Build AI Agent in NIM

Building a simple AI agent with NVIDIA's **Neural Module Toolkit (NVIDIA NIM)** can be straightforward and quick. Below are the steps you can follow to create a basic AI agent for tasks like text classification or speech recognition.

---

1. Prerequisites

Before starting, ensure you have:

- **NVIDIA GPU** with the latest CUDA drivers installed.
- **Python** (3.8 or later).
- **NVIDIA NGC account** (for downloading pre-trained models).
- **NVIDIA PyTorch containers** installed.

Install NVIDIA NIM via pip:

```bash
pip install nemo_toolkit
```

---

2. Setting Up the Environment

Import necessary libraries:

```python
import nemo
import nemo.collections.nlp as nemo_nlp
import nemo.collections.asr as nemo_asr
```

Create a NeMo Core instance:

```python
from nemo.core import NeMo

nemo_core = NeMo()
```

---

3. Choose a Pre-trained Model

NVIDIA NIM supports various domains:

- **NLP (Text Processing)**: Models for text classification, NER, or summarization.

- **ASR (Speech Recognition)**: Models for transcribing audio.

Example: Load a pre-trained ASR model:

```python
from nemo.collections.asr.models import ASRModel

asr_model = ASRModel.from_pretrained(model_name="stt_en_conformer_ctc_large")
```

For text-based tasks:

```python
from nemo.collections.nlp.models import TextClassificationModel

nlp_model = TextClassificationModel.from_pretrained(model_name="text_classification_bert")
```

---

4. Process Input Data

Prepare input data:

- **Audio input** for ASR:

```python
audio_file = "sample_audio.wav"
transcription = asr_model.transcribe(paths2audio_files=[audio_file])
print(f"Transcription: {transcription}")
```

- **Text input** for NLP:

```python
text_input = ["NVIDIA is amazing for AI development!", "AI is the future."]
predictions = nlp_model.classifytext(texts=text_input)
print(f"Predictions: {predictions}")
```

---

5. Fine-tuning (Optional)

If desired, fine-tune the model using custom data:

```python
# Specify dataset and hyperparameters
train_dataset = "path/to/dataset.json"
asr_model.finetune(train_dataset=train_dataset, num_epochs=5, lr=0.001)
```

6. Save the Model

Export the model for deployment:

```python
asr_model.save_to("asr_model.nemo")
```

---

7. Deployment

Deploy the model using NVIDIA Triton Inference Server for real-time inference.

Install Triton:

```bash
docker run --gpus all -it --rm nvcr.io/nvidia/tritonserver:latest-py3
```

Summary

Within five minutes, you can:

1. Set up NVIDIA NIM.
2. Load a pre-trained model.
3. Process input data for tasks like ASR or NLP.
4. (Optionally) Fine-tune the model.
5. Save and deploy it for production use.

Let me know if you'd like a more specific walkthrough or advanced configurations!

To implement an AI agent using **LangChain** (a framework for building applications powered by language models) and **Groq** (GroqChip is a platform for accelerated AI and ML tasks), follow these steps:

---

1. Prerequisites

- **LangChain**: Python library for orchestrating language model tasks.
- **Groq**: Hardware and software platform for AI/ML tasks.
- **Python Environment** with the necessary libraries installed.

Install LangChain:

```bash
pip install langchain openai
```

Ensure you have access to the **Groq SDK** and hardware for accelerated inference. Follow Groq's official setup guide to get started.

---

2. Conceptual Overview

We'll build an AI agent that:

1. **Receives user input.**
2. **Processes the input with LangChain** for context-driven responses.
3. **Offloads complex model inference** to Groq for high-speed execution.

---

3. Setting Up LangChain

LangChain allows chaining prompts and functions for powerful reasoning.

Example: Create a basic conversational agent:

```python
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# Set up the language model (OpenAI GPT)
llm = OpenAI(model="text-davinci-003", temperature=0.7)

# Define a prompt
prompt = PromptTemplate(
    input_variables=["user_input"],
    template="You are an assistant. Answer the question: {user_input}"
)

# Create a chain
conversation_chain = LLMChain(llm=llm, prompt=prompt)
```

---

4. Integrating Groq for Inference

Groq's hardware accelerates AI tasks like transformers and deep learning models. Use Groq's SDK to run computations.

