Lab 5: Real-time Inference of Deep Learning models on Edge Device

1. Why is it recommended to use a virtual environment named dlonedge for this lab?

Answer:
Because installing specialized libraries like **PyTorch**, **torchvision**, and **OpenCV** in a separate environment avoids version conflicts with other projects. It also ensures that any system-wide library updates or changes do not break the lab setup.

---

2. Which Python libraries are installed to enable deep learning and image processing on the Raspberry Pi?

Answer:

- **torch**, **torchvision**, and **torchaudio** (for PyTorch-based deep learning)
- **opencv-python** (for video capture and image processing)
- **numpy** (for efficient numerical operations and array handling)

---

3. What is MobileNetV2, and why might it be chosen for edge device deployment?

Answer:
MobileNetV2 is a **lightweight convolutional neural network** designed for resource-constrained environments. It uses **depthwise separable convolutions** to reduce the number of parameters and computational load, making it well-suited for real-time inference on edge devices like the Raspberry Pi.

---

4. How does the sample code measure and report frames per second (FPS)?

Answer:
The code increments a frame_count every time a frame is processed. It then checks the elapsed time (e.g., using time.time()) to compute:

ini

Copy

fps = frame_count / (now - last_logged)

It prints the FPS every second, resetting frame_count and last_logged for the next interval.

---

5. What is quantization in deep learning, and why is it useful on a Raspberry Pi?

Answer:
Quantization reduces the precision of model parameters and activations (e.g., from 32-bit floats to 8-bit integers). This lowers **memory usage**, speeds up **inference**, and makes models more efficient on hardware with limited computational resources, such as the Raspberry Pi.

---

6. In the sample code, how do you switch between a floating-point MobileNetV2 and a quantized MobileNetV2?

Answer:
By toggling the boolean variable:

python

Copy

quantize = True

If quantize is set to True, the code loads a **quantized** MobileNetV2 via:

python

Copy

models.quantization.mobilenet_v2(pretrained=True, quantize=True)

Otherwise, it loads the standard floating-point version.

---

7. What is the purpose of setting:

python

Copy

torch.backends.quantized.engine = 'qnnpack'

when using quantized models?

Answer:
It specifies the **quantization engine** to be used by PyTorch. **QNNPACK** is optimized for mobile/edge devices and can provide faster integer arithmetic performance on CPUs lacking advanced vector instructions (like AVX2 or NEON).

---

8. Why is the camera capture resolution set to 224×224 and FPS set to 36 in the sample code?

Answer:
MobileNetV2 typically expects 224×224 input images, so capturing at that resolution avoids additional resizing overhead. The code requests 36 FPS to ensure enough frames are available, aiming for a final **effective** 30 FPS after pre-processing and inference overhead.

---

9. What happens when you uncomment lines 57-61 in the sample code (the softmax block)?

Answer:
It prints the **top 10 class predictions** from the model in real time. The code enumerates the model's output, applies softmax(dim=0), sorts by confidence scores, and displays the percentage and label for the top 10 categories.

---

10. What are the two main quantization approaches discussed in the lab, and how do they differ?

Answer:

1. **Post-Training Quantization (PTQ):**
    - Converts a **fully trained** floating-point model to an 8-bit quantized model.
    - Straightforward to apply but can cause an **accuracy drop** if the model is sensitive to reduced precision.
2. **Quantization-Aware Training (QAT):**
    - Introduces "fake quantization" operators **during training**, letting the model learn to be robust under quantized conditions.
    - More complex but typically preserves **higher accuracy** than PTQ.

---

11. How does quantization typically improve the model's performance on the Raspberry Pi?

Answer:
By using **int8** arithmetic instead of **float32**, the model requires fewer bits for storage and fewer CPU cycles per operation. This often yields:

- **Reduced memory footprint** (model size).

- **Increased throughput** (more inferences per second).
- **Lower power consumption**.

---

12. What trade-offs might you encounter when applying quantization to a model?

Answer:

- **Accuracy Loss:** The reduced precision can cause minor to moderate drops in accuracy, depending on the model.
- **Compatibility Issues:** Not all layers or operations are supported by quantized kernels, potentially limiting which models can be quantized.
- **Retraining Needs:** QAT requires additional training resources and time, though it often yields better accuracy than PTQ.

---

13. Why might you see a large performance jump from ~5-6 FPS to nearly 30 FPS after enabling quantization?

Answer:
The quantized model runs significantly faster because integer operations are cheaper than floating-point operations on the Raspberry Pi's CPU. The model also requires less memory bandwidth, further reducing bottlenecks and enabling near-real-time inference.

---

14. What is the advantage of setting with torch.no_grad(): when performing inference?

Answer:
It disables the gradient-tracking mechanism, which is only needed for training. This saves memory and speeds up computations during inference, as PyTorch does not store intermediate gradients.

---

15. How could you further optimize real-time inference beyond quantization?

Answer:

- **Model Pruning:** Remove redundant weights or entire filters.
- **Neural Architecture Search:** Find a smaller architecture well-suited for the task.

