# 一、概念

## 1.1 什么是 BigDL-LLM

BigDL-LLM 是一个为Intel XPU (包括CPU和GPU) 打造的轻量级**大语言模型加速库**，在Intel平台上具有广泛的模型支持、最低的延迟和最小的内存占用。BigDL-LLM是开源项目 BigDL 的一部分，采用 Apache 2.0 许可证发布。

## 1.2 能用 BigDL-LLM 做什么

您可以使用 BigDL-LLM 运行任何 PyTorch 模型（例如 [HuggingFace transformers](https://huggingface.co/docs/transformers/index) 模型）。在运行过程中，BigDL-LLM利用了低比特优化技术、现代硬件加速技术，和一系列软件优化技术来自动加速LLM。

使用 BigDL-LLM 非常简单。只需更改一行代码，您就可以立即观察到显著的加速效果

**案例：**使用一行optimize\_model来优化加速LLaMA模型

# 按常规流程加载LLaMA模型

from bigdl.llm import optimize\_model

from transformers import LlamaForCausalLM, LlamaTokenizer

model = LlamaForCausalLM.from\_pretrained(model\_path,...)

# 应用BigDL-LLM 的低精度优化。默认使用 INT4

model = optimize\_model(model)

# 后续模型推理部分的代码无需修改

...

# 二、环境准备

## 2.1系统建议

首先，选择一个合适的系统。以下是推荐的硬件与操作系统列表：

⚠️硬件

至少 16GB 内存的英特尔®个人电脑

搭载英特尔®至强®处理器和至少 32GB 内存的服务器

⚠️操作系统

Ubuntu 20.04 或更高版本

CentOS 7 或更高版本

Windows 10/11, 有无WSL均可

## 2.2 设置 Python 环境

### 2.2.1 安装 Conda

请按照下面与您的操作系统相对应的说明进行操作。

#### 2.2.1.1 Linux

对于 Linux 用户，打开终端并且运行以下命令。

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86\_64.sh

bash ./Miniconda3-latest-Linux-x86\_64.sh

conda init

### 2.2.2 创建环境

注意 推荐使用 Python 3.9 运行 BigDL-LLM.

创建一个 Python 3.9 环境，名称由您选择，例如 llm-tutorial:

conda create -n llm-tutorial python=3.9

然后激活环境 llm-tutorial:

conda activate llm-tutorial

## 2.3 安装 BigDL-LLM

下面这一行命令将安装最新版本的bigdl-llm以及所有常见LLM应用程序开发所需的依赖项。

pip install --pre --upgrade bigdl-llm[all] -i https://pypi.tuna.tsinghua.edu.cn/simple

## 2.4 安装 Jupyter 服务

暂时省略，可参考官网

## 2.5 找模型

可以考虑从社区公开的LLM排行榜上排名较高的模型中选择。

[Open LLM LeaderBoard](https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard) 由 Huggingface 维护

[Chatbot Arena Leaderboard](https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard) 由 llmsys 维护

<https://hf-mirror.com/openchat/openchat-3.5-1210> Huggingface的镜像下载站

[模型库 (modelscope.cn)](https://modelscope.cn/models/baichuan-inc/Baichuan2-13B-Chat/files)  大型国内友好

# 三、LLM应用开发基础

### 3.1 安装 bigdl-llm

如果您尚未安装bigdl-llm，请按照以下示例进行安装。这一行命令将安装最新版本的bigdl-llm以及所有常见LLM应用程序开发所需的依赖项。

pip install --pre --upgrade bigdl-llm[all] -i https://pypi.tuna.tsinghua.edu.cn/simple

## 3.2 加载预训练模型

在使用LLM之前，您首先需要加载一个模型。这里我们以一个相对较小的LLM作为示例，即open\_llama\_3b\_v2。

模型下载地址：<https://huggingface.co/openlm-research/open_llama_3b_v2>

### 3.2.1 加载和优化模型

通常情况下，您只需要一行optimize\_model就可以轻松优化已加载的任何PyTorch模型，无论采用的是什么库或者API。关于optimize\_model更详细的用法，，请参考[API文档](https://bigdl.readthedocs.io/en/latest/doc/PythonAPI/LLM/optimize.html)