Example setup for a model offloaded to Groq:

```python
from groq.runtime import groq_model
from groq.api import groq_sdk

# Load a pre-trained model (e.g., BERT or GPT)
groq_model_path = "path_to_groq_model"

# Initialize the Groq runtime
groq_session = groq_sdk.Session()
groq_inference = groq_model.load_model(groq_model_path)

# Function to process input with Groq
def groq_process(input_text):
    encoded_input = groq_sdk.encode(input_text)
```

```
    result = groq_inference.run(groq_session, inputs=[encoded_input])
    return groq_sdk.decode(result)
```

---

5. Combining LangChain and Groq

Integrate the Groq processing step into LangChain's workflow.

```python
# Define a hybrid agent function
def ai_agent(user_query):
    # Step 1: Use LangChain for natural language understanding
    llm_response = conversation_chain.run({"user_input": user_query})

    # Step 2: Pass the processed response to Groq for enhanced computation
    groq_output = groq_process(llm_response)

    return groq_output

# Example usage
user_input = "What are the applications of quantum computing?"
response = ai_agent(user_input)
print(f"AI Agent Response: {response}")
```

---

6. Fine-Tuning and Optimization

- **Fine-Tune Models**: Use custom datasets and frameworks like PyTorch/TF on Groq hardware.
- **Dynamic Routing**: Use LangChain for handling high-level tasks and route performance-heavy tasks to Groq.

---

7. Deployment

Deploy the AI agent using APIs or a microservice architecture. For real-time deployment:

1. **Use FastAPI**:

```python
from fastapi import FastAPI

app = FastAPI()

@app.post("/query")
async def query(user_input: str):
    response = ai_agent(user_input)
    return {"response": response}
```

2. **Containerize**:

```bash
docker build -t langchain-groq-agent .
```

```
docker run --gpus all -p 8000:8000 langchain-groq-agent
```

8. Summary

By combining **LangChain**'s flexibility with **Groq**'s hardware acceleration, you can build an efficient AI agent capable of advanced reasoning and high-speed inference. This architecture is scalable, modular, and well-suited for production use.

Would you like help with specific integrations, such as fine-tuning models for Groq or expanding LangChain's capabilities?

To implement an AI agent using **LangChain** and **Groq** without relying on OpenAI, you can leverage **open-source models** like **Hugging Face Transformers** and deploy them efficiently with Groq hardware. Here's how:

1. Prerequisites

- **LangChain**: Framework for LLM-based workflows.
- **Hugging Face Transformers**: Open-source models for NLP tasks.
- **Groq SDK**: Required for running models on Groq hardware.

Install necessary packages:

```bash
pip install langchain transformers sentencepiece groq-runtime
```

2. Choose an Open-Source Model

Instead of OpenAI models, use Hugging Face models such as:

- **Flan-T5**: For conversational tasks.
- **BERT**: For classification tasks.
- **GPT-NeoX**: Open-source generative models.

Example: Load a Hugging Face model:

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load the model and tokenizer
model_name = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
```

3. Setting Up LangChain

LangChain can interact with Hugging Face models seamlessly.

Create a custom LangChain LLM wrapper:

```python
from langchain.llms.base import LLM

class HuggingFaceLLM(LLM):
    def __init__(self, model, tokenizer):
        self.model = model
        self.tokenizer = tokenizer

    def _call(self, prompt: str, stop: Optional[List[str]] = None) -> str:
        inputs = self.tokenizer.encode(prompt, return_tensors="pt")
        outputs = self.model.generate(inputs, max_length=100)
        return self.tokenizer.decode(outputs[0], skip_special_tokens=True)

    @property
    def _identifying_params(self):
        return {"model_name": "Hugging Face LLM"}
```

Initialize the LangChain LLM:

```python
llm = HuggingFaceLLM(model=model, tokenizer=tokenizer)
```

---

4. Integrating Groq

Offload computationally intensive tasks to Groq hardware for efficient inference.

1. **Convert the model to Groq format**: Use Groq SDK to compile the Hugging Face model:

```bash
groq-compile --input_model="path/to/huggingface/model" --output_model="model.groq"
```

2. **Run Inference on Groq**: Load the compiled model in Python:

```python
from groq.runtime import groq_model
from groq.api import groq_sdk

# Load Groq model
groq_model_path = "model.groq"
groq_session = groq_sdk.Session()
groq_inference = groq_model.load_model(groq_model_path)

