Intel Unnati Industrial Training Report

Date:- 11/07/2024 **Name:- Tejasvee Dwivedi** **Email:-** tejasvee.dwivedi@learner.manipal.edu

Manipal Institute of Technology (2021-25)

Problem Statement:

Running GenAI on Intel AI Laptops and Simple LLM Inference on CPU and fine-tuning of LLM Models using Intel® OpenVINO™.

Description:

This problem statement is crafted to introduce the fascinating domain of Generative Artificial Intelligence (GenAI) through a series of interactive exercises. Participants will gain foundational knowledge of GenAI, conduct basic Large Language Model (LLM) inference on a CPU, perform inference with OpenVINO, and delve into the process of fine-tuning an LLM to develop a personalized Chatbot.

Major Challenges:

1. Large Model File Sizes: Pre-trained language models often have large file sizes, necessitating considerable storage space and memory for loading and running them efficiently.
2. Learning LLM Inference on CPU: Mastering the process of running large language models (LLMs) on CPUs, including optimizing performance and resource usage, can be complex.
3. Building a Custom Chatbot: Developing a custom chatbot using fine-tuned pre-trained large language models with Intel AI tools involves several intricate steps and challenges.
4. Efficient Tokenization and History Management: Implementing efficient tokenization and handling conversational history to ensure smooth interaction flow and accurate context retention is essential.
5. Real-time Text Processing: Developing effective methods for real-time text processing and generation, including handling partial text and managing stopping criteria during generation, is challenging.

Approach:

Intel OpenVINO is an open-source toolkit used for accelerating, optimizing and deploying LLMs (Large Language Models) on the edge and on the cloud irrespective of the hardware present on the computing machines.

It also allows very large language models which require a lot of memory and space to be run locally on any intel machine.

Using OpenVINO to optimize and run the language model can significantly affect CPU usage in the following ways:

Imagine you have a traditional model and an optimized model:

- **Traditional Model**: Requires a lot of CPU power to process data, causing high CPU usage and potentially slower response times. And we have to shift towards dedicated GPUs.
- **Optimized Model with OpenVINO**: Uses less CPU power by making the computations simpler and more efficient, resulting in lower CPU usage and faster response times.

Steps taken:

Step 1:

Configuring and Fetching the LLM Configuration File:

- Configuring and fetching the LLM (Large Language Model) configuration file involves setting up and retrieving essential settings and parameters necessary for initializing and running the model

- Define paths and variables where the configuration file will be stored or accessed. This involves specifying file paths, URLs, other sources where the configuration data resides.

```
# Define paths for the shared config file and destination config file
config_shared_path = Path("../../utils/llm_config.py")
config_dst_path = Path("llm_config.py")
```

- Use methods like HTTP requests (requests.get()) to fetch the configuration file from a specified URL or filesystem location. Ensure the data is retrieved securely and validated if needed.

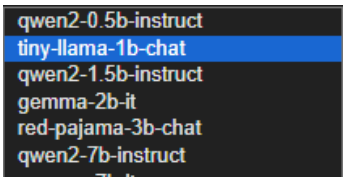
```
def fetch_and_write_config(url, path):
    # Fetch the config file from the given URL and write it to the specified path
    response = requests.get(url)
    with open(path, "w", encoding='utf-8') as f:
        f.write(response.text)
```

- Implement error handling to manage cases where the configuration file is not accessible or invalid. Validate the fetched data to ensure it meets expected formats and requirements before proceeding with model initialization.

```
if not config_dst_path.exists():
    if config_shared_path.exists():
        try:
            os.symlink(config_shared_path, config_dst_path)  # Try creating a symlink
        except OSError:
            shutil.copy(config_shared_path, config_dst_path)  # Copy if symlink fails
    else:
```

Step 2:

Importing SUPPORTED_LLM_MODELS and Selecting Model Language




```
from llm_config import SUPPORTED_LLM_MODELS
import ipywidgets as widgets

# Get the list of supported LLM models
model_languages = list(SUPPORTED_LLM_MODELS)
```

