OpenVINO™ ツールキットを使用して高速で小さな LLM をデプロイする理由と方法

この記事は、Medium に公開されている「Why and How to Use OpenVINO™ Toolkit to Deploy Faster, Smaller LLMs」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

大規模言語モデル (LLM) は、会話型 AI、理解、翻訳の分野に画期的なアプリケーションをもたらしています。企業やユーザーが LLM の可能性を模索する中、OpenVINO™ ツールキットは、効率良く、高速かつ柔軟に LLM を最適化してデプロイするためのソリューションとして注目されています。OpenVINO™ を使用して LLM を圧縮し、AI アシスタント・アプリケーションに統合して、パフォーマンスに優れたアプリケーションをエッジデバイスやクラウドにデプロイできます。

この記事では、OpenVINO™ が対応している LLM の利点と、OpenVINO™ を使用して LLM をロードおよびデプロイする方法を説明します。この記事は、OpenVINO™ を使用した LLM の最適化に関するソリューション・ホワイト・ペーパー (英語) から抜粋して要約したものです。

OpenVINO™ とは

OpenVINO™ は、LLM をデプロイするための効率の良いランタイム環境を提供します。ほかのフレームワークに比べて次のような利点があります。

- **デプロイのサイズが小さい:** OpenVINO™ は自己完結型のパッケージのため、必要な依存関係が Hugging Face や PyTorch* (ギガバイト単位) よりも少なくなります (数百メガバイト単位)。
- **速度:** OpenVINO™ は LLM 向けに最適化された推論パフォーマンスを提供します。また、継続的にアップデートを提供しています。パフォーマンスはサードパーティーのソリューションに匹敵するか上回っていて (英語)、C/C++ および Python* API も提供します。
- **インテルの公式サポート:** OpenVINO™ は、インテルが配布する公式 AI フレームワークとして、パッチ、アップグレード、機能のアップデートが提供されます。インテルのフィールド・アプリケーション・エンジニアの Q&A にアクセスすることもできます。
- **柔軟性:** OpenVINO™ は、LLM からコンピューター・ビジョン、画像生成、音声変換、データ分類など、マルチモーダル・アプリケーションの開発とデプロイをサポートする、さまざまなモデルとアーキテクチャーをサポートしています。
- **ハードウェア・サポート:** OpenVINO™ は、CPU、統合 GPU、ディスクリート GPU を含む幅広い x86/x64 および ARM ベースのハードウェア・ターゲットをサポートしていて、高性能サーバーやコンパクトなエッジデバイスにデプロイできます。

OpenVINO™ を使用して LLM をデプロイする方法

OpenVINO™ を使用して LLM を最適化およびデプロイするオプションは 2 つあります。どちらの場合も、OpenVINO™ ランタイムを推論のバックエンドとして使用し、OpenVINO™ ツールをモデルの最適化に使用します。主な違いは、使いやすさ、フットプリントのサイズ、カスタマイズ性です。

1. **Hugging Face:** Optimum Intel (英語) エクステンションを利用して、OpenVINO™ を Hugging Face Transformers API のバックエンドとして使用します。この API は習得が簡単で、インターフェイスも単純ですが、依存関係が多いため、LLM が重くなります。複雑さの多くは抽象化レイヤーの下に隠されているため、カスタマイズの選択肢は少なくなります。
2. **ネイティブ OpenVINO™:** カスタム・パイプライン・コードで OpenVINO™ ネイティブ API (C++ および Python*) を使用します。依存関係が少なく、アプリケーションのフットプリントが最小限に抑えられますが、テキスト生成ループ、トークン化、スケジューラー関数の明示的な実装が必要になります。習得は簡単ではありませんが、カスタマイズの選択肢は多くなります。

ここでは、両方のオプションを説明します。

要件

OpenVINO™ を使用するには、OpenVINO™ のインストール手順 (英語) に従って、OpenVINO™ 向けの Python* 仮想環境を設定します。環境を作成して有効にしたら、次のコマンドを実行して、Optimum Intel、OpenVINO™、NNCF および依存関係を Python* 環境にインストールします。

```
pip install optimum[openvino]
```