# Define Groq inference function
def groq_infer(input_text):
    encoded_input = groq_sdk.encode(input_text)
```

```
        result = groq_inference.run(groq_session, inputs=[encoded_input])
        return groq_sdk.decode(result)
```

---

5. Combining LangChain and Groq

Integrate Groq's processing into LangChain’s workflow.

Define an AI agent that uses LangChain for preprocessing and Groq for inference:

```python
def ai_agent(user_query):
    # Use LangChain to process the query
    prompt = f"Answer the following question: {user_query}"
    processed_query = llm(prompt)

    # Use Groq for high-speed inference
    groq_response = groq_infer(processed_query)

    return groq_response

# Example usage
user_input = "Explain the advantages of renewable energy."
response = ai_agent(user_input)
print(f"AI Agent Response: {response}")
```

---

6. Deployment

- **REST API** with FastAPI:

```python
from fastapi import FastAPI

app = FastAPI()

@app.post("/query")
async def query(user_input: str):
    response = ai_agent(user_input)
    return {"response": response}
```

- **Dockerize** the Application:

```dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

---

7. Summary

This solution creates an AI agent powered by **Hugging Face models** and **Groq hardware** while using **LangChain** for task orchestration. The integration allows for:

1. **Preprocessing and response chaining** using LangChain.
2. **Efficient inference acceleration** with Groq.

Would you like detailed guidance on Groq SDK usage, specific model conversions, or advanced LangChain configurations?

It looks like you are trying to set up the Groq SDK in your Python environment. The code snippet you provided seems to initialize a **Groq client** using an API key. Let me guide you through setting this up properly.

---

1. Install the Groq SDK

Ensure the Groq library is installed. If the `` `groq` `` package is available via pip, your command should work:

```bash
pip install -q -U groq
```

If Groq is not available via pip, you'll need to check its official documentation for setup instructions.

---

2. Setting Up the API Key

Replace `` `"your_key"` `` with your actual API key. If you don't have one, check how to obtain it via Groq's developer portal.

Example:

```python
api_keyy = "your_actual_api_key"
```

---

3. Initializing the Groq Client

Once the API key is set, initialize the Groq client:

```python
from groq import Groq

# Initialize the client
client = Groq(api_key=api_keyy)

# Verify the client connection
try:
```

```
    print("Connecting to Groq...")
    status = client.get_status()
    print(f"Groq Status: {status}")
except Exception as e:
    print(f"Error connecting to Groq: {e}")
```

---

4. Use the Groq Client

You can use the `client` object to interact with Groq services, such as uploading models, running inferences, or fetching system statuses. For example:

- **Upload a Model**:

  ```python
  model_path = "path/to/your/model.onnx"  # ONNX or other supported formats
  groq_model = client.upload_model(model_path)
  print(f"Model uploaded: {groq_model}")
  ```

- **Run Inference**:

  ```python
  input_data = {"input": [1, 2, 3, 4]}  # Replace with actual data
  response = client.run_inference(model_id=groq_model["id"], inputs=input_data)
  print(f"Inference Output: {response}")
  ```

- **Get Model Info**:

  ```python
  model_info = client.get_model_info(model_id=groq_model["id"])
  print(f"Model Info: {model_info}")
  ```

---

5. Troubleshooting

- **Invalid API Key**: Ensure your key is correctly set in `api_keyy`. Invalid keys will raise authentication errors.
- **Library Installation**: If the `groq` library isn't found, confirm its availability on pip or other channels.
- **Dependencies**: Ensure all dependencies for Groq's Python SDK are installed.

Next Steps

Once your client is set up and working:

1. Test it with a simple ONNX model to ensure inference works.
2. Combine Groq's high-speed inference with your AI agent logic in LangChain or other frameworks.

3. Deploy the agent for real-time interaction.

Would you like help debugging this, or guidance on how to structure your AI pipeline with Groq?

Here's a full example combining **LangChain** for workflow orchestration and **Groq** for accelerated inference. The example assumes you have the necessary dependencies and Groq SDK set up.