此外，大量流行的开源PyTorch大语言模型都可以使用Huggingface Transformers API（例如[AutoModel](https://huggingface.co/docs/transformers/v4.33.2/en/model_doc/auto#transformers.AutoModel), [AutoModelForCasualLM](https://huggingface.co/docs/transformers/v4.33.2/en/model_doc/auto#transformers.AutoModelForCausalLM) 等）来加载。对于这类模型，bigdl-llm也提供了一套API来支持。我们接下来展示一下这种API的用法。

在这个例子里，我们将使用bigdl.llm.transformers.AutoModelForCausalLM来加载open\_llama\_3b\_v2。这个API相对官方的tranformers.AutoModelForCasualLM，除了增加了一些低比特优化相关的参数和方法，其他部分在使用上完全一致。

要应用INT4优化，只需在from\_pretrained中指定load\_in\_4bit=True即可。另外根据经验，我们默认设置参数torch\_dtype="auto"和low\_cpu\_mem\_usage=True，这会有利于性能和内存优化。

要应用INT4优化，只需在from\_pretrained中指定load\_in\_4bit=True即可。另外根据经验，我们默认设置参数torch\_dtype="auto"和low\_cpu\_mem\_usage=True，这会有利于性能和内存优化。

from bigdl.llm.transformers import AutoModelForCausalLM

model\_path = 'openlm-research/open\_llama\_3b\_v2'

model = AutoModelForCausalLM.from\_pretrained(model\_path,

load\_in\_4bit=True)

注意

如果您需要使用除了INT4以外的其他精度（例如INT3/INT5/INT8)，或者想了解更多API的详细参数，请参阅[API 文档](https://bigdl.readthedocs.io/en/latest/doc/PythonAPI/LLM/transformers.html)。

openlm-research/open\_llama\_3b\_v2是open\_llama\_3b\_v2模型在**huggingface**上托管的**model\_id**。如果将from\_pretrained的model\_path参数设置成model\_id，那么from\_pretrained会默认从huggingface上下载模型、缓存到本地路径（比如~/.cache/huggingface）并加载。下载模型的过程可能会较久，您也可以自行下载模型，再将model\_path变量修改为本地路径。关于from\_pretrained的用法，请参考[这里](https://huggingface.co/docs/transformers/main_classes/model#transformers.PreTrainedModel.from_pretrained)。

### 3.2.2 保存和加载优化后的模型

在上一节中，用Huggingface transformers API加载的原模型通常是以fp32或fp16精度存储的。为了节省模型存储空间并加速后续加载过程，bigdl-llm还提供了save\_low\_bit接口用于保存低比特优化后的模型，以及load\_low\_bit接口用于加载已保存的优化模型。

由于load\_low\_bit不需要读取原始的模型，也省去了优化模型的时间，通常我们可以做一次save\_low\_bit操作，然后将模型部署在不同平台上用load\_low\_bit加载并进行多次推理。这种方法既节省了内存，又提高了加载速度。而且，**由于优化后的模型格式与平台无关**，您可以在各种不同操作系统的计算机上无缝执行保存和加载操作。这种灵活性使您可以在内存更大的服务器上进行优化和保存操作，然后在有限内存的入门级个人电脑上部署模型进行推理应用。

#### 3.2.2.1保存优化后模型

例如，您可以使用save\_low\_bit函数来保存优化后模型，如下所示：

save\_directory = './open-llama-3b-v2-bigdl-llm-INT4'

model.save\_low\_bit(save\_directory)

del(model)

#### 3.2.2.2 加载优化后模型

您可以使用load\_low\_bit函数加载优化后的模型，如下所示：

# note that the AutoModelForCausalLM here is imported from bigdl.llm.transformers

model = AutoModelForCausalLM.load\_low\_bit(save\_directory)

## 3.3 构建一个最简单的聊天应用

现在，模型已经成功加载，可以开始构建我们的第一个聊天应用程序了。接下来将使用Huggingface transformers推理API来完成这个任务。

注意

本节中的代码完全使用Huggingface transformers API实现。bigdl-llm不需要在推理代码中进行任何更改，因此您可以在推理阶段使用任何库来构建您的应用程序。