ネイティブ OpenVINO™ 環境に LLM をデプロイする場合は、トークン化用の OpenVINO™ トークナイザー (英語) もインストールする必要があります。Linux*、macOS*、Windows* オペレーティング・システムをサポートしています。サポートしているトークナイザーのタイプのリストは、OpenVINO™ トークナイザー・ドキュメントの「サポートしているトークナイザーのタイプ (英語)」セクションを参照してください。

このソリューション・ホワイト・ペーパーの「**依存関係のインストール**」セクションで設定されたものと同じ Python* 仮想環境で、次のコマンドを実行して OpenVINO™ トークナイザーをインストールします。

```
pip install openvino-tokenizers[transformers]
```

Hugging Face と Optimum Intel を使用した推論

AI テキスト生成アプリケーションでは、LLM 推論は 4 つのステップで構成され、Hugging Face を使用するかネイティブ OpenVINO™ API を使用するかにより大きく異なります。

1. モデルをロードする
2. 入力テキストをトークン化する
3. 推論ループを実行する
4. 出力トークンを処理する

Hugging Face LLM を OpenVINO™ にロードする

OpenVINO™ で LLM を使用する最も簡単な方法は、Optimum Intel (英語) を使用して Hugging Face Hub (英語) からモデルをロードすることです。Optimum Intel を使用してロードしたモデルは、Hugging Face Transformers API との互換性を保ったまま、OpenVINO™ 向けに最適化されています。OVModelForCausalLM クラスはモデル名を受け取り、Hugging Face からダウンロードして、メモリー内のオブジェクトとして初期化します。

Hugging Face からモデルを初期化するには、以下のコード例の OVModelForCausalLM.from_pretrained メソッドを使用します。パラメーター export=True を設定すると、モデルはオンザフライで OpenVINO™ 中間表現 (IR) 形式に変換されます。

```
from optimum.intel import OVModelForCausalLM
model_id = "HuggingFaceH4/zephyr-7b-beta"
model = OVModelForCausalLM.from_pretrained(model_id, export=True)
```

モデルを保存してロードする

Optimum Intel を使用してモデルを IR 形式に変換したら、保存してエクスポートし、将来のセッションやデプロイ環境で使用できます。変換プロセスには時間がかかるため、モデルを IR 形式に一度変換して保存し、後で圧縮したモデルをロードすると、最初の推論までの時間を短縮できます。

モデルとトークナイザーを保存してエクスポートするには、model.save_pretrained("your-model-name") と tokenizer.save_pretrained("your-model-name") を使用します。

```
# Save model for faster loading later
model.save_pretrained("zephyr-7b-beta-ov")
tokenizer.save_pretrained("zephyr-7b-beta-ov")
```

モデルは OpenVINO™ IR 形式 (openvino_model.bin、openvino_model.xml) でエクスポートされ、指定したディレクトリーの新しいフォルダーに保存されます。トークナイザーも同様に保存されます。将来のセッションでモデルとトークナイザーをロードするには、OVModelForCausalLM.from_pretrained("your-model-name") と AutoTokenizer.from_pretrained("your-model-name") を使用します。

```
# Load a saved model
model = OVModelForCausalLM.from_pretrained("zephyr-7b-beta-ov")
tokenizer = AutoTokenizer.from_pretrained("zephyr-7b-beta-ov")
```

モデルを初期化する

Hugging Face の高レベルの Transformers API (英語) は、モデルを初期化して推論を実行するための単純なオプションを提供します。Optimum Intel エクステンションでラップされていて、LLM を OpenVINO™ IR 形式に変換し、OpenVINO™ ランタイムをバックエンドとして設定します。

Hugging Face Transformers と Optimum Intel を使用して、単純なテキスト生成パイプラインを設定して実行する簡単な例を次に示します。

```
Python
from optimum.intel import OVModelForCausalLM
# new imports for inference
from transformers import AutoTokenizer, pipeline

# load the model
model_id = "meta-llama/Llama-2-7b-chat-hf"
model = OVModelForCausalLM.from_pretrained(model_id, export=True)

# inference
tokenizer = AutoTokenizer.from_pretrained(model_id)
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)
prompt = "What is OpenVINO?"
results = pipe(prompt)
```

