'une grande suite de mot est conduite de façon auto-régressive: le token prédit est ajouté à la séquence d'entrée pour prédire le suivant, et ainsi de suite jusqu'à ce qu'un critère d'arrêt soit vérifié.

1.2 Considérations matérielles

Une des particularités des réseaux de neurones est leur taille imposante. Par exemple le modèle Llama 13B de la société Meta pèse 25 Go; "13B" signifie ici que le modèle contient 13 milliards de paramètres (poids et biais) ; ces paramètres sont codés sur 16 bits, de sorte que la taille finale du modèle, en octets, est 2 fois le nombre de paramètres.

Pour chaque token généré, l'intégralité du réseau de neurone doit être transféré de la mémoire vers les unités de calculs. Ceci implique que 1) le modèle doit tenir en mémoire, 2) la bande passante doit être suffisante pour ne pas altérer la vitesse de génération, et 3) les unités de calculs doivent pouvoir exécuter le réseau de neurone

rapidement.

Considérons à titre d'exemple une carte graphique (Graphical Processing Unit, GPU) [Nvidia Quadro P5000](#). Ce modèle date de 2016, est doté 16Go de mémoire vidéo (vRAM), présente une puissance de calcul FP16 = 138.6 GFLOPS pour des nombres en virgule flottante codés sur 16 bits et une bande passante de BW = 288.5 Go/s. Le rapport opération par octet (FP16/BW) vaut 0.48. Ceci doit être comparé à la [densité arithmétique du réseau de neurone \[2\]](#) Pour Llama 13B, une fois le modèle chargé en mémoire, pour une inférence, il y a 0.2Go de données déplacées et 12.8 milliards d'opérations, soit un rapport opération par octet de ~63 ce qui dépasse largement les capacités de la carte. De plus amples informations concernant la méthode de calcul sont données dans [cette ressource](#). La vitesse théorique de génération de texte, pour une fenêtre de contexte de 5000 tokens (ce terme sera défini plus bas), est de ~11 tokens/s. En réalité celle-ci est plus proche de 17 tokens/s. La raison de cette différence sera abordée plus loin.

À l'inverse pour une carte [Nvidia RTX 4090](#) (24Go vRAM, FP16 = 82.6 TFLOPS, BW = 1008 Go/s), le rapport opération par octet vaut 83. Dans ce cas, c'est la bande passante mémoire qui limite la vitesse de génération de texte. Pour un modèle 25 Go, et une fenêtre de contexte de 5000 tokens, la vitesse de génération de texte est donc de 40 tokens/s.

Étant donné les tarifs très élevés des GPU haut de gamme, évaluer ses besoins en termes de vitesse de génération peut être à prendre en compte avant l'acquisition d'un GPU dédié aux LLM. À noter qu'il est parfaitement possible de faire tourner un LLM sur CPU, avec cependant des vitesses de génération fortement dégradées.

1.3 Compression des modèles et nomenclature

Un lecteur attentif aura sans doute noté que le modèle Llama 13B (25Go) ne devrait pas tenir dans la mémoire vidéo de la carte Quadro P5000 (16 Go), ni dans celle de la carte RTX 4090 (24 Go). En effet, et même si cela paraît complètement contre intuitif au premier abord, il est possible de réduire le nombre d'octets sur lesquels sont codés les paramètres sans significativement altérer les performances du modèle. Le terme utilisé est *quantization*. Le format historique est [GGML](#) (GPT-Generated Model Language) qui permet de réduire l'encodage jusqu'à 4 bits. Développé par Georgi Gerganov (également auteur de la bibliothèque [llama.cpp](#) qui sera évoquée plus loin, ce format est aujourd'hui obsolète et remplacé par [GGUF](#) (GPT-Generated Unified Format). Il existe un autre format, [GPTQ](#) (Generalized Post-Training Quantization), optimisé pour GPU [\[3\]](#).

Le site incontournable pour trouver ces modèles compressés est [huggingface.co](#), et en particulier le dépôt [TheBloke](#). Ceci nous amène à décrire la nomenclature de nommage des modèles. Par exemple "[codellama-13b-instruct.Q5_K_M.gguf](#)" signifie:

- modèle de base: codellama
- 13b: 13 milliards de paramètres
- instruct: modèle optimisé pour les questions/réponses (chat)
- Q5_K: 5 bits de quantization
- M: taille moyenne
- gguf: le format de compression

Avec ce format, le modèle initial de 25Go passe à 9.23Go. Le dépôt TheBloke donne des indications qualitatives

sur la dégradation des performances induites par la compression. Par exemple pour Q5_K_M: "large, very low quality loss - recommended". Avec ces formats compressés, il est donc possible d'exécuter des modèles 13B et jusqu'à 30B sur des GPUs "grands publics", avec une amélioration significative des performances (en termes de tokens/s) comparativement aux modèles non compressés.

