🧪 Lab: Optimize an Inference Pipeline with TensorRT

Goal:
Take a PyTorch vision model, establish a **baseline FP32** deployment, then optimize it with **TensorRT FP16** (and optionally **INT8**) and **Triton Inference Server**. You'll measure latency/throughput and apply server-side optimizations (dynamic batching, multiple instances).

What you'll learn:

- Exporting a PyTorch model to ONNX
- Building TensorRT engines (FP16, optional INT8)
- Serving models with Triton (PyTorch/ONNX vs. TensorRT backends)
- Measuring performance with `perf_analyzer`
- Enabling dynamic batching and instance pooling for higher QPS

---

0) Prereqs & Environment

Hardware/OS: 1× NVIDIA GPU (A100/RTX/etc.), Linux (Ubuntu 20.04+)
Software: Docker + NVIDIA Container Toolkit, `nvidia-smi` works

Containers (pull once):

```
docker pull nvcr.io/nvidia/pytorch:24.03-py3
docker pull nvcr.io/nvidia/tritonserver:24.03-py3
```

---

1) Prepare a Baseline Model (PyTorch → ONNX)

We'll use ResNet50 (ImageNet) as a representative model.

1.1 Export to ONNX (dynamic batch dimension)

```
mkdir -p ~/trt-lab && cd ~/trt-lab

cat > export_onnx.py << 'PY'
import torch
import torchvision.models as models

model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT).eval().cuda()
dummy = torch.randn(1, 3, 224, 224, device='cuda')

torch.onnx.export(
    model, dummy, "resnet50.onnx",
    input_names=["input"], output_names=["logits"],
    opset_version=13, do_constant_folding=True,
    dynamic_axes={"input": {0: "batch"}, "logits": {0: "batch"}}
)
print("Exported resnet50.onnx")
PY

docker run --rm --gpus all -v $PWD:/work -w /work \
  nvcr.io/nvidia/pytorch:24.03-py3 \
  python export_onnx.py
```

Why: ONNX gives a portable graph TensorRT can optimize. Dynamic axes allow multiple batch sizes.

---

2) Baseline: Serve FP32 (ONNX backend) in Triton

2.1 Create a Triton model repo (baseline ONNX)

```bash
mkdir -p model_repository/resnet50_fp32/1
cp resnet50.onnx model_repository/resnet50_fp32/1/model.onnx

cat > model_repository/resnet50_fp32/config.pbtxt << 'PBTXT'
name: "resnet50_fp32"
platform: "onnxruntime_onnx"
max_batch_size: 32
input [
  { name: "input"  data_type: TYPE_FP32 dims: [3, 224, 224] }
]
output [
  { name: "logits" data_type: TYPE_FP32 dims: [1000] }
]
# Enable modest dynamic batching to improve throughput under load
dynamic_batching {
  preferred_batch_size: [ 8, 16, 32 ]
  max_queue_delay_microseconds: 1000
}
# Run multiple instances (if GPU has headroom)
instance_group [{ kind: KIND_GPU, count: 2 }]
PBTXT
```

2.2 Run Triton with baseline model

```bash
docker run --rm --gpus all -p8000:8000 -p8001:8001 -p8002:8002 \
  -v $PWD/model_repository:/models \
  nvcr.io/nvidia/tritonserver:24.03-py3 \
  tritonserver --model-repository=/models
```

Why: This gives us a reference point to compare TensorRT gains.

---

3) Measure Baseline Performance

In a new terminal:

```
docker exec -it $(docker ps --filter ancestor=nvcr.io/nvidia/tritonserver:24.0
  perf_analyzer -m resnet50_fp32 -b 8 -u localhost:8000 -i grpc --concurrency-
```

Try a few settings:

```
perf_analyzer -m resnet50_fp32 -b 1  -i grpc --concurrency-range 1:32
perf_analyzer -m resnet50_fp32 -b 16 -i grpc --concurrency-range 1:16
perf_analyzer -m resnet50_fp32 -b 32 -i grpc --concurrency-range 1:8
```

Record: p50/p90 latency and throughput (infer/s).

Why: You need numbers to prove the improvement later.

---

4) Build a TensorRT FP16 Engine (Fast Win)

We'll use `trtexec` inside the Triton container (it's included) to create a plan that supports dynamic batch sizes.

