gpt-j

Local GPT-J inference on your computer using C/C++

- 使用C/C++在本地进行GPT-J的推理

No video card required. You just need to have 16 GB of RAM.

- 显卡不是必须的，仅仅需要16GB的RAM

Motivation（动机）

The GPT-J 6B model is the open-source alternative to OpenAI's GPT-3. It's basically a neural network that allows you to generate coherent, human-like text given a certain context (prompt).

- GPT-J 6B 模型是 OpenAI 的 GPT-3 的开源替代品。它基本上是一个神经网络，可以让你在给定特定上下文（提示）的情况下生成连贯的、类似人类的文本。

The GPT-J model is quite big - the compact version of the model uses 16-bit floating point representation of the weights and is still 12 GB big. This means that in order to run inference on your computer, you would need to have a video card with at least 12 GB of video RAM. Alternatively, you can try to run the python implementations on the CPU, but that would probably not be very efficient as they are primarily optimized for running on a GPU (or at least this is my guess - I don't have much experience with python).

- GPT-J 模型相当大 - 该模型的紧凑版本使用权重的 16 位浮点表示 并且仍然有 12 GB 大。这意味着为了在您的计算机上运行推理，您需要拥有至少具有 12 GB 视频 RAM 的视频卡。或者，您可以尝试在 CPU 上运行 Python 实现，但这可能不是很高效，因为它们主要针对在 GPU 上运行进行了优化（或者至少这是我的猜测 - 我对 Python 没有太多经验）。

I wanted to try and run the model on my MacBook, so I decided to implement the model inference from scratch using my own custom build tensor library. The tensor library (called ggml, written in C) is in early development stage, but it already allows me to run the GPT-J model.

- 我想尝试在我的 MacBook 上运行该模型，所以我决定使用我自己的 自定义构建张量库从头开始实现模型推理。该张量库（称为 ggml，用 C 语言编写）处于早期开发阶段，但它已经允许我运行 GPT-J 模型。

On my 32GB MacBook M1 Pro, I achieve an inference speed of about `125 ms/token` or about ~6 words per second (1 word typically consists of 1 or 2 tokens).

- 在我的 32GB MacBook M1 Pro 上，我实现了大约 `125 ms/token` 或大约每秒 ~6 个字的推理速度（1 个字通常由 1 或 2 个 token 组成）。

Here is a sample run with prompt `int main(int argc, char ** argv) {`:

```
$ time ./bin/gpt-j -p "int main(int argc, char ** argv) {"

gptj_model_load: loading model from 'models/gpt-j-6B/ggml-model.bin' - please wait
...
```

```
gptj_model_load: n_vocab = 50400
gptj_model_load: n_ctx   = 2048
gptj_model_load: n_embd  = 4096
gptj_model_load: n_head  = 16
gptj_model_load: n_layer = 28
gptj_model_load: n_rot   = 64
gptj_model_load: f16     = 1
gptj_model_load: ggml ctx size = 13334.86 MB
gptj_model_load: memory_size =  1792.00 MB, n_mem = 57344
gptj_model_load: ..................................... done
gptj_model_load: model size = 11542.79 MB / num tensors = 285
main: number of tokens in prompt = 13

int main(int argc, char ** argv) {
    (void)argc;
    (void)argv;

    {
        struct sockaddr_in addr;
        int addrlen;
        char * ip = "192.168.1.4";
        int i;

        if ( (addrlen = sizeof(addr)) == -1 )
            return -1;

        for (i = 0; i < 10; ++i) {
            addr.sin_family = AF_INET;
            addr.sin_addr.s_addr = inet_addr(ip);

main: mem per token = 16430420 bytes
main:     load time =  6211.48 ms
main:   sample time =    13.74 ms
main:  predict time = 26420.34 ms / 124.62 ms per token
main:    total time = 33035.37 ms

real    0m33.171s
user    3m32.269s
sys      0m3.686s

$
```

It took ~6.2 seconds to load the model to memory. After that, it took ~26.4 seconds to generate 200 tokens of what looks like to be the beginning of a networking program in C. Pretty cool!