この例では、3 つのクラスとメソッドが使用されています。

- Optimum Intel の OVModelForCausalLM.from_pretrained: Hugging Face から LLM をロードし、OpenVINO™ IR 形式に変換して、OpenVINO™ を推論バックエンドとして使用してターゲットデバイスでコンパイルします。
- Hugging Face Transformers の AutoTokenizer: LLM のテキスト・トークナイザーを初期化します。
- Hugging Face Transformers の pipeline: 入力のトークン化、推論ループの実行、出力の処理を含む、テキスト生成の大部分を処理します。

これらのクラスは、テキスト生成を設定するための単純なインターフェイスを提供します。各クラスまたはメソッドには、LLM をさらに構成するために使用できるほかのパラメーターもあります。Transformers API には、model.generate() メソッドなど、推論パラメーターを細かく制御できるほかの機能もあります。詳細は、Hugging Face Transformers のドキュメント (英語) を参照してください。

Hugging Face で LLM をコンパイルするデバイスを選択する

LLM をコンパイルするデバイス (CPU、iGPU、GPU など) を選択するオプションは 2 つあります。

1. .from_pretrained() 呼び出しでデバイスのパラメーターを指定します。例えば、GPU でモデルを実行するには、OVModelForCausalLM.from_pretrained (model_id, export=True, device="GPU.0") を使用します。詳細は、デバイスクエリーのドキュメント (英語) を参照してください。
2. モデルをロードした後に model.to メソッドを使用して、ターゲットデバイスの名前を渡します。例えば、GPU でモデルを実行するには、model.to("GPU.0") を使用します。

Hugging Face API は、テキスト生成を実装するコードを大幅に簡素化できますが、C/C++ で実装できないという欠点があります。対照的に、ネイティブ OpenVINO™ API は、C/C++ でソリューションを構築できます。

ネイティブ OpenVINO™ Python* API を使用した推論

ネイティブ OpenVINO™ API を使用して LLM で推論を実行することもできます。テキスト生成 LLM の推論パイプラインは、次の 4 つのステップで設定できます。

1. モデルを読み取ってコンパイルする
2. テキストをトークン化してモデルの入力を設定する
3. トークン生成ループを実行する
4. 出力を逆トークン化する

このセクションでは、ネイティブ OpenVINO™ Python* API を使用して各ステップを実装する方法を示すコード例を提供します。これらのコード例は、LLM のメモリー効率を高めるためにステートフル・モデル手法を実装します。この手法では、モデルのコンテキスト — 内部の状態 (KV キャッシュ) — が推論の複数の反復間で共有されます。特定のテキストシーケンスに属する KV キャッシュは、生成ループ中にモデル内に累積されます。ステートフル・モデルの実装は、LLM のグリーディー検索とビーム検索 (プレビュー) の両方をサポートします。

開始する前に、この記事の「要件」セクションの指示に従って、必要なエクステンションと API をすべてインストールしていることを確認してください。

Hugging Face トークナイザーとモデルを OpenVINO™ IR 形式に変換する

LLM とトークナイザーをネイティブ OpenVINO™ API で使用するには、OpenVINO™ IR 形式に変換する必要があります。OpenVINO™ トークナイザーには、Hugging Face Hub のトークナイザーを OpenVINO™ IR 形式に変換するコマンド・ライン・インターフェイス (CLI) ツール、convert_tokenizer が用意されています。

```
convert_tokenizer HuggingFaceH4/zephyr-7b-beta --with-detokenizer -o
openvino_tokenizer
```

上記の例は、Hugging Face Hub の HuggingFaceH4/zephyr-7b-beta トークナイザーを変換します。`--with-detokenizer` 引数は、逆トークナイザーも変換するようにコマンドに指示します。`-o` 引数は、変換したオブジェクトを保存する出力ディレクトリーの名前 (この場合は openvino_tokenizer) を指定します。

次に、**optimum-cli** ツールを使用して、LLM を OpenVINO™ IR 形式に変換します。このツールは、Python* スクリプトを使用しないでモデルを変換する場合に役立ちます。

この変換を実行するコマンドの構造を次に示します。

```
optimum-cli export openvino --model <MODEL_NAME> <NEW_MODEL_NAME>
```