Prerequisites

1. Install dependencies:

```bash
pip install langchain transformers sentencepiece groq
```

2. Make sure you have:
    - A valid **Groq API key**.
    - A **Hugging Face model** like GPT-2 or BERT in ONNX format.

Complete Code Example

```python
# Import libraries
from langchain.llms.base import LLM
from transformers import AutoTokenizer
from groq import Groq

# Set up Groq API key
api_keyy = "your_actual_api_key"
client = Groq(api_key=api_keyy)

# Hugging Face tokenizer for input/output processing
model_name = "gpt2"  # Replace with your model
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Upload model to Groq
def upload_model_to_groq(model_path):
    print("Uploading model to Groq...")
    groq_model = client.upload_model(model_path)
    print(f"Model uploaded successfully: {groq_model}")
    return groq_model

# Groq inference function
def groq_infer(groq_model, input_text):
    print("Running inference on Groq...")
    # Tokenize input
    encoded_input = tokenizer.encode(input_text, return_tensors="pt").tolist()

    # Send input to Groq model
    response = client.run_inference(model_id=groq_model["id"], inputs={"input": encoded_input})

    # Decode response
    output_tokens = response["output"]
    return tokenizer.decode(output_tokens[0], skip_special_tokens=True)
```

```
# LangChain LLM Wrapper
class GroqLangChainLLM(LLM):
    def __init__(self, groq_model):
        self.groq_model = groq_model

    def _call(self, prompt: str, stop: Optional[list] = None) -> str:
        return groq_infer(self.groq_model, prompt)

    @property
    def _identifying_params(self):
        return {"model_name": "Groq LLM"}

# AI Agent Function
def ai_agent(query, groq_model):
    # Define the prompt
    prompt = f"Answer the following question: {query}"
    # Run the model
    response = groq_infer(groq_model, prompt)
    return response

# Main Script
if __name__ == "__main__":
    # Path to your ONNX model file
    model_path = "path/to/your/model.onnx"

    # Upload model to Groq
    groq_model = upload_model_to_groq(model_path)

    # Test the AI Agent
    user_query = "What are the benefits of renewable energy?"
    response = ai_agent(user_query, groq_model)
    print(f"AI Agent Response: {response}")
```

Explanation

1. **Model Setup**:
    - Use a Hugging Face model converted to ONNX format.
    - Upload the model to Groq using their SDK.
2. **LangChain Integration**:
    - Wrap Groq inference into a LangChain-compatible LLM class (`` `GroqLangChainLLM` ``).
    - Use it to process prompts and generate responses.
3. **Inference with Groq**:
    - Tokenize user input.
    - Send it to Groq for accelerated inference.
    - Decode the output and return the response.
4. **AI Agent**:
    - Combines the above components to create an interactive AI agent.

Next Steps

1. **Model Preparation**: If your Hugging Face model isn't in ONNX format, export it:

```bash
transformers-cli convert --model_type=gpt2 --model_name_or_path=gpt2 --save_dir=onnx_model
```

2. **Optimization**:
   - Test your model performance on Groq and tune batch sizes for efficiency.
   - Use LangChain workflows (e.g., chains and memory) for more complex logic.
3. **Deployment**:
   - Wrap this in a web server (e.g., FastAPI).
   - Deploy it on a cloud environment connected to Groq hardware.

Let me know if you'd like help with any of these steps!

To replace the OpenAI-style `` `client.chat.completions.create` `` with a Groq-compatible model, you need to use Groq's SDK to send your prompts to a model deployed on Groq hardware. Here's how you can adapt your code:

Using a Groq Model for Chat Completions

Here's a replacement implementation:

```python
# Import required libraries
from groq import Groq
from transformers import AutoTokenizer

# Set up Groq API client
api_keyy = "your_actual_api_key"
client = Groq(api_key=api_keyy)

# Hugging Face tokenizer for prompt processing
model_name = "path_to_your_llama_model_tokenizer"
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Upload Groq model (if not already uploaded)
def upload_model_to_groq(model_path):
    print("Uploading model to Groq...")
    groq_model = client.upload_model(model_path)
    print(f"Model uploaded successfully: {groq_model}")
    return groq_model

# Groq inference function for chat completions
def groq_chat_completion(groq_model, sensor_log, max_tokens=150):
    print("Running chat completion on Groq...")

    # Prepare the input as a single prompt
    prompt = f"prompts:\n\n{sensor_log}"

    # Tokenize input
    encoded_input = tokenizer.encode(prompt, return_tensors="pt").tolist()

    # Send input to Groq model
    response = client.run_inference(
        model_id=groq_model["id"],
        inputs={"input": encoded_input, "max_tokens": max_tokens}
    )

    # Decode the output
    output_tokens = response["output"]
    return tokenizer.decode(output_tokens[0], skip_special_tokens=True)

# Main script
if __name__ == "__main__":
    # Path to the ONNX model
```

```
    model_path = "path_to_your_llama3-8b-8192_model.onnx"

    # Upload model to Groq
    groq_model = upload_model_to_groq(model_path)

    # Example sensor log
    sensor_log = "Temperature is stable, battery voltage normal, motor functioning within sa
fe limits."

    # Run chat completion
    response = groq_chat_completion(groq_model, sensor_log)
    print(f"Groq Model Response: {response}")
```

