

hi i am a university student taking a module on "edge computing and analytics". i need your help guiding me through a lab. the lab is on "Real-time Inference of Deep Learning models on Edge Device"

Memory updated

Hey YR! I'd be happy to guide you through it. Could you share what you've been given for the lab?

For example:

- Objectives of the lab
- Any setup or hardware (e.g., Raspberry Pi, NVIDIA Jetson, webcam, etc.)
- Instructions or code snippets provided
- Where you're currently stuck or unsure

Once I see that, we can go step by step together!



okay i will be providing you the lab sheet in sections

Perfect — send it over in chunks and I'll help you through each part. Just let me know when you're ready to begin!

Real-time Inference of Deep Learning models on Edge Device

Objective: By the end of this session, participants will understand

1. How to run a deep learning model on an edge device (aka raspberryPi)
2. How does quantization work and different quantization methods for deep learning models

Prerequisites:

1. Raspberry Pi with Raspbian OS installed.
2. MicroSD card (16GB or more recommended).
3. Web camera compatible with Raspberry Pi.
4. Internet connectivity (Wi-Fi or Ethernet).
5. Basic knowledge of Python and Linux commands.

1. Introduction

Edge analytics with real-time processing capabilities is challenging but important and inevitable due to privacy/security concerns. However, edge devices like RaspberryPi are constrained with limited hardware resources, which at times are not sufficient to run complex deep learning models. These models require lot of computational resource and memory due to their size and complex architecture. Therefore, in such scenarios, we optimize the model such that it can run efficiently with reduced inference time critical for real-time analytics. Optimization can be achieved by combination of techniques like quantization and converting trained model into architecture specific lite model.

Thanks for sharing that section! Here's a quick breakdown of what this lab is about and what you'll need:

Goals of the Lab

By the end, you should be able to:

1. Run a deep learning model on a Raspberry Pi (edge device).
 2. Understand **quantization** — a technique to reduce model size and make inference faster.
-

What You Need (Prerequisites)

- A Raspberry Pi (with Raspbian OS).
 - A microSD card (16GB or more).
 - A webcam that works with Raspberry Pi.
 - Internet access.
 - Basic Python and Linux command knowledge.
-

Introduction Summary

- Running deep models on edge devices is hard due to limited compute/memory.
 - Edge devices are useful for **privacy/security** but lack resources.
 - Solution: **Model optimization**, using:
 - **Quantization** (reduces model size and precision to speed up inference)
 - **Model conversion** for hardware compatibility
-

When you're ready, go ahead and send the next part of the lab. We'll go step by step and I can help explain or guide the setup and coding parts too.

2. Running Deep Learning Model On RaspberryPi

- This section guide you on how to setup a Raspberry Pi for running PyTorch and deploy a MobileNet v2 image classification model in real time on the CPU.
 - Set up and activate a virtual environment named "dلونedge" for this experiment (to avoid conflicts in libraries) as below:
- ```
sudo apt install python3-venv
python3 -m venv dلونedge
source dلونedge/bin/activate
```
- Installing PyTorch and OpenCV:
- ```
pip install torch torchvision torchaudio
pip install opencv-python
pip install numpy --upgrade
```
- Same as last lab, for video capture we're going to be using OpenCV to stream the video frames. The model we are going to use in this lab is MobileNetV2, which takes in image sizes of 224x224. We are targeting 30fps for the model but we will request a slightly higher framerate of 36 fps than that so there is always enough frames and bandwidth of image pre-processing and model prediction.

Great — this section walks you through setting up your Raspberry Pi for **real-time image classification** using **MobileNet v2**. Here's a breakdown with a few extra tips along the way:

🔧 Step-by-Step: Running Deep Learning Model on Raspberry Pi

✓ 1. Create and Activate a Virtual Environment

Keeps your project's dependencies isolated.

```
bash
```

Copy Edit

```
sudo apt install python3-venv      # Installs the venv package
python3 -m venv dلونedge          # Creates a virtual environment named
'dلونedge'
source dلونedge/bin/activate       # Activates the environment
```

✓ Tip: If you're on a fresh Pi, make sure Python 3 is installed (`python3 --version`) and update your system with `sudo apt update && sudo apt upgrade` .

✓ 2. Install Required Python Packages

Inside the activated `dلونedge` environment:

```
bash
```

Copy Edit

```
pip install torch torchvision torchaudio  
pip install opencv-python  
pip install numpy --upgrade
```

⚠️ **Important:** Depending on your Pi model (e.g., Pi 3 vs Pi 4), PyTorch installation might be tricky. If pip install torch fails or is too slow, you may need to install a pre-built wheel optimized for ARM (I can help you with that if needed).

⌚ 3. Video Capture Notes

- You're using **OpenCV** to stream video.
- The model is **MobileNetV2** (lightweight, good for edge).
- Input image size: 224x224 .
- Target framerate: **36 fps** (higher than 30 fps to ensure smooth capture + prediction).