-- model <MODEL_NAME>: コマンドのこの部分は、変換するモデルの名前を指定します。<MODEL_NAME> を Hugging Face の実際のモデル名に置き換えます。

<NEW_MODEL_NAME>: ここで、OpenVINO™ IR 形式の新しいモデルに付ける名前を指定します。<NEW_MODEL_NAME> を新しいモデルに付ける名前に置き換えます。

例えば、Hugging Face の Llama 2–7B モデル (正式名称は meta-llama/Llama-2–7b-chat-hf) を OpenVINO™ IR モデルに変換して、「ov_llama_2」という名前を付けるには、次のコマンドを使用します。

```
optimum-cli export openvino --model meta-llama/Llama-2-7b-chat-hf ov_llama_2
```

この例では、**meta-llama/Llama-2–7b-chat-hf** が Hugging Face のモデル名で、**ov_llama_2** が変換後の OpenVINO™ IR モデルの新しい名前です。

さらに、CLI を使用してモデルをエクスポートするときに、`--weight-format` 引数を指定して 8 ビットまたは 4 ビットの重み量子化 (英語) を適用できます。モデル **gpt2** に 8 ビットの量子化を適用するコマンドの例を次に示します。

```
optimum-cli export openvino --model gpt2 --weight-format int8 ov_gpt2_model
```

モデルとトークナイザーが openvino_model フォルダーと openvino_tokenizer フォルダーに保存されます。

ステップ 1: モデルを読み取ってコンパイルする

モデルとトークナイザーが OpenVINO™ IR 形式に変換されたら、ov.core.compile_model メソッドを使用してモデルとトークナイザーを読み取ってコンパイルします。

```
import numpy as np
from pathlib import Path
import openvino_tokenizers
from openvino import compile_model, Tensor
model_dir = Path("path/to/model/directory")

# Compile the tokenizer, model, and detokenizer using OpenVINO. These files
are XML representations of the models optimized for OpenVINO
tokenizer = compile_model(model_dir / "openvino_tokenizer.xml")
detokenizer = compile_model(model_dir / "openvino_detokenizer.xml")
infer_request = compile_model(model_dir /
"openvino_model.xml").create_infer_request()
```

モデルとトークナイザーがコンパイルされ、推論に使用できるようになりました。

ステップ 2: 入力テキストをトークン化する

推論を実行する前に、入力テキストをトークン化して、モデルが想定する構造に設定する必要があります。トークン化は、入力テキストをモデルが理解して処理できる形式の、数値のシーケンス (「トークン」) に変換します。