- 将模型加载到内存大约需要 6.2 秒。之后，大约需要 26.4 秒来生成 200 个 token，看起来像是 C 语言网络程序的开头。太酷了！

Here is another run, just for fun:

```
time ./bin/gpt-j -n 500 -t 8 -p "Ask HN: Inherited the worst code and tech team I
have ever seen. How to fix it?
"

gptj_model_load: loading model from 'models/gpt-j-6B/ggml-model.bin' - please wait
...
gptj_model_load: n_vocab = 50400
gptj_model_load: n_ctx   = 2048
gptj_model_load: n_embd  = 4096
gptj_model_load: n_head  = 16
gptj_model_load: n_layer = 28
gptj_model_load: n_rot   = 64
gptj_model_load: f16     = 1
gptj_model_load: ggml ctx size = 13334.86 MB
gptj_model_load: memory_size =  1792.00 MB, n_mem = 57344
gptj_model_load: ..................................... done
gptj_model_load: model size = 11542.79 MB / num tensors = 285
main: number of tokens in prompt = 24

Ask HN: Inherited the worst code and tech team I have ever seen. How to fix it?

I've inherited a team with some very strange and un-documented practices, one of
them is that they use an old custom
application with a very slow tech stack written in Python that the team doesn't
want to touch but also doesn't want to
throw away as it has some "legacy" code in it.

The problem is, the tech stack is very very slow.

They have a single web server on a VM that is slow.
The server is a little bit busy (not very busy though) and they have a lot of
processes (30+ that are constantly being
spawned by the application)
They have an application that is single threaded and was written in Python and the
team don't want to touch this, and
the application is very slow.

My task as a new member of the team is to fix this.

I'm a senior dev on the team (3 years on the project) and have been told that I
will take the lead on this task. I know
next to nothing about Python. So here is what I have so far.

What I have done is I've been trying to debug the processes with the "ps" command.
This way I can see what is running
and where. From what I see, the application spawns 10 processes a minute and some
of them are used for nothing.

I have also started to look for the code. The application source is not in GitHub
or any other repository, it is only on
our internal GitLab.

What I've found so far:
```

```
The application uses a custom SQLAlchemy implementation to interact with the data.
I've looked at the source, it looks
like an object cache or something like that. But from what I've seen, the cache
gets full every 20 minutes and then gets
cleared with a special command.

Another strange thing is that the application creates a file for every entry in
the database (even if the entry already
exists). I've looked at the file to see if it contains something, but it seems to
be a JSON file with lots of records.

The other strange thing is that I can only find the database tables in the GitLab
repository and not the code. So I
can't really understand how the application is supposed to interact with the
database.

I also found a "log" directory, but the code is encrypted with AES. From what I've
found, it is in

main: mem per token = 16430420 bytes
main:     load time =  3900.10 ms
main:   sample time =    32.58 ms
main:  predict time = 68049.91 ms / 130.11 ms per token
main:    total time = 73020.05 ms

real    1m13.156s
user    9m1.328s
sys.    0m7.103s
```

Implementation details（实现的细节）

The high level implementation of the model is contained in the main.cpp file. The core computations are performed by the ggml library.

- 该模型的高级实现包含在 main.cpp 文件中。核心计算由 ggml 库执行。

Matrix multiplication（矩阵乘法）

The most performance critical part of the implementation is of course the matrix multiplication routine. 99% of the time is spent here, so it was important to optimize this as much as possible.

- 实现过程中对性能影响最大的部分当然是矩阵乘法例程。99% 的时间都花在这里，因此尽可能地优化这一点非常重要。

On Arm64, I utilize the 128-bit NEON intrinsics for 16-bit floating point operations:

- 在 Arm64 上，我利用 128 位 NEON 内部函数进行 16 位浮点运算：

https://github.com/ggerganov/ggml/blob/fb558f78d905f85c54813602649ddd628ffe0f3a/src/ggml.c#L187-L243

These instructions allow each core to operate simultaneously on 64 16-bit floats. I'm no expert in SIMD, but after quite some trials this was the most efficient code for dot product of a row and column that I could come up with. Combined with the parallel computation on 8 CPU threads, I believe I'm close to the maximum performance that one could possibly get on the M1 CPU. Still, I'm curious to know if there is a more efficient way to implement this.

- 这些指令允许每个核心同时对 64 个 16 位浮点数进行操作。我不是 SIMD 方面的专家，但经过多次尝试，这是我能想到的计算行和列点积的最高效代码。结合 8 个 CPU 线程上的并行计算，我相信我已经接近 M1 CPU 上可能获得的最高性能。不过，我很好奇是否有更有效的方法来实现这一点。

Attempt to use the M1 GPU（尝试使用M1 GPU）

One interesting property of the GPT-J transformer architecture is that it allows you to perform part of the inference in parallel - i.e. the Feed-forward network can be computed in parallel to the Self-attention layer:

- GPT-J Transformer 架构的一个有趣特性是，它允许你并行执行部分推理 - 即前馈网络可以与自注意力层并行计算：

https://github.com/ggerganov/ggml/blob/fb558f78d905f85c54813602649ddd628ffe0f3a/examples/gpt-j/main.cpp#L507-L531

So I thought why not try and bring in the M1 GPU to compute half of the neural network in parallel to the CPU and potentially gain some extra performance. Thanks to the M1's shared memory model, it was relatively easy to offload part of the computation to the GPU using Apple's Metal Performance Shaders. The GPU shares the host memory, so there is no need to copy the data back and forth as you would normally do with Cuda or OpenCL. The weight matrices are directly available to be used by the GPU.

