🧪 7.5 Lab: Deploy an AI Model to Jetson Nano

Goal:
Get a Jetson Nano running real-time image classification with **TensorRT** (native), and—optionally—wire it into a simple **DeepStream** pipeline. You'll set power/thermal profiles, build a TensorRT engine from ONNX, run inference, and validate FPS/latency on-device.

What you'll practice: device prep, power tuning, ONNX→TensorRT conversion, Python inference with TensorRT, optional DeepStream, basic perf measurement.

---

0) Prereqs & Lab Topology (what/why)

- **Hardware:** Jetson Nano (4 GB), 5V⎓4A PSU, micro-SD (32 GB+), HDMI, keyboard/mouse, fan (recommended).
- **Software:** JetPack (preloads CUDA/cuDNN/TensorRT), Python 3, OpenCV.
- **Dev workstation (x86)** (optional): to export a model to ONNX and SCP it to the Nano.

Why: JetPack includes the NVIDIA drivers + TensorRT on ARM64 so you can optimize and run models locally without internet access.

---

1) Flash & First-Boot Setup

1. **Flash JetPack SD image** for Nano (on your workstation) and boot the board.
   Why: This gives you CUDA, cuDNN, TensorRT, and the L4T (Ubuntu for Jetson) stack preinstalled.
2. On first boot, finish setup (locale, network). Then update:

```
sudo apt update && sudo apt -y upgrade
```

3. Install basic tools:

```
sudo apt -y install python3-pip python3-venv python3-opencv \
                    nano tmux htop
```

---

2) Power, Clocks & Swap (keep it stable)

1. **Max performance mode & clocks**:

```
sudo nvpmodel -m 0        # 10W mode (Nano devkit)
sudo jetson_clocks        # lock max clocks
```

   Why: Ensures reproducible performance and avoids throttling.

2. **Add swap** (compiles/conversions can spike RAM):

```
sudo fallocate -l 4G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
```

```
echo '/swapfile swap swap defaults 0 0' | sudo tee -a /etc/fstab
sudo swapon -a
```

Why: Prevent OOM while building the TensorRT engine.

3. **Thermals:** make sure a fan/heat-sink is active to avoid throttling.

---

3) Get a Model to the Nano (ONNX)

Option A — Export on your workstation (recommended)

1. Create and run this once on your x86 machine to export **ResNet50**:

```
python3 - <<'PY'
import torch, torchvision as tv
m = tv.models.resnet50(weights=tv.models.ResNet50_Weights.DEFAULT).eval().cuda()
d = torch.randn(1,3,224,224, device='cuda')
torch.onnx.export(m, d, "resnet50.onnx",
                  input_names=["input"], output_names=["logits"],
                  opset_version=13,
                  dynamic_axes={"input":{0:"batch"}, "logits":{0:"batch"}})
print("wrote resnet50.onnx")
PY
```

2. Copy `resnet50.onnx` to the Nano:

```
scp resnet50.onnx ubuntu@<nano_ip>:/home/ubuntu/
```

Option B — Download a small ONNX model on the Nano

- Use any lightweight ONNX classifier (e.g., MobileNet/SqueezeNet) if bandwidth allows.

Why ONNX? TensorRT can parse/optimize ONNX graphs into highly efficient plan engines on the Nano's GPU.

---

4) Build a TensorRT Engine on the Nano

1. Verify TensorRT tools exist:

```
which trtexec   # should resolve to /usr/src/tensorrt/bin/trtexec or in PATH
```

2. Build an FP16 engine (fast win) with dynamic shapes:

```
trtexec --onnx=resnet50.onnx --saveEngine=resnet50_fp16.plan \
        --fp16 --explicitBatch \
        --minShapes=input:1x3x224x224 \
        --optShapes=input:8x3x224x224 \
        --maxShapes=input:32x3x224x224 \
        --workspace=2048
```

Why: FP16 leverages Tensor Cores (where available) and cuts latency. Min/opt/max define your batching envelope.

3. Quick sanity benchmark:

```
trtexec --loadEngine=resnet50_fp16.plan --shapes=input:8x3x224x224 --fp16 --separateProfileRun
```

Why: Confirms the plan loads and gives a baseline throughput on the Nano.

---

5) Python Inference with TensorRT (single image)