4.1 Create FP16 engine with min/opt/max shapes

Stop the server (Ctrl+C), then:

```
docker run --rm --gpus all -v $PWD:/work -w /work \
  nvcr.io/nvidia/tritonserver:24.03-py3 \
  trtexec --onnx=resnet50.onnx --saveEngine=resnet50_fp16.plan \
          --fp16 --explicitBatch \
          --minShapes=input:1x3x224x224 \
          --optShapes=input:8x3x224x224 \
          --maxShapes=input:32x3x224x224 \
          --workspace=4096
```

Why:

- `--fp16` enables mixed precision for substantial speedups with minimal accuracy loss.
- The min/opt/max shapes define an **optimization profile** matching our batching strategy.

---

5) Serve the TensorRT FP16 Engine in Triton

5.1 Add a new TensorRT model entry

```
mkdir -p model_repository/resnet50_trt_fp16/1
cp resnet50_fp16.plan model_repository/resnet50_trt_fp16/1/model.plan

cat > model_repository/resnet50_trt_fp16/config.pbtxt << 'PBTXT'
name: "resnet50_trt_fp16"
platform: "tensorrt_plan"
max_batch_size: 32
input [
  { name: "input"  data_type: TYPE_FP32 dims: [3, 224, 224] }
]
output [
  { name: "logits" data_type: TYPE_FP32 dims: [1000] }
]
# Match the profile ranges used in the engine
optimization {
  execution_accelerators {
    gpu_execution_accelerator : [ { name : "tensorrt" } ]
  }
}
dynamic_batching {
  preferred_batch_size: [ 8, 16, 32 ]
  max_queue_delay_microseconds: 800
}
instance_group [{ kind: KIND_GPU, count: 2 }]
PBTXT
```

5.2 Relaunch Triton

```
docker run --rm --gpus all -p8000:8000 -p8001:8001 -p8002:8002 \
  -v $PWD/model_repository:/models \
  nvcr.io/nvidia/tritonserver:24.03-py3 \
  tritonserver --model-repository=/models
```

---

6) Measure FP16 Performance Gains

Run the same perf tests (new terminal):

```
perf_analyzer -m resnet50_trt_fp16 -b 1  -i grpc --concurrency-range 1:32
perf_analyzer -m resnet50_trt_fp16 -b 8  -i grpc --concurrency-range 1:8
perf_analyzer -m resnet50_trt_fp16 -b 16 -i grpc --concurrency-range 1:16
perf_analyzer -m resnet50_trt_fp16 -b 32 -i grpc --concurrency-range 1:8
```

What to expect:

- **Lower latency** (especially p50/p90) vs. FP32
- **Higher throughput** (inf/s), especially at medium/high concurrency
- With `instance_group count: 2`, you should see better concurrency utilization

---

7) Optional: Build an INT8 Engine (Max Speed, Needs Calibration)

INT8 can improve performance further but requires calibration. Below is a minimal **Python TensorRT** workflow that builds an INT8 engine from ONNX using an entropy calibrator. You'll need a small folder of ~100–500 representative images (e.g., `calib/`) for calibration.

7.1 Create INT8 calibrator script

```
cat > build_int8_engine.py << 'PY'
import os, glob, numpy as np, tensorrt as trt, pycuda.driver as cuda, pycuda.a
from PIL import Image

TRT_LOGGER = trt.Logger(trt.Logger.INFO)

class ImageBatchStream:
    def __init__(self, batch_size, calib_dir, shape=(3,224,224)):
        self.batch_size = batch_size
        self.files = glob.glob(os.path.join(calib_dir, '*.jpg')) + glob.glob(c
        self.shape = shape
        self.index = 0

    def reset(self):
        self.index = 0

    def next_batch(self):
        if self.index + self.batch_size > len(self.files):
            return None
        batch_files = self.files[self.index:self.index+self.batch_size]
        self.index += self.batch_size
        batch = []
        for f in batch_files:
            img = Image.open(f).convert('RGB').resize((224,224))
            arr = np.asarray(img).astype(np.float32) / 255.0
            arr = (arr - 0.5)/0.5    # simple normalize to match training
            arr = np.transpose(arr, (2,0,1))  # CHW
            batch.append(arr)
        return np.ascontiguousarray(batch)

class EntropyCalibrator(trt.IInt8EntropyCalibrator2):
    def __init__(self, batchstream, cache_file="calib.cache"):
        super().__init__()
        self.stream = batchstream
        self.d_input = cuda.mem_alloc(trt.volume((batchstream.batch_size, *bat
        self.cache_file = cache_file
        self.stream.reset()

    def get_batch_size(self):
        return self.stream.batch_size

    def get_batch(self, names):
```

```python
        batch = self.stream.next_batch()
        if batch is None:
            return None
        cuda.memcpy_htod(self.d_input, batch)
        return [int(self.d_input)]

    def read_calibration_cache(self):
        if os.path.exists(self.cache_file):
            with open(self.cache_file, "rb") as f:
                return f.read()
        return None

    def write_calibration_cache(self, cache):
        with open(self.cache_file, "wb") as f:
            f.write(cache)

def build_int8_engine(onnx_path, engine_path, calib_dir, batch_size=16):
    builder = trt.Builder(TRT_LOGGER)
    network_flags = 1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
    network = builder.create_network(network_flags)
    parser = trt.OnnxParser(network, TRT_LOGGER)

    with open(onnx_path, "rb") as f:
        assert parser.parse(f.read()), "ONNX parse failed"

    config = builder.create_builder_config()
    config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 4<<30)  # 4GB
    if not builder.platform_has_fast_int8:
        raise RuntimeError("INT8 not supported on this GPU")
    config.set_flag(trt.BuilderFlag.INT8)
    # Also allow FP16 fallback for layers that need it
    if builder.platform_has_fast_fp16:
        config.set_flag(trt.BuilderFlag.FP16)

    # Optimization profile (dynamic batch)
    profile = builder.create_optimization_profile()
    profile.set_shape("input",
                      min=(1,3,224,224),
                      opt=(8,3,224,224),
                      max=(32,3,224,224))
    config.add_optimization_profile(profile)

    calibrator = EntropyCalibrator(ImageBatchStream(batch_size, calib_dir))
```

```
    config.int8_calibrator = calibrator

    engine = builder.build_engine(network, config)
    with open(engine_path, "wb") as f:
        f.write(engine.serialize())
    print("Wrote", engine_path)

if __name__ == "__main__":
    build_int8_engine("resnet50.onnx", "resnet50_int8.plan", calib_dir="calib"
PY
```