- 所以我想为什么不尝试引入 M1 GPU 来与 CPU 并行计算一半的神经网络并潜在地获得一些额外的性能。得益于 M1 的共享内存模型，使用 Apple 的 Metal Performance Shaders 将部分计算卸载到 GPU 上相对容易。GPU 共享主机内存，因此无需像使用 Cuda 或 OpenCL 那样来回复制数据。权重矩阵可直接供 GPU 使用。

However, to my surprise, using MPS together with the CPU did not lead to any performance improvement at all. My conclusion was that the 8-thread NEON CPU computation is already saturating the memory bandwidth of the M1 and since the CPU and the GPU on the MacBook are sharing that bandwidth, it does not help to offload the computation to the GPU.

- 然而，令我惊讶的是，将 MPS 与 CPU 一起使用并没有带来任何性能提升。我的结论是，8 线程 NEON CPU 计算已经饱和了 M1 的内存带宽，并且由于 MacBook 上的 CPU 和 GPU 共享该带宽，因此将计算卸载到 GPU 上没有帮助。

Another observation was that the MPS GPU matrix multiplication using 16-bit floats had the same performance as the 8-thread NEON CPU implementation. Again, I explain this with a saturated memory channel. But of course, my explanation could be totally wrong and somehow the implementation wasn't utilizing the resources correctly.

- 另一个观察结果是，使用 16 位浮点数的 MPS GPU 矩阵乘法具有与 8 线程 NEON CPU 实现相同的性能。再次，我用饱和的内存通道来解释这一点。但当然，我的解释可能完全错误，并且实现在某种程度上没有正确利用资源。

In the end, I decided to not use MPS or the GPU all together.

- 最后，我决定完全不使用 MPS 或 GPU。

Zero memory allocations（零内存分配）

Another property of my implementation is that it does not perform any memory allocations once the model is loaded into memory. All required memory is allocated at the start of the program with a single `malloc` (technically 2 calls, but that is not important).

- 我的实现的另一个特性是，一旦模型加载到内存中，它就不会执行任何内存分配。所有所需的内存都在程序启动时通过单个"malloc"分配（技术上是 2 次调用，但这并不重要）。

Usage

If you want to give this a try and you are on Linux or Mac OS, simply follow these instructions:

- 如果您想尝试一下并且使用的是 Linux 或 Mac OS，请按照以下说明操作：

```
# Download the ggml-compatible GPT-J 6B model (requires 12GB disk space)
../examples/gpt-j/download-ggml-model.sh 6B

# Run the inference (requires 16GB of CPU RAM)
./bin/gpt-j -m models/gpt-j-6B/ggml-model.bin -p "This is an example"

# Input prompt through pipe and run the inference.
echo "This is an example" > prompt.txt
cat prompt.txt | ./bin/gpt-j -m models/gpt-j-6B/ggml-model.bin
```

To run the `gpt-j` tool, you need the 12GB `ggml-model.bin` file which contains the GPT-J model in ggml compatible format. In the instructions above, the binary file is downloaded from my repository on Hugging Face using the download-ggml-model.sh script. You can also, download the file manually from this link:

https://huggingface.co/ggerganov/ggml/tree/main

Alternatively, if you don't want to download the 12GB ggml model file, you can perform the conversion yourself using python.

- 或者，如果您不想下载 12GB 的 ggml 模型文件，您可以使用 python 自行执行转换。

First, you need to download the full GPT-J model from here: https://huggingface.co/EleutherAI/gpt-j-6B

- 首先，你需要从这里下载完整的 GPT-J 模型：https://huggingface.co/EleutherAI/gpt-j-6B

Note that the full model is quite big - about 72 GB. After you download it, you need to convert it to ggml format using the convert-h5-to-ggml.py script. This will generate the `ggml-model.bin` file, which you can then use with the `gpt-j` program.

- 请注意，完整模型相当大 - 大约 72 GB。下载后，您需要使用convert-h5-to-ggml.py 脚本将其转换为 ggml 格式。这将生成 `ggml-model.bin` 文件，然后您可以将其与 `gpt-j` 程序一起使用。

GPT-2

I also implemented a tool for CPU inference using the smaller GPT-2 models. They have worse quality compared to GPT-J, but are much faster to execute.

For example, the Small GPT-2 model is only 240 MB big and the inference speed on my MacBook is about 200 tokens/sec.

For more details, checkout the GPT-2 example here: gpt-2

- 我还使用较小的 GPT-2 模型实现了一个 CPU 推理工具。与 GPT-J 相比，它们的质量较差，但执行速度要快得多。
- 例如，小型 GPT-2 模型只有 240 MB，在我的 MacBook 上的推理速度约为 200 个 token/秒。
- 有关更多详细信息，请查看此处的 GPT-2 示例：gpt-2