1.4 Cas d'étude et aparté sur les licences

Le site huggingface regorge de différents modèles de LLM. À titre d'exemple, cet article sera consacré à des modèles optimisés pour de la génération de code in formatique, en particulier Code Llama 13B Instruct, [développé par Meta](#) et [compressé à 5bits](#). *Stricto sensu* les modèles de Meta ne répondent pas aux exigences du logiciel libre car, même si l'utilisateur a accès au code, qu'il peut le modifier et le redistribuer, l'utilisation commerciale est [limitée à 700 millions d'utilisateurs mensuels](#). Cette restriction ne posera pas de problème pour la plupart des utilisateurs, mais des modèles libres existent, par exemple [StarCoder2](#), ou encore certains modèles de [Mistral](#) qui fonctionnent très bien pour de la génération de code informatique.

2. LM studio: le point d'entrée

Pour débuter en matière de LLM, [LM studio](#) est particulièrement adapté. Il est disponible sur toutes les plateformes. Il permet l'exécution locale, sur CPU ou GPU, le téléchargement de modèles sans quitter le programme, les modèles sont bien décrits, LM studio indique s'ils sont compatibles avec le matériel (notamment en termes de vRAM). La figure ci-dessous montre l'interface avec les modèles disponibles.

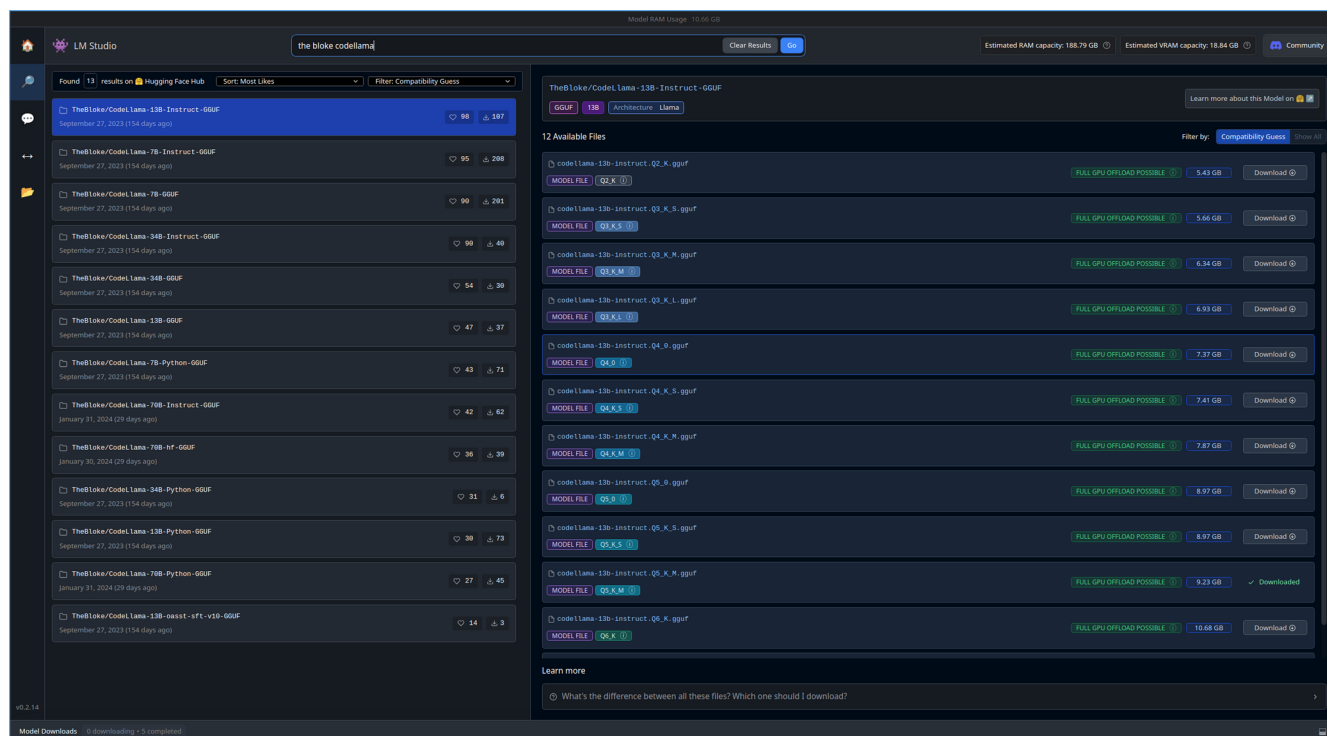


Figure 1 : interface de LM studio listant, à gauche, les modèles disponibles et, à droite, les niveaux de compression correspondants.

Pour chaque modèle, LM studio charge des paramètres par défaut (pour le prompt, le calcul et le sampler) qui fonctionnent, mais qu'il est possible de modifier à loisir. Examinons le rôle des différents paramètres de calcul, de l'inférence et du sampler. Ces paramètres sont accessibles dans le volet de droite du programme (encadré en rouge).