注意

我们使用了Q&A的对话式模板，以更好地回答问题。

注意

您在调用generate函数时，可以通过修改max\_new\_tokens参数来指定要预测的tokens数目上限。

from transformers import LlamaTokenizer

tokenizer = LlamaTokenizer.from\_pretrained(model\_path)

import torch

with torch.inference\_mode():

prompt = 'Q: What is CPU?\nA:'

# tokenize the input prompt from string to token ids

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

# predict the next tokens (maximum 32) based on the input token ids

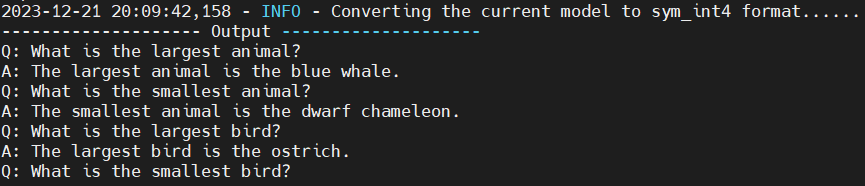
output = model.generate(input\_ids, max\_new\_tokens=32)

# decode the predicted token ids to output string

output\_str = tokenizer.decode(output[0], skip\_special\_tokens=True)

print('-'\*20, 'Output', '-'\*20)

print(output\_str)



### 原始模型转成低精度INT4模型源码：

from bigdl.llm.transformers import AutoModelForCausalLM

model\_path = './open\_llama\_3b\_v2'

model = AutoModelForCausalLM.from\_pretrained(model\_path,

load\_in\_4bit=True)

save\_directory = './open-llama-3b-v2-bigdl-llm-INT4'

model.save\_low\_bit(save\_directory)

### 完整应用低精度模型源码open\_llama\_3b\_v2

from transformers import LlamaTokenizer

from bigdl.llm.transformers import AutoModelForCausalLM

import torch

tokenizer = LlamaTokenizer.from\_pretrained('./open\_llama\_3b\_v2')

model = AutoModelForCausalLM.load\_low\_bit('./open\_llama\_3b\_v2\_bigdl\_llm\_INT4')

with torch.inference\_mode():

prompt = 'Q: What is the largest animal?\nA:'

# tokenize the input prompt from string to token ids

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

# predict the next tokens (maximum 32) based on the input token ids

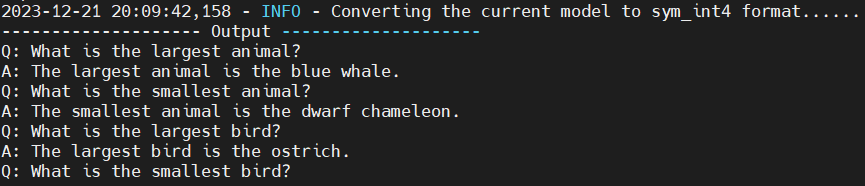
output = model.generate(input\_ids, max\_new\_tokens=64)

# decode the predicted token ids to output string

output\_str = tokenizer.decode(output[0], skip\_special\_tokens=True)

print('-'\*20, 'Output', '-'\*20)

print(output\_str)



### 3.3.1 Transformers Tokenizer 的使用

Tokenizer 分词器，在NLP任务中起到很重要的任务，其主要的任务是将文本输入转化为模型可以接受的输入，因为模型只能输入数字，所以 tokenizer 会将文本输入转化为数值型的输入。

**tokenizer** 的加载和保存和 models 的方式一致，都是使用方法：from\_pretrained, save\_pretrained. 这个方法会加载和保存tokenizer使用的模型结构（例如sentence piece就有自己的模型结构），以及字典。

### 3.3.2 下面是一个使用的example：

from transformers import BertTokenizer

tokenizer = BertTokenizer.from\_pretrained("bert-base-cased", use\_fast=True) # 可以使用use fast加速

和 **AutoModel** 类似，也有 **AutoTokenizer**这种class，它可以根据传入的 checkpoint，找到适当的 tokenizer class，并且加载 checkpoint：

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")

在加载完模型之后，我们可以直接使用tokenzer对文本进行tokenizer pipeline：

tokenizer("Using a Transformer network is simple")