Changes Made

1. **Replace OpenAI API**:
    - The OpenAI-style API (`` `client.chat.completions.create` ``) is replaced with Groq's `` `run_inference` `` method.
2. **Tokenization**:
    - Preprocess the input using a Hugging Face tokenizer compatible with your model (e.g., Llama 3).
3. **Inference on Groq**:
    - Use the Groq client to send tokenized input to the deployed model.
4. **Decoding**:
    - Convert the tokenized output back to human-readable text using the same tokenizer.

Important Notes

1. **Model Format**: Ensure your Llama 3 model is converted to ONNX or a format Groq supports:

    ```bash
    transformers-cli convert --model_type=llama --model_name_or_path=llama3-8b --save_dir=o
    nnx_model
    ```

2. **Inputs and Outputs**:
    - Confirm that Groq expects a specific input/output schema for text generation. Adjust `` `inputs` `` and `` `response` `` parsing as needed.
3. **Optimization**:
    - Test different `` `max_tokens` `` and batch sizes to maximize performance on Groq hardware.

Next Steps

- **Debugging**: Ensure the Groq model processes the tokenized input correctly.
- **Deployment**: Wrap this functionality in a web service (e.g., FastAPI).
- **Improvement**: Add chaining logic using LangChain for more complex workflows.

Let me know if you'd like further assistance!

To perform anomaly detection using Groq in a manner similar to the provided script, you would use Groq's API and processing pipeline for sending the `sensor_log` as input. Below is the updated code adapted for Groq:

Groq Implementation for Anomaly Detection

```python
# Import required libraries
from groq import Groq
from transformers import AutoTokenizer

# Initialize Groq API client
api_key = "your_actual_api_key"
client = Groq(api_key=api_key)

# Hugging Face tokenizer for preprocessing
model_name = "path_to_your_llama_model_tokenizer"
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Example sensor data
sensor_data = [
    "Cycle 1",
    "Non-Sidewinder Charging: Charging Section 0 by 74 units",
    # (remaining sensor data is unchanged for brevity)
    "--- Comparison ---",
    "Non-Sidewinder Average Health: 100.00, Overload Events: 1",
    "Sidewinder Average Health: 99.97, Overload Events: 3"
]

# Prepare the log data
sensor_log = "\n".join(sensor_data)

# Upload the Groq model
def upload_model_to_groq(model_path):
    print("Uploading model to Groq...")
    groq_model = client.upload_model(model_path)
    print(f"Model uploaded successfully: {groq_model}")
    return groq_model

# Groq inference function for anomaly detection
def groq_anomaly_detection(groq_model, sensor_log, max_tokens=150):
    print("Running anomaly detection on Groq...")

    # Prepare the input prompt
    prompt = f"Please analyze the following sensor data and highlight any anomalies:\n\n{sen
sor_log}"

    # Tokenize the prompt
    encoded_input = tokenizer.encode(prompt, return_tensors="pt").tolist()

    # Send the input to the Groq model
    response = client.run_inference(
        model_id=groq_model["id"],
        inputs={"input": encoded_input, "max_tokens": max_tokens}
    )

    # Decode the output tokens
    output_tokens = response["output"]
    return tokenizer.decode(output_tokens[0], skip_special_tokens=True)

# Main script
if __name__ == "__main__":
    # Path to the ONNX model
    model_path = "path_to_your_llama3-8b-8192_model.onnx"
```

```
    # Upload model to Groq
    groq_model = upload_model_to_groq(model_path)

    # Perform anomaly detection
    result = groq_anomaly_detection(groq_model, sensor_log)
    print("Anomaly Detection Result:", result)
```

Key Components

1. **Model Upload to Groq**:
   - `upload_model_to_groq`: Handles uploading the ONNX model to Groq for inference.
2. **Prompt Preparation**:
   - The sensor log is formatted as a single string prompt and tokenized using the Hugging Face tokenizer.
3. **Inference with Groq**:
   - The `run_inference` method is used to send the input and receive the output from Groq.
4. **Output Decoding**:
   - The response tokens are decoded into a human-readable format using the tokenizer.