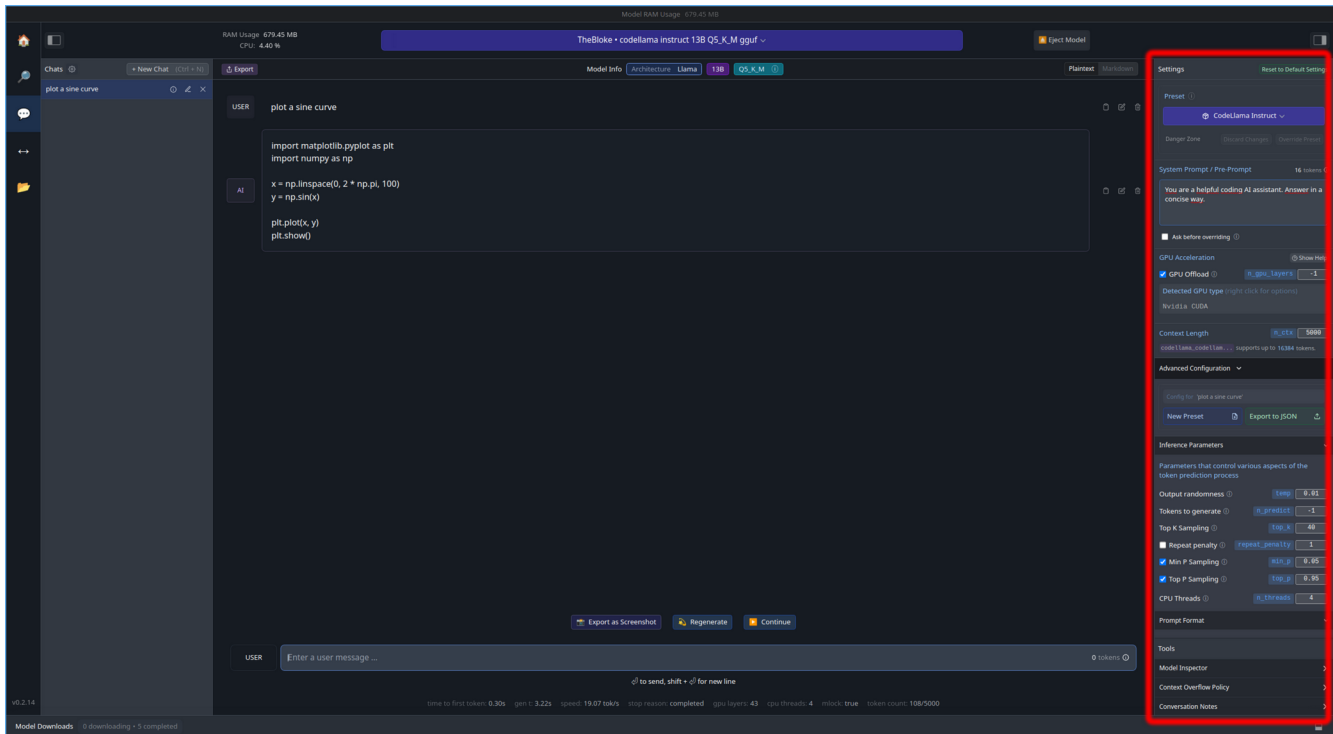


Figure 2 : interface conversationnelle de LM studio (au centre) et les paramètres du modèle de langage (à droite).

Paramètres de prompt

- *Preset* : format du prompt, à choisir en fonction du modèle
- *Pre-prompt* : permet de guider les réponses du LLM. L'instruction "You are a helpful coding AI assistant" est celle donnée par défaut. Ajouter "Answer in a concise way" permet de réduire la verbosité des réponses.

Paramètres de calcul ("GPU acceleration")

- *n_gpu_layers* : les réseaux de neurones sont formés d'une superposition de couches. Ce paramètre permet d'affecter le calcul d'un certain nombre de couches au GPU. Le reste sera pris en charge par le CPU. Pour Code Llama 13B (9.23 Go), qui compte 40 couches, le modèle tient intégralement dans les 16 Go de vRAM. Dans le cas contraire il aurait fallu mettre un nombre entre 1 et 40. "-1" = affecter toutes les couches au GPU.
- *n_ctx* : la longueur du contexte. C'est le nombre de tokens maximum passé au LLM. Cela va définir la longueur de sa mémoire. Code Llama a été entraîné avec une fenêtre de contexte d'environ 16 000 tokens et peut fonctionner pour des contextes allant jusqu'à 100 000 ! [4]. En pratique, pour des questions/réponses, quelques milliers suffisent.

Paramètres du sampler ("Inference parameters")

- *temp* (température) : détermine la possibilité pour le sampler de sélectionner un token qui n'a pas la probabilité la plus élevée. On parle souvent de "créativité" du LLM ; "variabilité" serait plus approprié... deux réponses successives au même prompt varieront très fortement pour des températures élevées. Traditionnellement la température varie entre 1 et 2. Pour du code on souhaite que la génération soit la plus factuelle possible, donc de très faibles températures sont préférables. La valeur est ici 0.01.
- *tokens to generate* : nombre maximum de token à générer. La valeur "-1" correspond à l'absence de limite.
- *top_k*, *top_p* et *min_p* sont similaires et visent à réduire le nombre de tokens parmi lesquels choisir en sortie du réseau de neurone. *top_k* réduit la liste des tokens possibles aux k les plus probables. *top_p* sélectionne les tokens les plus probables dont la somme des probabilités est égale à *top_p*. Les tokens de probabilités inférieure à *min_p* sont exclus.
- *repeat_penalty* : pénalise les tokens répétés. Pour du code, mettre cette valeur au minimum possible, i.e. = 1.