- **Hardware Acceleration:** Use specialized accelerators (e.g., Google Coral TPU, NVIDIA Jetson) if available.
- **Batching or Pipeline Optimizations:** Minimize overheads between capturing frames, pre-processing, and model execution.

---

16. What is the purpose of the optional exercises, such as running a quantized large language model on Raspberry Pi?

Answer:
They encourage exploration of more **complex models** and advanced quantization techniques. By testing large language models or bigger CNNs, participants can see how quantization drastically impacts performance, memory usage, and how the Raspberry Pi handles heavier tasks.

---

17. What is the difference between static quantization and dynamic quantization in PyTorch?

Answer:

- **Static Quantization:**
    - Applies quantization parameters (scale and zero-point) that are **calibrated** ahead of inference time, often using representative data.
    - Both **weights and activations** are quantized.
    - Typically yields better performance on smaller devices but requires a calibration step.
- **Dynamic Quantization:**
    - Quantizes weights ahead of time but calculates activations' scale/zero-point **on the fly** (i.e., dynamically) during inference.
    - Commonly used for models with large fully connected layers (like LLMs), especially for text.
    - Easier to apply but may not provide as large a speedup as static quantization.

---

18. Why might setting cap.set(cv2.CAP_PROP_FPS, 36) not always guarantee a true 36 FPS in real-world scenarios?

Answer:
Because cv2.CAP_PROP_FPS is a **request** to the camera driver rather than a hard mandate. The actual achievable FPS depends on factors like:

- Camera hardware capabilities.
- The CPU load from **preprocessing** and **model inference**.
- Operating system scheduling and other resource constraints.
  In practice, if the system cannot process frames fast enough, the real FPS will be lower than 36.

---

19. How can you measure memory usage of your PyTorch model on the Raspberry Pi?

Answer:

- Use **torch.cuda.memory_allocated()** or **torch.cuda.memory_reserved()** if you have a GPU (though typically not on a standard Pi).
- For CPU usage, you can rely on **system-level tools** like htop, free -h, or psutil in Python.
- You can also measure the size of the model's state dict (.pth file) on disk, which indicates the approximate memory usage for weights.

---

20. Why is preprocess = transforms.Compose([...]) essential before passing frames to the MobileNetV2 model?

Answer:
Because the model expects **normalized** tensors in a specific shape (e.g., [Batch, Channels, Height, Width]). The transforms:

- Convert the frame from NumPy arrays (H×W×C) to PyTorch tensors (C×H×W).
- Normalize pixel intensities based on the model's training mean and standard deviation.
- Ensure consistent input format that matches the pretrained model's expectations.

---

21. If you enable quantization but see no improvement in FPS, what might be the reasons?

Answer:

- The CPU architecture or OS might **lack** optimized integer operations or QNNPACK support.
- The overhead of capturing and preprocessing frames could be the **bottleneck**, rather than the model's computation.
- The model might contain layers that are not fully quantized or supported by the backend, limiting speedup.
- You may need to re-calibrate or re-train the model for better quantization support.

---

22. In the sample code, what does softmax(dim=0) do in the top-10 predictions block?

Answer:
It converts raw model outputs (logits) into **probability-like scores** across the output dimension (dim=0). Each index in the output vector represents a class, and after softmax, their values sum to 1. Sorting these probabilities lets you see which classes the model thinks are most likely.

---

23. Can you use GPUs or hardware accelerators for quantized models on the Raspberry Pi?

Answer:

- By default, the standard Raspberry Pi does **not** have a dedicated GPU for general-purpose computations (only a VideoCore for video).
- However, you can attach external accelerators (e.g., Google Coral TPU, Intel Movidius NCS) that support quantized operations.
- PyTorch on Raspberry Pi typically uses the CPU unless you have a specialized board like the NVIDIA Jetson (which is a different platform).

---

24. What might cause accuracy drops when switching from a floating-point model to a quantized model?

Answer:

- Reduced **numerical precision** (8-bit vs. 32-bit) can introduce rounding errors, especially for layers sensitive to small changes in weights or activations.
- Certain operations or layers may be more prone to quantization-induced **saturation** or **clamping**.
- If **calibration** or **fake quantization** steps are insufficient, the model might not adapt well to the reduced range.

---

25. Beyond quantization, name two other methods to optimize a PyTorch model for edge deployment.

Answer:

1. **Model Pruning:** Remove unnecessary connections or entire filters, lowering model size and compute cost.
2. **Knowledge Distillation:** Train a smaller "student" model to mimic the outputs of a larger "teacher" model, preserving accuracy in a lighter architecture.

---

26. How can you confirm that PyTorch is indeed using the quantized model during inference?

Answer:

- Inspect the model layers: a quantized model typically shows **quantized** modules (e.g., QuantizedConv2d) instead of standard Conv2d.
- Print or log the **model architecture**.
- Check the PyTorch quantization engine setting (qnnpack or fbgemm) and verify it's recognized at runtime.
- Observe the significantly **reduced** model size and higher FPS if everything is configured correctly.