Step 3:

Generating and Executing Commands for Model Conversion to INT8 and INT4

- Converting a model into INT8 and INT4 formats compatible with OpenVINO. It begins by identifying the model ID and extracting its name for directory setup.
- Functions convert_to_int8() and convert_to_int4() handle the conversion process, utilizing configuration parameters for compression.

```
# Function to convert model to INT8 format
def convert_to_int8():
    if (int8_model_dir / "openvino_model.xml").exists():
        return
    int8_model_dir.mkdir(parents=True, exist_ok=True)
    export_command = generate_export_command(pt_model_id, "text-generation-with-past", "int8", int8_mo
    display(Markdown(f"**Export command:**\n\n`{export_command}`"))
    os.system(export_command)

# Function to convert model to INT4 format
def convert_to_int4():
    compression_configs = {
        "default": {"sym": False, "group_size": 128, "ratio": 0.8},
    }
    params = compression_configs.get(model_id.value, compression_configs["default"])
    if (int4_model_dir / "openvino_model.xml").exists():
        return
    additional_args = f"--group-size {params['group_size']} --ratio {params['ratio']}"
    if params["sym"]:
        additional_args += " --sym"
```

- The generate_export_command() function constructs commands for exporting the model to OpenVINO, ensuring compatibility for text generation tasks with past context. Commands are executed to create .xml and .bin files necessary for each format. This approach streamlines the preparation of optimized models for deployment, facilitating efficient inference on hardware supporting reduced precision computations.

```python
# Function to generate export command for OpenVINO
def generate_export_command(model_id, task, weight_format, output_dir, remote_code=False, additional_args=""):
    base_command = f"optimum-cli export openvino --model {model_id} --task {task} --weight-format {weight_format} {additional_args}"
    if remote_code:
        base_command += " --trust-remote-code"
    return f"{base_command} {output_dir}"
```

Step 4:

Setting up OpenVINO configuration using certain parameters.

The configuration settings (e.g., PERFORMANCE_HINT, NUM_STREAMS) allow fine-tuning of the model's performance characteristics. By adjusting these settings, you can optimize the balance between speed and CPU usage, potentially reducing the overall CPU load.

```python
# OpenVINO configuration settings
ov_config = {"PERFORMANCE_HINT": "LATENCY", "NUM_STREAMS": "1", "CACHE_DIR": ""}
```

PERFORMANCE_HINT: This parameter provides a high-level performance hint to the inference engine. Common values include "LATENCY", "THROUGHPUT", and "BALANCED".

- Set to "LATENCY" to optimize for lower latency.

NUM_STREAMS: This parameter sets the number of inference streams (or parallel inference requests) that can be processed simultaneously.

- Set to "1", indicating that only one stream will be used.

CACHE_DIR: This parameter specifies the directory where the model cache is stored.

- Set to an empty string, which might mean no caching is used

Step 5:

Setting Up Conversational AI with OpenVINO Model Integration

Text Processing and History Conversion:

- Functions like default_partial_text_processor() handle how new text is integrated into ongoing conversation history. convert_history_to_token() converts conversation history into tokenized format suitable for model input, considering optional templates for message formatting.

```python
# Define text processing function
def default_partial_text_processor(partial_text: str, new_text: str):
    partial_text += new_text
    return partial_text

text_processor = model_configuration.get("partial_text_processor", default_partial_text_processor)
```

User and Bot Callback Functions:

- user(message, history) and bot(history, ...): These functions manage interactions between users and the AI bot. The user function accepts new messages and updates the conversation history. The bot function generates responses based on the current conversation history and specified generation parameters (temperature, top_p, top_k, repetition_penalty).

```python
# Define user callback function
def user(message, history):
    return "", history + [[message, ""]]

# Define bot callback function
def bot(history, temperature, top_p, top_k, repetition_penalty, conversation_id):
    input_ids = convert_history_to_token(history)
    streamer = TextIteratorStreamer(tok, timeout=30.0, skip_prompt=True, skip_special_tokens=True)
    generate_kwargs = dict(
        input_ids=input_ids,
        max_new_tokens=max_new_tokens,
        temperature=temperature,
        do_sample=temperature > 0.0,
        top_p=top_p,
        top_k=top_k,
        repetition_penalty=repetition_penalty,
        streamer=streamer,
    )
```