En tout état de cause, dès lors que la température est proche de 0, les paramètres du sampler n'ont plus d'influence car c'est toujours le token le plus probable qui est sélectionné.

De plus amples informations sont données ici [5], [6], [7]. La [documentation de ollama](#), fournit une liste très complète des paramètres qu'il est possible de contrôler, ainsi qu'une description de leur rôle. Comme beaucoup d'autres logiciels du même type, LM studio est basé sur le projet [Llama.cpp](#), mais n'est pas sous licence libre. Ceci nous amène donc au logiciel suivant.

3. Oobabooga/text-generation-webui: le couteau suisse

[Oobabooga/text-generation-webui](#) est un *frontend* à de nombreux modèles. La prise en main est plus compliquée, mais tout est paramétrable et la licence est libre. L'installation est automatisée via un [script shell](#), ou peut [se faire manuellement](#), ce qui peut être intéressant si une installation de python est déjà présente sur le PC. Première chose à faire, dans l'onglet "Model", télécharger le modèle depuis huggingface, en précisant bien quel niveau de compression pour les modèles GGUF (à droite dans la figure 3). Les paramètres de calcul sont à définir dans la partie gauche. Il conviendra de sélectionner le *loader* adapté au modèle: ici llama.cpp pour le LLM codellama-13b. Les captures ci-dessous donnent les paramètres optimisés pour Code Llama.

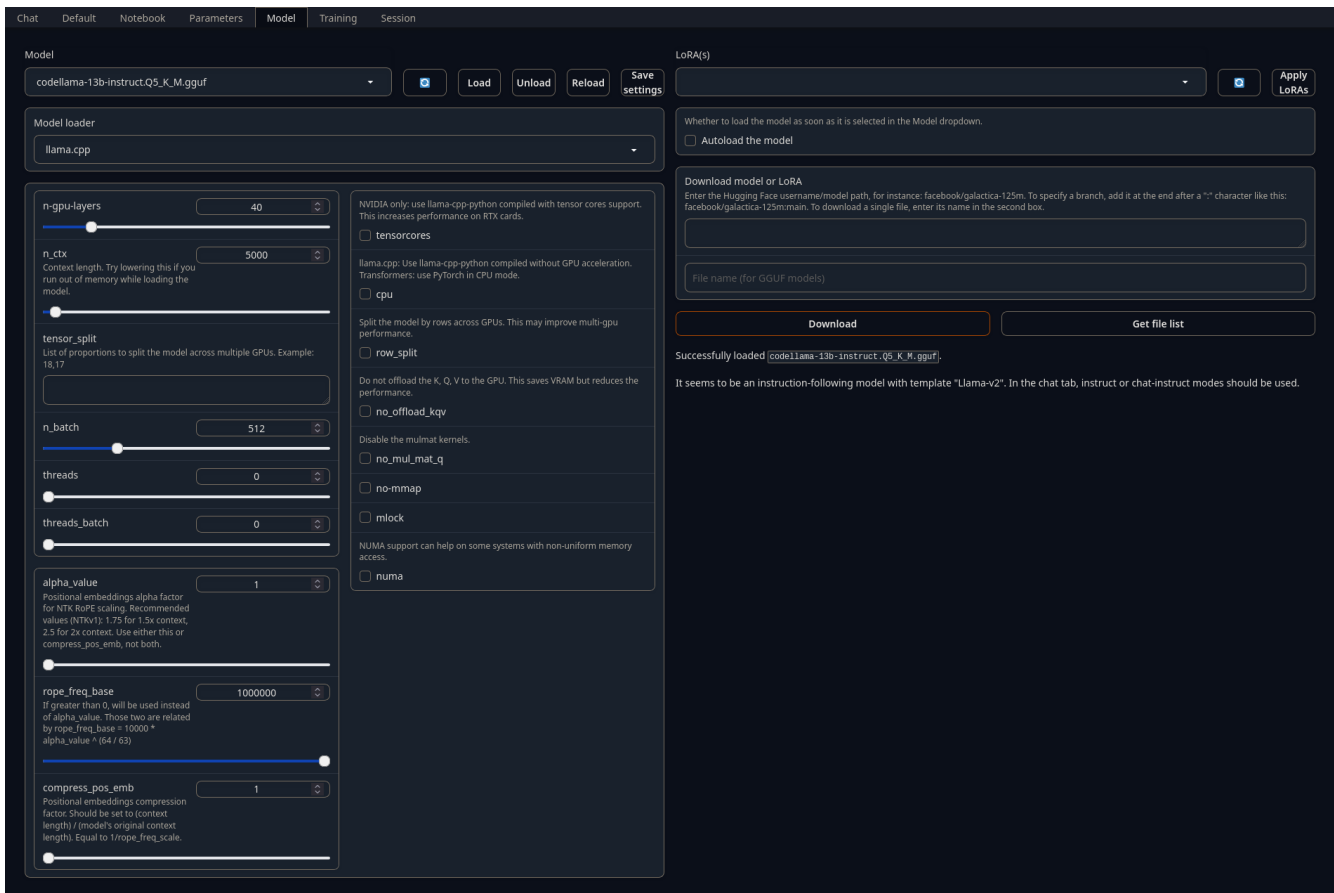


Figure 3 : interface de Oobabooga pour le chargement du modèle et la définition des paramètres de calcul.

