🧪 Lab 8.4: Deploy Triton with TensorFlow and ONNX Models

Objective

By the end of this lab, you will:

- Deploy TensorFlow and ONNX models inside NVIDIA Triton Inference Server
- Configure model repositories for multi-framework serving
- Query both models through Triton's unified HTTP/gRPC APIs
- Understand how Triton abstracts framework differences

🔷 Step 1: Prepare the Environment

1. **Provision a GPU machine**
    - Use an NVIDIA GPU-enabled instance (AWS p3, GCP A100, or local workstation with CUDA installed).
    - Ensure **NVIDIA drivers + Docker** are installed.
    - Run:

      ```
      nvidia-smi
      ```

      to confirm the GPU is visible.

    👉 *Why?* Triton requires GPU acceleration for TensorFlow and ONNX backends.

2. **Pull the Triton Inference Server container**

    ```
    docker pull nvcr.io/nvidia/tritonserver:23.12-py3
    ```

    👉 *Why?* This container includes Triton with backends for TensorFlow, ONNX Runtime, and TensorRT.

🔷 Step 2: Set Up the Model Repository

1. **Create a directory structure**

    ```
    mkdir -p ~/triton_models/tf_model/1
    mkdir -p ~/triton_models/onnx_model/1
    ```

    👉 *Why?* Triton expects each model in a dedicated folder with version numbers as subfolders (`1/` for first version).

2. **Obtain TensorFlow Model**
    - Example: ResNet50 saved in **SavedModel format**.

    ```
    wget https://storage.googleapis.com/tfhub-modules/tensorflow/resnet_50/classification/1.tar.gz
    tar -xvzf 1.tar.gz -C ~/triton_models/tf_model/1
    ```

👉 *Why?* SavedModel is the standard TensorFlow format supported natively by Triton.

3. **Obtain ONNX Model**
   - Example: MobileNetV2 exported as ONNX.

```
wget https://github.com/onnx/models/raw/main/vision/classification/mobilenet/model/mobilenetv2-7.onnx -O ~/tri
```

👉 *Why?* ONNX models are portable across frameworks and Triton has a built-in ONNX Runtime backend.

🔷 Step 3: Create Configuration Files

1. **TensorFlow config.pbtxt** (inside `tf_model/`)

```
name: "tf_model"
platform: "tensorflow_savedmodel"
max_batch_size: 8
input [
  { name: "input_tensor:0" data_type: TYPE_FP32 dims: [224,224,3] }
]
output [
  { name: "softmax_tensor:0" data_type: TYPE_FP32 dims: [1000] }
]
```

👉 *Why?* Defines how Triton interprets the TF model's inputs and outputs.

2. **ONNX config.pbtxt** (inside `onnx_model/`)

```
name: "onnx_model"
platform: "onnxruntime_onnx"
max_batch_size: 8
input [
  { name: "input" data_type: TYPE_FP32 dims: [3,224,224] }
]
output [
  { name: "output" data_type: TYPE_FP32 dims: [1000] }
]
```

👉 *Why?* Ensures Triton can correctly feed input tensors into the ONNX model.

🔷 Step 4: Launch Triton with Both Models

Run:

```
docker run --gpus all --rm -it \
  -p8000:8000 -p8001:8001 -p8002:8002 \
  -v ~/triton_models:/models \
  nvcr.io/nvidia/tritonserver:23.12-py3 \
  tritonserver --model-repository=/models
```

- **Ports:**
  - `8000` : HTTP endpoint
  - `8001` : gRPC endpoint
  - `8002` : Metrics (Prometheus)

👉 *Why?* Triton automatically loads both models from the repository and exposes them under different endpoints.

🔷 Step 5: Verify Model Deployment

1. **Check loaded models**

   ```
   curl localhost:8000/v2/health/ready
   curl localhost:8000/v2/models/tf_model
   curl localhost:8000/v2/models/onnx_model
   ```

   👉 *Why?* Confirms Triton successfully loaded both models.

2. **Run inference with TensorFlow model**

   ```
   curl -X POST localhost:8000/v2/models/tf_model/infer \
     -H "Content-Type: application/json" \
     -d @sample_request.json
   ```

3. **Run inference with ONNX model**

   ```
   curl -X POST localhost:8000/v2/models/onnx_model/infer \
     -H "Content-Type: application/json" \
     -d @sample_request.json
   ```

   👉 *Why?* Shows how the same client request format works across frameworks.

🔷 Step 6: Compare and Analyze

- **Check latency differences**: ONNX Runtime vs TensorFlow may have different performance.
- **Batching**: Try sending multiple inputs to see batching benefits.
- **Resource usage**: Monitor with `nvidia-smi` and Prometheus (on port 8002).

👉 *Why?* Analysis demonstrates Triton's ability to unify multi-framework serving while letting you evaluate performance trade-offs.

✅ Lab Wrap-Up

In this lab, you successfully:

- Configured TensorFlow and ONNX models in a Triton repository

- Served both under one unified server
- Queried models through Triton's API
- Learned how multi-framework support simplifies deployment

This exercise demonstrates the **real-world power of Triton**: serving models across different ecosystems seamlessly, at scale, with minimal custom glue code.