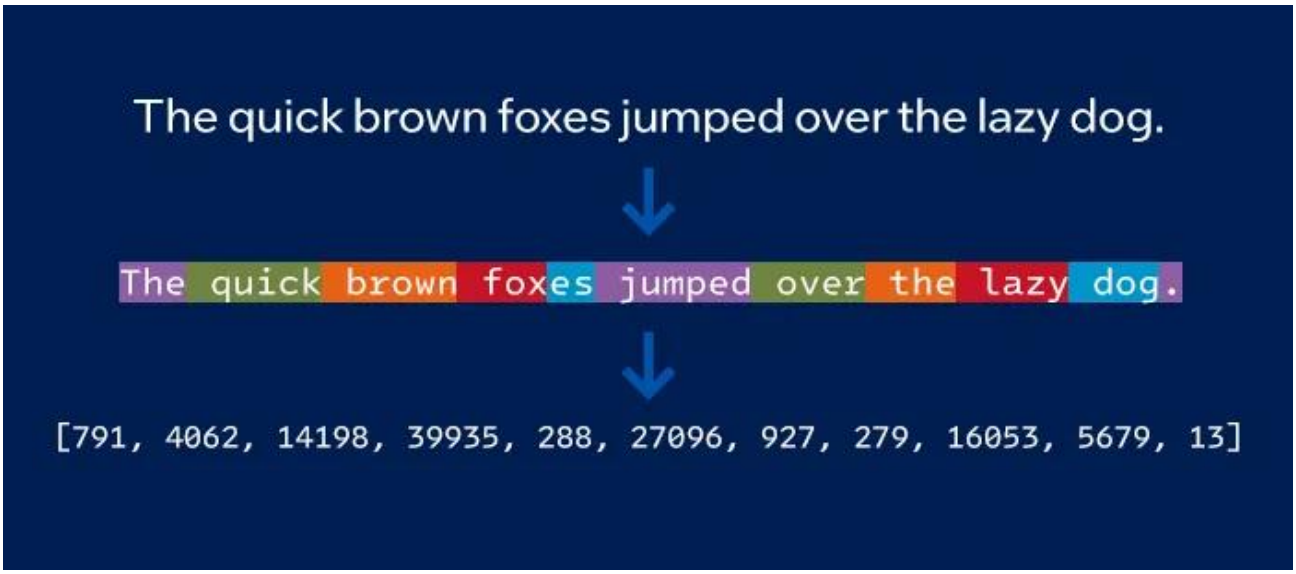

図 1. トークンに分割されたフレーズの例。各トークンには独自の数値が含まれます。[出典 (英語)]

次に示すように、コンパイルされたトークナイザーを使用して入力テキスト文字列をトークンに変換します。

```
text_input = [" What is OpenVINO?"]
model_input = {name.any_name: Tensor(output) for name, output in
tokenizer(text_input).items()}
```

ステップ 3: トークン生成ループを実行する

テキスト生成の中心となるのは、推論とトークン選択ループです。このループの各反復で、モデルは入力シーケンスに推論を実行し、新しいトークンを生成して選択し、既存のシーケンスに追加します。

```
if "position_ids" in (input.any_name for input in
infer_request.model_inputs):
    model_input["position_ids"] =
np.arange(model_input["input_ids"].shape[1], dtype=np.int64)[np.newaxis, :]

# no beam search, set idx to 0
model_input["beam_idx"] = Tensor(np.array(range(len(text_input)),
dtype=np.int32))

# end of sentence token - the model signifies the end of text generation
# for now can be obtained from the original tokenizer
`original_tokenizer.eos_token_id`
eos_token = 2

tokens_result = [[]]

# reset KV cache inside the model before inference
infer_request.reset_state()
max_infer = 10
```

```
for _ in range(max_infer):
    infer_request.start_async(model_input)
    infer_request.wait()

    # use greedy decoding to get most probable token as the model prediction
    output_token = np.argmax(infer_request.get_output_tensor().data[:, -
1, :], axis=-1, keepdims=True)
    tokens_result = np.hstack((tokens_result, output_token))

    if output_token[0][0] == eos_token:
        break

    # Prepare input for new inference
    model_input["input_ids"] = output_token
    model_input["attention_mask"] =
np.hstack((model_input["attention_mask"].data, [[1]]))
    model_input["position_ids"] = np.hstack(
        (model_input["position_ids"].data,
[[model_input["position_ids"].data.shape[-1]]])
    )
```

ステップ 4: 出力を逆トークン化する

プロセスの最後のステップは逆トークン化です。逆トークナイザーを使用して、モデルにより生成されたトークンの ID のシーケンスを人間が解読可能なテキストの文字列に変換します。

```
# Decode the model output back to string
text_result = detokenizer(tokens_result)["string_output"]
print(f"Prompt:\n{text_input[0]}")
print(f"Generated:\n{text_result[0]}")
```

この例を実行した結果の出力を次に示します。

[‘ <s> OpenVINO is an open-source toolkit for building and optimizing deep learning applications using Intel® hardware.’]