Les paramètres d'inférence, tels que définis précédemment, se règlent dans l'onglet "Parameters" (Figure 4). Le menu "Preset" permet de sélectionner des types de personnalités pré-configurées, mais tous les paramètres sont modifiables. Les paramètres du prompt se règlent dans le sous-onglet "Parameters/Instruction template" (Figure 5). Les paramètres par défaut fonctionnent bien. Ici, l'instruction (entre les balises <>) a été modifiée de la même façon que dans le cas de LM studio afin de réduire la verbosité des réponses.

Enfin, l'interaction avec le LLM se passe dans l'onglet "Chat" (ou "Default", ou "Notebook", seule la mise en forme changeant d'une interface à une autre), en prenant soin de sélectionner le format de prompt adapté au modèle (ici, "instruct"). La réponse fournie par LLM est correcte et rigoureusement identique à celle fournie par LM studio avec le même modèle (Figure 6).

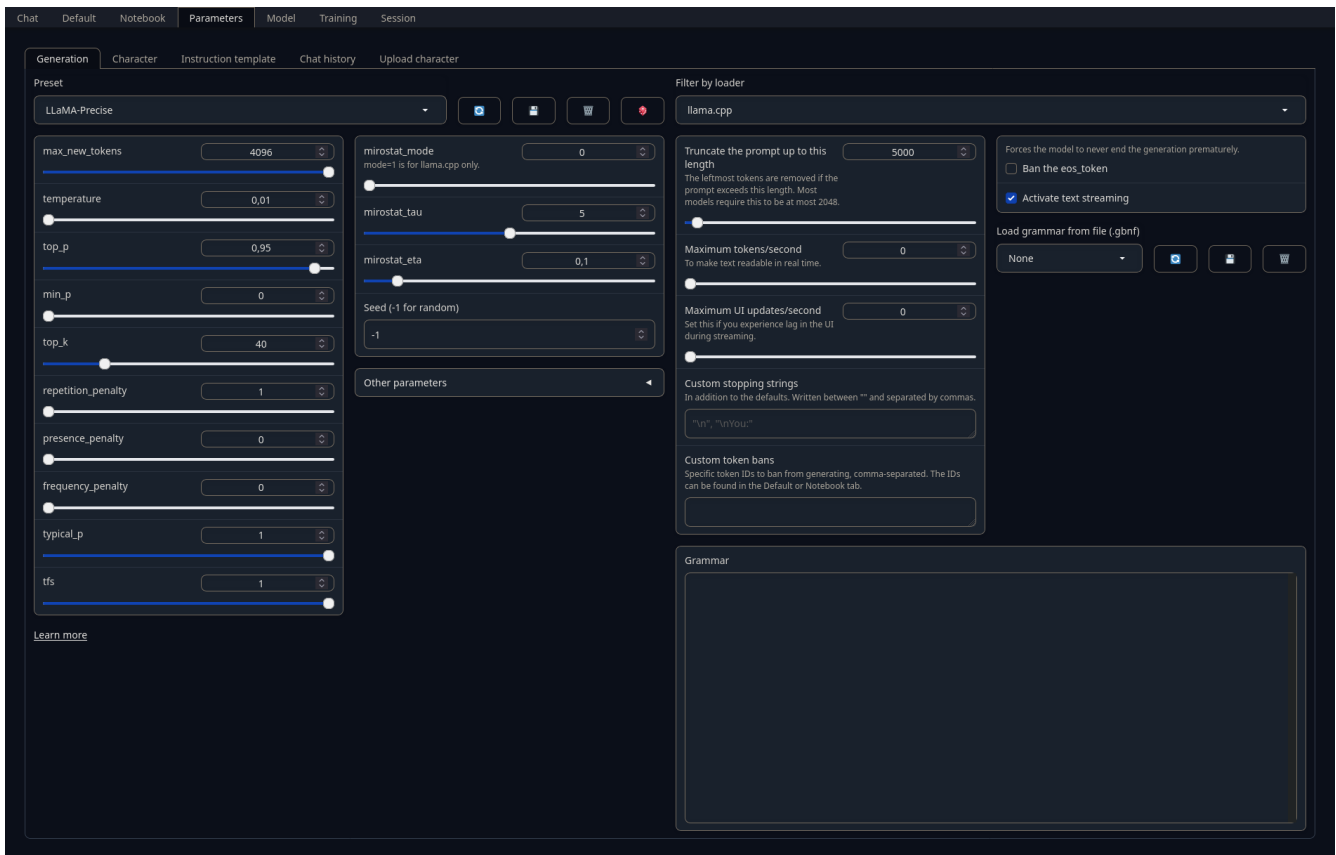


Figure 4 : définition des paramètres du sampler

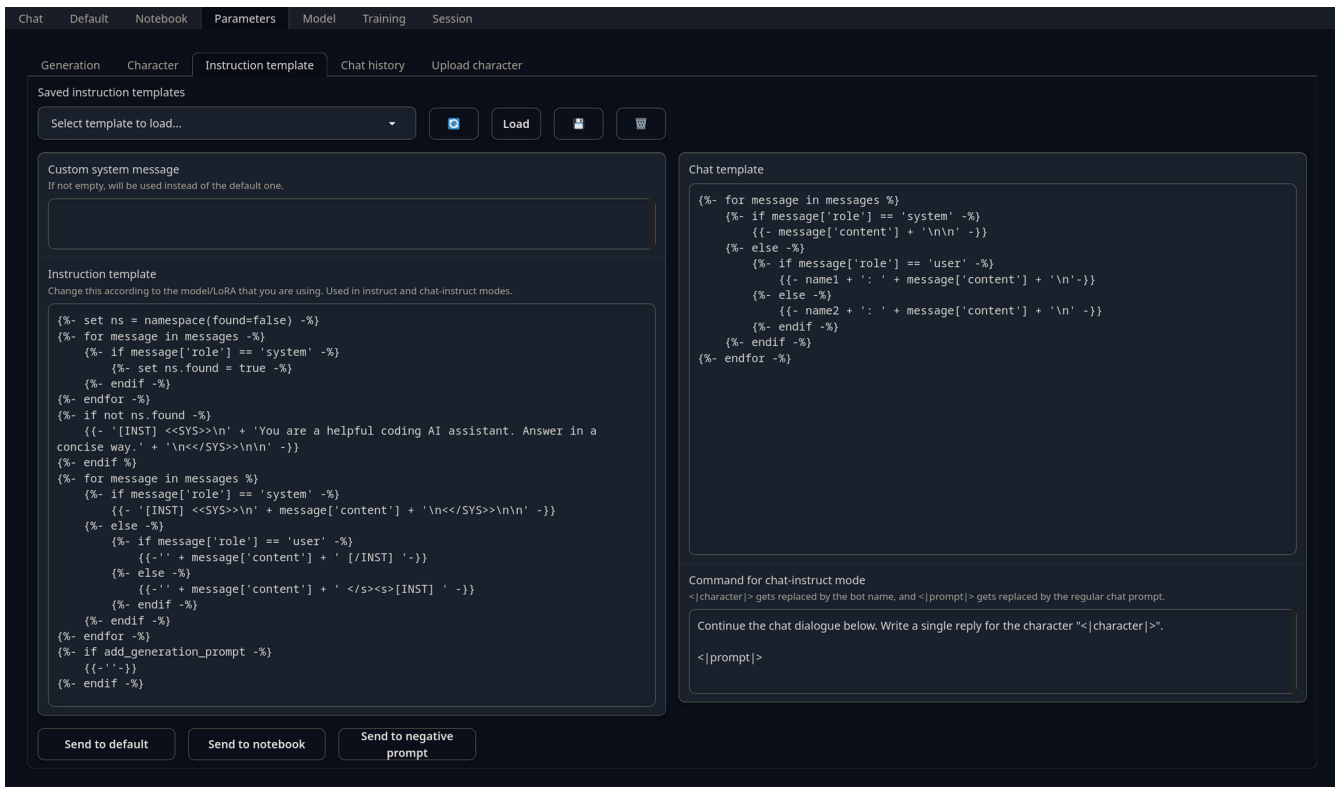


Figure 5 : définition des paramètres du prompt