Run it (inside a container with TensorRT Python):

```
# Place ~100-500 JPEG/PNG files in ./calib/ beforehand
mkdir -p calib

docker run --rm --gpus all -v $PWD:/work -w /work \
  nvcr.io/nvidia/tritonserver:24.03-py3 \
  python build_int8_engine.py
```

7.2 Serve INT8 in Triton

```
mkdir -p model_repository/resnet50_trt_int8/1
cp resnet50_int8.plan model_repository/resnet50_trt_int8/1/model.plan

cat > model_repository/resnet50_trt_int8/config.pbtxt << 'PBTXT'
name: "resnet50_trt_int8"
platform: "tensorrt_plan"
max_batch_size: 32
input  [{ name: "input"  data_type: TYPE_FP32 dims: [3,224,224] }]
output [{ name: "logits" data_type: TYPE_FP32 dims: [1000] }]
dynamic_batching { preferred_batch_size: [8,16,32] max_queue_delay_microseconc
instance_group [{ kind: KIND_GPU, count: 2 }]
PBTXT
```

Restart Triton (same command as before), then benchmark:

```
perf_analyzer -m resnet50_trt_int8 -b 8 -i grpc --concurrency-range 1:16
```

Note: Always validate accuracy when moving to INT8. If accuracy drops too much, increase/refresh calibration data, consider per-channel quantization, or keep latency-critical layers in FP16.

---

8) Validate Correctness (quick sanity)

Write a tiny client to compare top-1 labels across FP32 vs. TensorRT FP16/INT8 on a handful of images. If top-1 agrees for most examples (or top-5 overlaps), your optimization likely preserved accuracy.

(Pseudo-steps):

- Preprocess 10 images → send to `resnet50_fp32` and `resnet50_trt_fp16`
- Compare argmax of softmax outputs (or compare cosine similarity on logits)

---

9) Turn Server Knobs for More Throughput

Inside each model's `config.pbtxt` , you already set:

- **dynamic_batching** with preferred sizes `[8,16,32]` and small queue delay (≤1 ms)
- **instance_group count: 2** to run multiple model instances per GPU

Why it helps:

- Dynamic batching merges requests to better utilize GPU.
- Multiple instances increase pipeline parallelism when kernels are short.

Experiment: Increase `instance_group count` to 3–4 if there's headroom, and re-run `perf_analyzer`. Watch `nvidia-smi` to avoid VRAM pressure.

---

10) Cleanup

Stop Triton (Ctrl+C), then:

```
rm -rf model_repository resnet50.onnx resnet50_fp16.plan resnet50_int8.plan ca
```

✅ You did it!

- Established a **baseline** (FP32 ONNX/Triton) and **measured** it
- Built and served a **TensorRT FP16** engine with **dynamic shapes**
- (Optional) Built **INT8** with calibration and deployed it
- Tuned **dynamic batching** + **multiple instances** for higher cluster throughput
- Compared performance to quantify the gains