ネイティブ OpenVINO™ C++ API を使用した推論

前の例は、ステートフル・モデル手法を利用して C++ で実装することもできます。GitHub* の OpenVINO™ 生成 AI (英語) の次のプログラムは、トークナイザー、逆トークナイザー、OpenVINO™ IR 形式のモデルを OpenVINO™ にロードします。プロンプトはトークン化されて、モデルに渡されます。モデルは、特別なシーケンス終了 (EOS) トークンを取得するまで、トークンごとに予測を生成します。予測されたトークンは文字に変換され、ストリーミング形式で出力されます。

```
// Copyright (C) 2023-2024 Intel Corporation
// SPDX-License-Identifier: Apache-2.0

#include <openvino/openvino.hpp>

namespace {
std::pair<ov::Tensor, ov::Tensor> tokenize(ov::InferRequest& tokenizer,
std::string&& prompt) {
```

```
    constexpr size_t BATCH_SIZE = 1;
    tokenizer.set_input_tensor(ov::Tensor{ov::element::string, {BATCH_SIZE},
&prompt});
    tokenizer.infer();
    return {tokenizer.get_tensor("input_ids"),
tokenizer.get_tensor("attention_mask")};
}

std::string detokenize(ov::InferRequest& detokenizer, std::vector<int64_t>&
tokens) {
    constexpr size_t BATCH_SIZE = 1;
    detokenizer.set_input_tensor(ov::Tensor{ov::element::i64, {BATCH_SIZE,
tokens.size()}, tokens.data()});
    detokenizer.infer();
    return detokenizer.get_output_tensor().data<std::string>()[0];
}

// The following reasons require TextStreamer to keep a cache of previous
tokens:
// detokenizer removes starting ' '. For example detokenize(tokenize(" a"))
== "a",
// but detokenize(tokenize("prefix a")) == "prefix a"
// 1 printable token may consist of 2 token ids:
detokenize(incomplete_token_idx) == "□"
struct TextStreamer {
    ov::InferRequest detokenizer;
    std::vector<int64_t> token_cache;
    size_t print_len = 0;

    void put(int64_t token) {
        token_cache.push_back(token);
        std::string text = detokenize(detokenizer, token_cache);
        if (!text.empty() && '\n' == text.back()) {
            // Flush the cache after the new line symbol
            std::cout << std::string_view{text.data() + print_len,
text.size() - print_len};
            token_cache.clear();
            print_len = 0;
        }
        if (text.size() >= 3 && text.compare(text.size() - 3, 3, "□") == 0)
{
            // Don't print incomplete text
            return;
        }
        std::cout << std::string_view{text.data() + print_len, text.size() -
print_len} << std::flush;
        print_len = text.size();
    }

    void end() {
        std::string text = detokenize(detokenizer, token_cache);
        std::cout << std::string_view{text.data() + print_len, text.size() -
print_len} << '\n';
        token_cache.clear();
        print_len = 0;
    }
};
}
```

```
int main(int argc, char* argv[]) try {
    if (argc != 3) {
        throw std::runtime_error(std::string{"Usage: "} + argv[0] + "
<MODEL_DIR> '<PROMPT>'");
    }
    // Compile models
    ov::Core core;
    core.add_extension(USER_OV_EXTENSIONS_PATH);  // USER_OV_EXTENSIONS_PATH
is defined in CMakeLists.txt
    // tokenizer and detokenizer work on CPU only
    ov::InferRequest tokenizer = core.compile_model(
        std::string{argv[1]} + "/openvino_tokenizer.xml",
"CPU").create_infer_request();
    auto [input_ids, attention_mask] = tokenize(tokenizer, argv[2]);
    ov::InferRequest detokenizer = core.compile_model(
        std::string{argv[1]} + "/openvino_detokenizer.xml",
"CPU").create_infer_request();
    // The model can be compiled for GPU as well
    ov::InferRequest lm = core.compile_model(
        std::string{argv[1]} + "/openvino_model.xml",
"CPU").create_infer_request();
    // Initialize inputs
    lm.set_tensor("input_ids", input_ids);
    lm.set_tensor("attention_mask", attention_mask);
    ov::Tensor position_ids = lm.get_tensor("position_ids");
    position_ids.set_shape(input_ids.get_shape());
    std::iota(position_ids.data<int64_t>(), position_ids.data<int64_t>() +
position_ids.get_size(), 0);
    constexpr size_t BATCH_SIZE = 1;
    lm.get_tensor("beam_idx").set_shape({BATCH_SIZE});
    lm.get_tensor("beam_idx").data<int32_t>()[0] = 0;
    lm.infer();
    size_t vocab_size = lm.get_tensor("logits").get_shape().back();
    float* logits = lm.get_tensor("logits").data<float>() +
(input_ids.get_size() - 1) * vocab_size;
    int64_t out_token = std::max_element(logits, logits + vocab_size) -
logits;

    lm.get_tensor("input_ids").set_shape({BATCH_SIZE, 1});
    position_ids.set_shape({BATCH_SIZE, 1});
    TextStreamer text_streamer{std::move(detokenizer)};
    // There's no way to extract special token values from the detokenizer
for now
    constexpr int64_t SPECIAL_EOS_TOKEN = 2;
    while (out_token != SPECIAL_EOS_TOKEN) {
        lm.get_tensor("input_ids").data<int64_t>()[0] = out_token;
        lm.get_tensor("attention_mask").set_shape({BATCH_SIZE,
lm.get_tensor("attention_mask").get_shape().at(1) + 1});
        std::fill_n(lm.get_tensor("attention_mask").data<int64_t>(),
lm.get_tensor("attention_mask").get_size(), 1);
        position_ids.data<int64_t>()[0] =
int64_t(lm.get_tensor("attention_mask").get_size() - 2);
        lm.start_async();
        text_streamer.put(out_token);
        lm.wait();
        logits = lm.get_tensor("logits").data<float>();
        out_token = std::max_element(logits, logits + vocab_size) - logits;
```

```
    }
    text_streamer.end();
    // Model is stateful which means that context (kv-cache) which belongs to
a particular
    // text sequence is accumulated inside the model during the generation
loop above.
    // This context should be reset before processing the next text sequence.
    // While it is not required to reset context in this sample as only one
sequence is processed,
    // it is called for education purposes:
    lm.reset_state();
} catch (const std::exception& error) {
    std::cerr << error.what() << '\n';
    return EXIT_FAILURE;
} catch (...) {
    std::cerr << "Non-exception object thrown\n";
    return EXIT_FAILURE;
}
```