Considerations

- **Model Format**: Ensure the Llama model is converted to ONNX or another Groq-compatible format.
- **API Schema**: Adapt the `run_inference` method to the expected input/output schema for your specific Groq deployment.
- **Performance Tuning**: Experiment with `max_tokens` and other parameters to optimize performance.
- **Error Handling**: Add error handling for API calls, model uploads, and inference runs.

This script mirrors the functionality of the original while leveraging Groq's infrastructure. Let me know if you need further refinements!

To convert a model into the ONNX format, you can use tools such as **PyTorch ONNX Exporter** or **TensorFlow ONNX Converter**, depending on the framework in which your model is implemented. Here's a step-by-step guide for both PyTorch and TensorFlow models:

---

1. PyTorch to ONNX

Step 1: Install Dependencies

```bash
pip install torch torchvision onnx
```

OpenAI

Step 2: Prepare Your Model

Ensure that your PyTorch model is loaded and ready for export:

```python
import torch
import torch.nn as nn

# Example model (Replace this with your actual model)
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(10, 2)

    def forward(self, x):
        return self.fc(x)

# Load or define your model
model = SimpleModel()
model.eval()  # Set to evaluation mode
```

Step 3: Export to ONNX

```python
# Dummy input tensor for model tracing
dummy_input = torch.randn(1, 10)

# Export the model
torch.onnx.export(
    model,                             # Model instance
    dummy_input,                       # Example input tensor
    "model.onnx",                      # Output file name
    export_params=True,                # Store the parameters
    opset_version=12,                  # ONNX opset version
    input_names=["input"],             # Input tensor name
    output_names=["output"],           # Output tensor name
    dynamic_axes={"input": {0: "batch_size"}, "output": {0: "batch_size"}} # Support dynamic batch size
)

print("Model exported to ONNX format!")
```

Step 4: Verify the ONNX Model

You can verify the exported ONNX model using the `onnx` package:

```python
import onnx

# Load and check the ONNX model
onnx_model = onnx.load("model.onnx")
onnx.checker.check_model(onnx_model)
print("ONNX model is valid!")
```

---

2. TensorFlow/Keras to ONNX

Step 1: Install Dependencies

```bash
pip install tensorflow tf2onnx
```

Step 2: Convert TensorFlow/Keras Model

```python
import tensorflow as tf
import tf2onnx

# Example model (Replace this with your actual model)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation="relu", input_shape=(10,)),
    tf.keras.layers.Dense(2, activation="softmax")
])

# Save the model
model.save("model_saved")  # Save in TensorFlow SavedModel format

# Convert the model to ONNX
!python -m tf2onnx.convert --saved-model model_saved --output model.onnx --opset 12

print("Model exported to ONNX format!")
```

---

3. Hugging Face Models to ONNX

If you're working with transformer models (e.g., Llama, BERT, GPT):

Step 1: Install Dependencies

```bash
pip install transformers optimum
```

Step 2: Convert Hugging Face Model

```python
from transformers import AutoTokenizer, AutoModel
from optimum.onnxruntime import ORTModelForSequenceClassification
from optimum.exporters.onnx import export_models

# Load the model and tokenizer
model_name = "bert-base-uncased"  # Replace with your model
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

```
# Export to ONNX
onnx_path = "bert_model.onnx"
export_models(
    model=model,
    output_dir=onnx_path,
    opset=12,
    task="sequence-classification"
)

print("Model exported to ONNX format!")
```

---

4. Tools to Convert ONNX

- **Netron**: Use Netron to visualize and verify the exported ONNX model.
- **ONNX Runtime**: Test and deploy the ONNX model using ONNX Runtime.

Let me know if you need help with a specific model or framework!