1. Install Python deps (OpenCV already present via apt):

```
python3 -m pip install --user numpy pycuda
```

2. Save this script as `infer_trt.py`:

```python
import time, numpy as np, cv2, pycuda.autoinit
import pycuda.driver as cuda
import tensorrt as trt

ENGINE_PATH = "resnet50_fp16.plan"
IMAGE_PATH  = "dog.jpg"           # replace with your test image
INPUT_NAME  = "input"
OUTPUT_NAME = "logits"
H, W = 224, 224

# 1) Build runtime/context
logger = trt.Logger(trt.Logger.INFO)
with open(ENGINE_PATH, "rb") as f, trt.Runtime(logger) as runtime:
    engine = runtime.deserialize_cuda_engine(f.read())
context = engine.create_execution_context()

# 2) Allocate device buffers
def vol(dims): return int(np.prod(dims))
input_idx  = engine.get_binding_index(INPUT_NAME)
output_idx = engine.get_binding_index(OUTPUT_NAME)
input_shape  = (1,3,H,W)
output_shape = (1,1000)

# For dynamic shapes, set binding shape
context.set_binding_shape(input_idx, input_shape)

d_input = cuda.mem_alloc(vol(input_shape)  * np.float32().nbytes)
d_output= cuda.mem_alloc(vol(output_shape) * np.float32().nbytes)
bindings = [None] * engine.num_bindings
bindings[input_idx]  = int(d_input)
bindings[output_idx] = int(d_output)
stream = cuda.Stream()

# 3) Preprocess
img = cv2.imread(IMAGE_PATH)
img = cv2.resize(img, (W,H))
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).astype(np.float32) / 255.0
# Normalize similar to torchvision pretrained
mean = np.array([0.485,0.456,0.406], dtype=np.float32)
std  = np.array([0.229,0.224,0.225], dtype=np.float32)
img = (img - mean) / std
img = np.transpose(img, (2,0,1))[None, ...]       # NCHW
inp = np.ascontiguousarray(img)
```

```
# 4) Inference
h_output = np.empty(output_shape, dtype=np.float32)

# Warmup
for _ in range(5):
    cuda.memcpy_htod_async(d_input, inp, stream)
    context.execute_async_v2(bindings=bindings, stream_handle=stream.handle)
    cuda.memcpy_dtoh_async(h_output, d_output, stream)
    stream.synchronize()

# Timed runs
t0 = time.time()
runs = 50
for _ in range(runs):
    cuda.memcpy_htod_async(d_input, inp, stream)
    context.execute_async_v2(bindings=bindings, stream_handle=stream.handle)
    cuda.memcpy_dtoh_async(h_output, d_output, stream)
    stream.synchronize()
t1 = time.time()

avg_ms = (t1 - t0) * 1000.0 / runs
print(f"avg latency: {avg_ms:.2f} ms")

# 5) Postprocess top-5
probs = np.exp(h_output - h_output.max())  # softmax (numerically stable-ish)
probs /= probs.sum()
top5 = probs.ravel().argsort()[-5:][::-1]
print("top5 class indices:", top5)
```

3. Run it:

```
python3 infer_trt.py
```

Expected: A printed average latency (ms) and top-5 class indices.

Why this matters: You've proven end-to-end: ONNX → TensorRT engine → on-device inference with preprocessing on the Nano.

---

6) Optional: Live camera / video stream with DeepStream

Install DeepStream (often included with JetPack; otherwise install the DeepStream runtime for your JetPack version).

1. Create a minimal **deepstream-app** config `ds_resnet.txt` :

```
[application]
enable-perf-measurement=1

[source0]
enable=1
type=1
uri=file:///home/ubuntu/video.mp4        # or use v4l2 camera: type=1 + camera device
num-sources=1
gpu-id=0

[streammux]
width=1280
height=720
batch-size=1
batched-push-timeout=40000
```

```
live-source=1

[primary-gie]
enable=1
gpu-id=0
batch-size=1
model-engine-file=/home/ubuntu/resnet50_fp16.plan
network-mode=2             # 0:FP32, 1:INT8, 2:FP16
num-detected-classes=1000
interval=0

[sink0]
enable=1
type=3                     # 3 = fakesink (headless). Use type=2 for screen display
sync=0
```

2. Run:

```
deepstream-app -c ds_resnet.txt
```

Why: DeepStream uses hardware-accelerated decode + TensorRT inference and shows FPS in logs; ideal for video use-cases.

---

7) Measure & Tune Performance

1. **System monitor**:

```
tegrastats
```

Why: Realtime view of GPU load, RAM, thermals; watch for throttling.

2. **Batch & precision:**
   - Try `--shapes=input:4x3x224x224` in `trtexec` and update the Python script to send 4 images per enqueue.
   - Compare FP16 vs FP32 by rebuilding the plan without `--fp16`.
3. **I/O cost:**
   Convert images to **NCHW FP16** ahead of time or pin memory (PyCUDA) to reduce transfer overhead.
4. **Thermals/power:**
   If clocks drop or temperatures spike, improve cooling or reduce clocks slightly for sustained throughput.

---

8) (Optional) INT8 for extra speed

1. Prepare a small calibration set (100–500 images) on the Nano.
2. Build an INT8 plan (requires calibrator—tooling/scripts vary).
3. Validate accuracy vs FP16; if drop is unacceptable, increase calibration data or keep critical layers in FP16.

Why: INT8 can further reduce latency on constrained devices, but needs careful calibration to preserve accuracy.

---

9) Troubleshooting Cheatsheet

- **“trtexec not found”:** ensure TensorRT is installed with JetPack; check `/usr/src/tensorrt/bin`.
- **Engine won’t load:** rebuild the plan on the **same device**/JetPack as where it will run; mismatched TensorRT versions can fail.
- **OOM during build:** keep swap enabled; use smaller batch sizes; lower `--workspace` value.
- **Low FPS:** verify `sudo nvpmodel -m 0` and `sudo jetson_clocks`; check tegrastats for throttling; use FP16; reduce input resolution.
- **Camera not found:** ensure v4l2 device exists (`/dev/video0`) and user is in `video` group.

---

11) Cleanup

```
rm -f resnet50.onnx resnet50_fp16.plan infer_trt.py ds_resnet.txt
sudo swapoff /swapfile && sudo sed -i '/\/swapfile/d' /etc/fstab && sudo rm -f /swapfile
```

✅ You’ve deployed an AI model on Jetson Nano

- Prepped the device for reliable perf (power, clocks, swap).
- Converted an **ONNX** model to a **TensorRT** engine and served it via Python.
- (Optionally) ran a **DeepStream** video pipeline using your engine.
- Measured latency/FPS and tuned for better throughput.