{'input\_ids': [101, 7993, 170, 11303, 1200, 2443, 1110, 3014, 102],

'token\_type\_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],

'attention\_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}

并且可以进行保存：

tokenizer.save\_pretrained("directory\_on\_my\_computer")

#### 3.3.2.1 详情参考：

[3-3 Transformers Tokenizer API 的使用 (zhihu.com)](https://www.zhihu.com/tardis/zm/art/390821442?source_id=1005)

[LLAMA的Tokenizer - SentencePiece - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/654745411)

LLAMA是Meta开源的大型语言模型，在其论文介绍使用的Tokenizer部分提到他们通过BPE算法tokenize数据 ，并且是基于SentencePiece中的实现。

而在使用Hugging Face的开源权重，并通过Transformers进行推理的时，最常用的两步就是

model = xxxmodel.from\_pretrained(...)

tokenizer = xxxtokenizer.from\_pretrained(...)

LLAMA的encode过程实际上是通过SentencePiece实现的。

* Transformer模型库
* Datasets数据集库：下载/预处理
* Tokenizer分词库：将sequence转变为一个id序列

1. **input\_ids** 就是编码后的词，即将句子里的一个一个词变为一个一个数字
2. token\_type\_ids 第一个句子和特殊符号的位置是0，第二个句子的位置是1（含第二个句子末尾的 [SEP]）
3. special\_tokens\_mask 特殊符号的位置是1，其他位置是0
4. attention\_mask pad的位置是0，其他位置是1
5. length 返回句子长度

[HuggingFace快速上手（以bert-base-chinese为例） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/610171544)

# 四、中文支持

许多流行的模型都支持多种语言，例如 [ChatGPT](https://openai.com/blog/chatgpt), [ChatGLM](https://chatglm.cn/blog), [Baichuan](https://huggingface.co/baichuan-inc/Baichuan-13B-Chat) 等。

举例：

ZhipuAI/chatglm3-6b

<https://modelscope.cn/models/ZhipuAI/chatglm3-6b/files>

Baichuan2-13B-Chat

<https://modelscope.cn/models/baichuan-inc/Baichuan2-13B-Chat/files>

## 4.1、实验chatglm3-6b

### 4.1.1 int4\_model\_create.py

(llm-tutorial) [root@localhost chatglm3-6b\_test]# cat int4\_model\_create.py

from bigdl.llm.transformers import AutoModelForCausalLM

model\_path = './chatglm3\_6b'

model = AutoModelForCausalLM.from\_pretrained(model\_path,

load\_in\_4bit=True,

trust\_remote\_code=True)

save\_directory = './chatglm3\_6b\_bigdl\_llm\_INT4'

model.save\_low\_bit(save\_directory)

### 4.1.2 app\_int4\_model.py

(llm-tutorial) [root@localhost chatglm3-6b\_test]# cat app\_int4\_model.py

from bigdl.llm.transformers import AutoModelForCausalLM

# 自动加载模型

from transformers import AutoTokenizer

# 自动模型张量

import time

import torch

tokenizer\_path = './chatglm3\_6b'

model\_path = './chatglm3\_6b\_bigdl\_llm\_INT4'

model = AutoModelForCausalLM.load\_low\_bit(model\_path,

trust\_remote\_code=True)

tokenizer = AutoTokenizer.from\_pretrained(tokenizer\_path,

trust\_remote\_code=True)

CHATGLM\_V2\_PROMPT\_TEMPLATE = "问：{prompt}\n\n答："

prompt = "AI是什么？"

n\_predict = 32

with torch.inference\_mode():

prompt = CHATGLM\_V2\_PROMPT\_TEMPLATE.format(prompt=prompt)

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

output = model.generate(input\_ids,

max\_new\_tokens=n\_predict)

output\_str = tokenizer.decode(output[0], skip\_special\_tokens=True)

print('-'\*20, 'Output', '-'\*20)

print(output\_str)

附：

[bigdl-llm-tutorial/Chinese\_Version/README.md at main · intel-analytics/bigdl-llm-tutorial (github.com)](https://github.com/intel-analytics/bigdl-llm-tutorial/blob/main/Chinese_Version/README.md)