- **Temperature**: Controls the randomness of the model's output; lower values make the output more focused and deterministic, while higher values make it more diverse and creative.
- **Top-p (nucleus sampling)**: Limits the next token selection to a subset of tokens that together have a cumulative probability of p, encouraging diversity while maintaining coherence.
- **Top-k**: Limits the next token selection to the top k most probable tokens, reducing the likelihood of rare and potentially irrelevant tokens.
- **Repetition Penalty**: Reduces the probability of previously generated tokens to avoid repetitive loops and encourage varied output.

Asynchronous Generation with Threads:

- Response generation (ov_model.generate(**generate_kwargs)) is handled asynchronously using threads (Thread). This setup allows the bot to generate responses concurrently while maintaining responsiveness to ongoing user interactions.

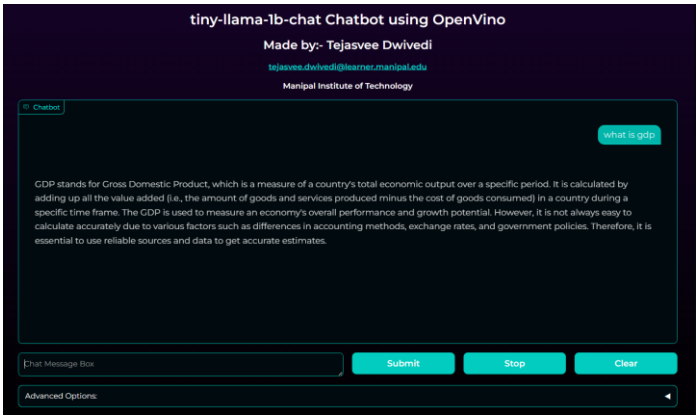
```
def generate_and_signal_complete():
    ov_model.generate(**generate_kwargs)
    stream_complete.set()

t1 = Thread(target=generate_and_signal_complete)
t1.start()
```

Integration with OpenVINO:

- The model integration with OpenVINO (OVModelForCausalLM) optimizes model performance for inference on CPUs. This includes setting performance hints (LATENCY) and managing computational resources (NUM_STREAMS).

Step 6:

Now using gradio interface we get the input generate the output and send it to the webpage starting a conversation here we have made a function which conserves the history of a conversation and this is helpful to retain context while answering a query.



Why We Chose the Tiny LLaMA 1B Chat Model

Advantages:

- **Model Size**: Approximately 1 billion parameters.
- **Inference Speed**: Optimized for faster inference on CPUs, achieving speeds suitable for real-time applications.
- **Memory Usage**: Requires significantly less memory compared to larger language models, enhancing efficiency on resource-constrained devices.
- **Performance**: Demonstrates competitive performance metrics in conversational AI tasks, balancing accuracy with computational efficiency.
- **Deployment Time**: Quick deployment due to its smaller size and optimized architecture, facilitating integration into various applications.

Limitations:

Despite its benefits, the Tiny LLaMA 1B may not generate responses as nuanced or contextually rich as larger models. This limitation can affect its ability to handle complex dialogue scenarios requiring extensive background knowledge or highly detailed responses. However, its efficiency and speed outweigh these drawbacks in applications prioritizing rapid response times and resource conservation.

Whats the benefit of OpenVINO

Reduced CPU Load

Model Compression:

- INT8 and INT4 Formats: Converting the model to lower-precision formats (INT8 and INT4) reduces the computational complexity of the model. This means the model requires fewer CPU resources to perform the same tasks, resulting in lower CPU load during inference.

Faster Execution

Optimized Inference:

- **OpenVINO Runtime**: OpenVINO is designed to optimize the execution of deep learning models on Intel hardware. By using OpenVINO, the model can take advantage of specific optimizations and instructions that are available on Intel CPUs, leading to faster inference times and more efficient CPU usage.