OpenVINO™ の重み圧縮最適化を使用して効率良く LLM を実行する

LLM を実行する前に、OpenVINO™ の重み圧縮を使用してストレージサイズとメモリー・フットプリントを削減します。同様の精度で、LLM を元のサイズの 1/4 または 1/8 に削減できます。詳細は、こちらの記事 (英語) を参照してください。

OpenVINO™ を使用して LLM をデプロイする、スマートで高速な方法を見つける

Python*、C++、または Optimum Intel API を使用して Hugging Face でデプロイされたかどうかに関係なく、OpenVINO™ の LLM は、会話型 AI の高度な機能を提供し、速度、インテルの公式サポート、柔軟性という追加のメリットも提供します。OpenVINO™ ツールキットを自分で試して、OpenVINO™ の旅を続けてください。

著者の帰属

この記事は、インテル AI エバンジェリストの Ria Cheruvu とインテル OpenVINO™ プロダクト・マネージャーの Ryan Loney によるソリューション・ホワイト・ペーパー「OpenVINO™ ツールキットを使用した大規模言語モデルの最適化 (英語)」の情報を抜粋して要約したものです。

追加のクレジット (敬称略): Ekaterina Aidova、Alexander Kozlov、Helena Kloosterman、Artur Paniukov、Dariusz Trawinski、Ilya Lavrenov、Nico Galoppo、Jan Iwaszkiewicz、Sergey Lyalin、Adrian Tobiszewski、Dariusz Trawinski、Jason Burris、Ansley Dunn、Michael Hansen、Raymond Lo、Yury Gorbachev、Adam Tumialis、Milosz Zeglarski

OpenVINO™ ツールキットとは

AI を加速する無償のツールである OpenVINO™ ツールキットは、インテルが無償で提供しているインテル製の CPU や GPU、VPU、FPGA などのパフォーマンスを最大限に活用して、コンピューター・ビジョン、画像関係をはじめ、自然言語処理や音声処理など、幅広いディープラーニング・モデルで推論を最適化し高速化する推論エンジン/ツールスイートです。

OpenVINO™ ツールキット・ページでは、ツールの概要、利用方法、導入事例、トレーニング、ツール・ダウンロードまでさまざまな情報を提供しています。ぜひ特設サイトにアクセスしてみてください。

https://www.intel.co.jp/content/www/jp/ja/internet-of-things/openvino-toolkit.html