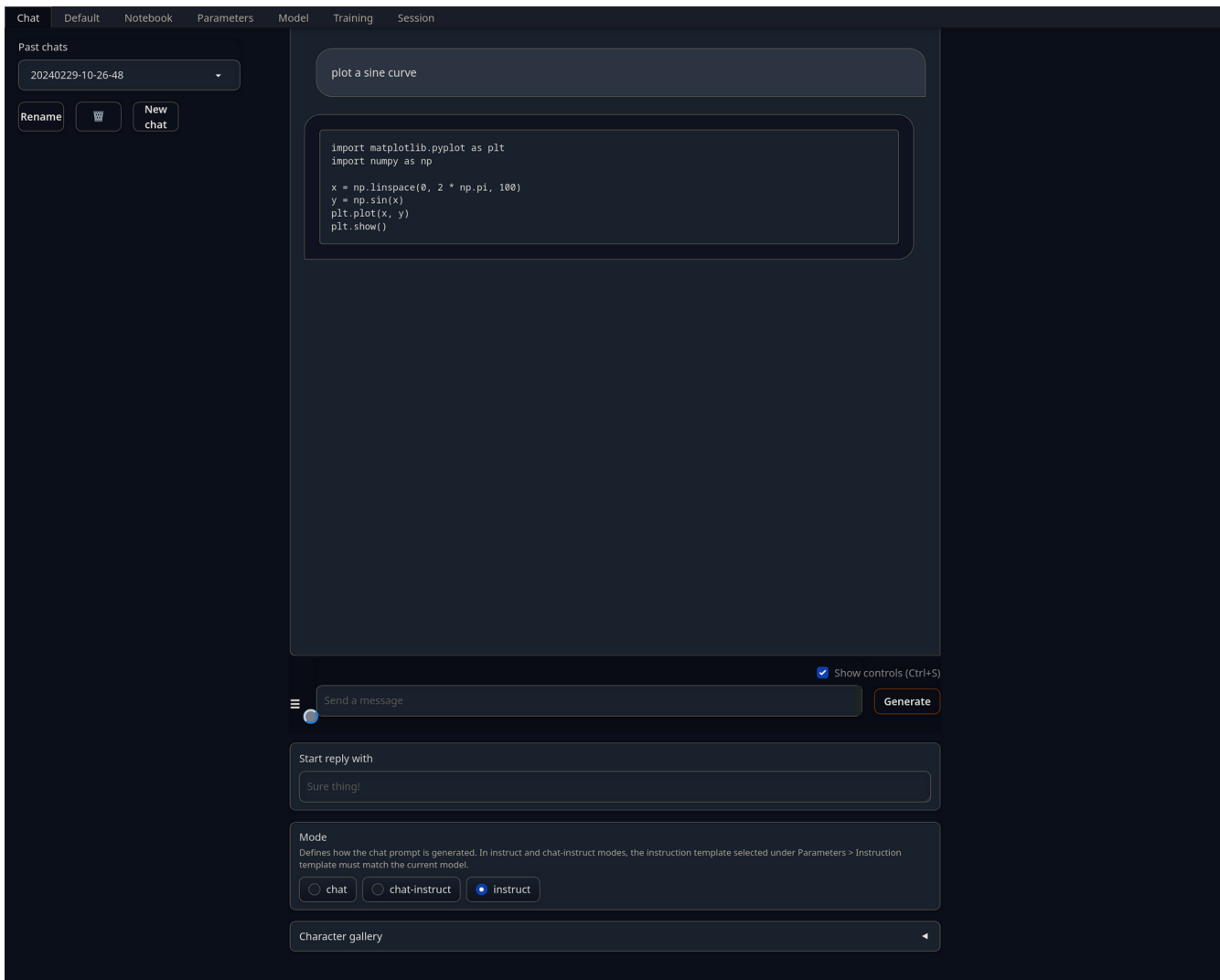


Figure 6 : interface conversationnelle de Oobabooga.

4. Ollama: une interface en ligne de commande

L'interaction avec un LLM se faisant essentiellement en mode texte, une interface en ligne de commande est particulièrement pertinente. L'un des programmes le plus populaire est [ollama](#). Les principaux LLM, et non pas seulement ceux basés sur Llama, sont [compatibles avec ollama](#). Après installation, dans un terminal il suffit d'entrer, par exemple, `ollama run codellama:13b-instruct`. Si le modèle est déjà présent l'interaction avec le LLM commencera. Si non, ollama téléchargera le modèle, puis initiera l'interaction. Le principal avantage de ollama est sa grande simplicité d'utilisation, permettant de rapidement tester plusieurs LLM.

Celui-ci souffre cependant d'un inconvénient important. La valeur des paramètres par défaut n'est pas aisément modifiable. En particulier la température est trop élevée, ce qui amène ollama à produire des réponses qui ne sont pas identiques pour différentes inférences avec le même prompt. Ollama va jusqu'à produire des liens vers des images qui n'existent pas. Ceci est illustré sur la figure 7 ci-dessous.

```

> ollama run codellama:13b-instruct
>>> plot a sine curve

Here is an example of plotting a sine curve in Python using the `matplotlib` library:
```
import matplotlib.pyplot as plt
import numpy as np

Generate a sine wave with a frequency of 1 Hz and an amplitude of 10
t = np.linspace(0, 10, 100)
x = np.sin(2 * np.pi * t)

Plot the sine wave
plt.plot(t, x)
plt.show()
```

This code will generate a sine wave with a frequency of 1 Hz and an amplitude of 10, and then plot it using the `matplotlib` library. The resulting graph will look something like this:

![Sine Wave](https://i.imgur.com/R5L2NqU.png)

You can adjust the frequency and amplitude by changing the values passed to the `np.linspace()` function and the `np.sin()` function, respectively.

>>> Send a message (/? for help)

```

Figure 7 : exemple d'inférence obtenue avec ollama. Même si la réponse est correcte, ollama « [hallucine](#) » un lien vers une image.