Improved Resource Utilization

Performance Settings:

- Configuration Settings: The configuration settings (e.g., PERFORMANCE_HINT, NUM_STREAMS) allow fine-tuning of the model's performance characteristics. By adjusting these settings, you can optimize the balance between speed and CPU usage, potentially reducing the overall CPU load.

Efficient Multi-threading

Thread Management:

- Efficient Multi-threading: OpenVINO efficiently manages multiple threads during model inference. This means it can distribute the workload more effectively across multiple CPU cores, improving overall CPU utilization and preventing any single core from becoming a bottleneck.

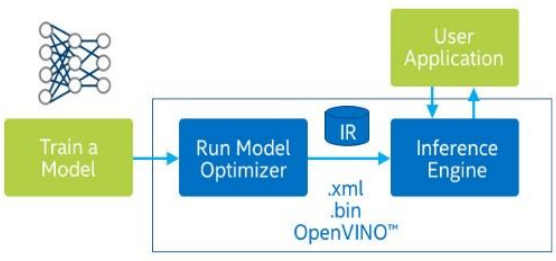
Without OpenVINO:

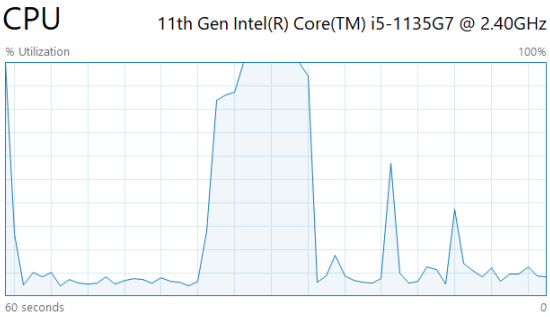
- Running a high-precision model (e.g., FP32) directly on the CPU might result in high CPU usage because the model computations are more complex and resource-intensive.

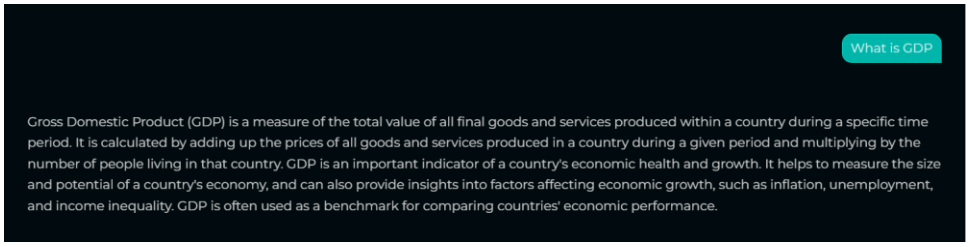
With OpenVINO:

- By using INT8 or INT4 compressed models and optimized inference with OpenVINO, the same tasks can be performed with lower CPU usage. This means the CPU can handle more requests simultaneously or perform other tasks while running the model.

❖ RESULTS:- CPU USAGE WHILE WE ARE RUNNING THE PROMPT IN THE CHATBOT







Project Contribution:

(Individual project)

Name- TEJASVEE DWIVEDI

Mentor- Mr. Manjunath Vishweshwar Hegde (MAHE-MIT)

REG. NO.- 210968158

College- Manipal Institute of Technology

Email- tejasvee.dwivedi@learner.manipal.edu

References:

1. Intel Github Repositories:- **https://github.com/openvinotoolkit/openvino**
2. Intel Dev Cloud for Edge**:-** **https://www.intel.com/content/www/us/en/developer/tools/devcloud/edge/overview.html**
3. **https://huggingface.co/Intel/neural-chat-7b-v3-3**
4. Gradio themes**:-** **https://huggingface.co/spaces/gradio/theme-gallery**
5. Python Langchain**:-** **https://python.langchain.com/v0.2/docs/integrations/llms/openvino/**
6. YouTube:-https://www.youtube.com/@AbhishekNandy/videos
7. **https://medium.com/@abhishek.nandy81**