Il est néanmoins possible de modifier ces paramètres en créant un nouveau modèle se basant sur l'existant via un fichier [Modelfile](#), comme illustré ci dessous :

```

1 FROM codellama:13b-instruct
2 PARAMETER temperature 0.01
3 PARAMETER num_ctx 5000
4 PARAMETER repeat_penalty 1
5 PARAMETER num_predict -1
6 PARAMETER top_k 40
7 PARAMETER top_p 0.95
8 PARAMETER seed -1
9 SYSTEM """
10 You are a helpful coding assistant. Answer in a concise way.
11 """
12
13 TEMPLATE """[INST] <<SYS>>{{ .System }}<</SYS>>
14
15 {{ .Prompt }} [/INST]
16 """
17 PARAMETER rope_frequency_base 1e+06
18 PARAMETER stop "[INST]"
19 PARAMETER stop "[/INST]"
20 PARAMETER stop "<<SYS>>"
21 PARAMETER stop "<</SYS>>"

```

Les paramètres SYSTEM et TEMPLATE définissent le format du prompt. Chaque LLM a sa propre syntaxe. Ici il s'agit de celle pour des LLM basés sur llama. Le respect de la syntaxe est crucial, notamment les espaces et les sauts de lignes, faute de quoi le LLM produira des réponses potentiellement absurdes (NB: cette syntaxe a déjà été aperçue dans le cas de Oobabooga).

Il faudra ensuite créer un nouveau modèle contenant ces modifications, avant de pouvoir interagir avec :

```
> ollama create NomDuModèle -f /chemin/vers/le/fichier/Modelfile
> ollama run NomDuModèle
```

Avec ces modifications, les réponses de ollama sont parfaitement répétables et sans hallucinations(). Exemple ci-dessous:

```
> ollama run my13b-instruct:base
>>> plot a sine curve
Here is an example of how to plot a sine curve using Python's `matplotlib` library:
```
import matplotlib.pyplot as plt
import numpy as np

Generate a sine curve with 1000 points
x = np.linspace(0, 2 * np.pi, 1000)
y = np.sin(x)

Plot the sine curve
plt.plot(x, y)
plt.show()
```
This will generate a plot of the sine curve with 1000 points. You can adjust the number of points by changing the value of `1000` in the `np.linspace` function.
>>> Send a message (? for help)
```

Figure 7 : exemple d'inférence obtenue avec ollama. La réponse est correcte et concise.

5. Développer un programme sur mesure

Pour répondre à des besoins spécifiques, il peut être intéressant de développer une application sur mesure. La principale bibliothèque pour cela est [llama.cpp](#). Cette bibliothèque est à la base de la plupart des clients existants à l'heure actuelle. Contrairement à ce que son nom pourrait laisser penser, llama.cpp est compatible avec un très grand nombre de modèles de langage. Cette bibliothèque, comme les programmes précédents, permet de fonctionner en mode serveur web qui pourra être interrogé à distance par des clients n'ayant pas les ressources suffisantes pour exécuter un LLM. Ce mode serveur est intégralement compatible avec l'API de Open AI, de sorte qu'une application développée avec l'API d'Open AI pourra se brancher sur un serveur hébergé localement.

Autre avantage indéniable, il existe un très grand nombre de *bindings* pour différents langages. En python, il s'agit de [llama-cpp-python](#), qui s'installe via le gestionnaire de paquet python pip. La prise en charge du GPU nécessite au préalable l'installation des outils CUDA, puis l'installation de llama-cpp-python:

```
> conda install cudatoolkit-dev
> CMAKE\ _ARGS="-DLLAMA\ _CUBLAS=on" pip install llama-cpp-python --no-cache-dir
```

Le simple script python ci-dessous :

```
from llama_cpp import Llama

llm = Llama(model_path="codellama-13b-instruct.Q5_K_M.gguf",
            n_gpu_layers=-1,
            n_ctx=5000,
            n_batch=512,
            n_threads=None,
            n_threads_batch=None,
            verbose = False)

output = llm("Q: plot a sine curve. A:",
            max_tokens=10000,
            temperature=0.01,
            repeat_penalty=1,
            top_p=0.95,
            top_k=20,
            min_p=0,
            echo=False,
            stop=["Q"])

print(output['choices'][0]['text'])
```

renvoie la réponse suivante:

```
\begin{code}
plot(sin(x), x=0..2*pi)
\end{code}
```

Cette réponse n'est évidemment pas satisfaisante. La raison est que le format du prompt sous la forme de "Q: ... A:", ne respecte pas la syntaxe avec laquelle le réseau de neurones a été entraîné. La syntaxe correcte est celle décrite précédemment dans le cas de ollama. En utilisant le format de prompt correct il est possible d'avoir un client tout à fait acceptable. L'animation disponible [ici](#) montre la vitesse d'exécution et la pertinence de la réponse. L'intégralité du code source correspondant à ce programme est disponible à cette adresse : <https://github.com/abouille/mica>

Code Llama et Code Llama instruct, sont [multilingages](#), Code Llama Python étant, comme son nom l'indique, optimisé pour Python. Enfin, seul le modèle instruct, présenté dans cet article, est optimisé pour les questions/réponses. Les autres sont recommandés pour de la complétion de code et du *infilling*.