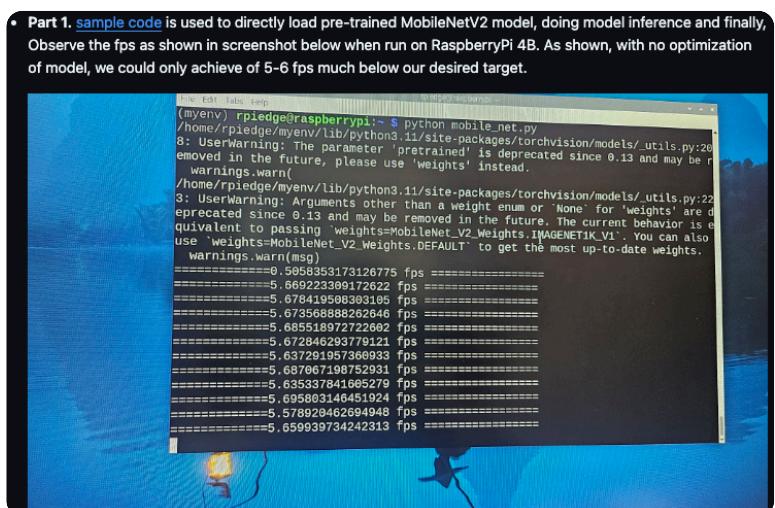
🎥 Make sure your webcam is working. Test with:

bash

Copy Edit

```
python3 -c "import cv2; cap=cv2.VideoCapture(0); print(cap.isOpened())"
```

Send over the next part when you're ready — probably setting up the model or the real-time inference code!



```
sample code:  
import time  
  
import torch  
import numpy as np  
from torchvision import models, transforms  
from torchvision.models.quantization import  
MobileNet_V2_QuantizedWeights  
  
import cv2  
from PIL import Image  
  
quantize = False  
  
if quantize:  
    torch.backends.quantized.engine = 'qnnpack'  
  
cap = cv2.VideoCapture(0)  
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 224)  
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 224)  
cap.set(cv2.CAP_PROP_FPS, 36)  
  
preprocess = transforms.Compose([  
    transforms.ToTensor(),  
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=  
[0.229, 0.224, 0.225]),  
])  
  
weights = MobileNet_V2_QuantizedWeights.DEFAULT  
classes = weights.meta["categories"]  
net = models.quantization.mobilenet_v2(pretrained=True,  
quantize=quantize)  
  
started = time.time()  
last_logged = time.time()  
frame_count = 0  
  
with torch.no_grad():  
    while True:  
        # read frame  
        ret, image = cap.read()  
        if not ret:  
            raise RuntimeError("failed to read frame")  
  
        # convert opencv output from BGR to RGB
```

```
image = image[:, :, [2, 1, 0]]  
permuted = image
```

```
# preprocess  
input_tensor = preprocess(image)
```

```
# create a mini-batch as expected by the model  
input_batch = input_tensor.unsqueeze(0)
```

```
# run model  
output = net(input_batch)
```

```
# Uncomment below 5 lines to print top 10 predictions  
#top = list(enumerate(output[0].softmax(dim=0)))  
#top.sort(key=lambda x: x[1], reverse=True)  
#for idx, val in top[:10]:  
#    print(f"{val.item() * 100:.2f}% {classes[idx]}")
```

```
#print(f"=====-----  
=====-----")
```

```
# log model performance  
frame_count += 1  
now = time.time()  
if now - last_logged > 1:  
    print(f"===== {frame_count / (now -  
last_logged)} fps =====")  
    last_logged = now  
    frame_count = 0
```

Awesome, you're doing great so far — this section demonstrates **basic real-time inference using MobileNetV2** on a Raspberry Pi **without quantization**.

Let's break this down:

💡 What's Happening in This Code

1. Quantization Toggle

```
python
```

Copy

Edit

```
quantize = False
```

This disables quantization for now. If you flip this to `True`, the model uses reduced precision (`int8`) for faster inference.

2. Model Loading

```
python
```

 Copy

 Edit

```
weights = MobileNet_V2_QuantizedWeights.DEFAULT  
net = models.quantization.mobilenet_v2(pretrained=True, quantize=quantize)
```

- Even though it's the quantized version of MobileNet, it runs in full precision when `quantize=False`.

3. Camera Setup with OpenCV

- Frame size: 224×224 (MobileNetV2 input size)
- Target FPS: 36

4. Preprocessing (TorchVision style):

- Converts image to tensor.
- Normalizes with ImageNet mean and std.

5. Inference Loop

- Captures a frame from webcam
- Converts it from BGR \rightarrow RGB
- Preprocesses and feeds it to the model
- Logs FPS every second

Why It's Slow (5–6 FPS)

You're seeing ~5–6 fps on a **Raspberry Pi 4B**, which is expected because:

- It's running full precision MobileNetV2 on CPU.
- No quantization = larger model, slower inference.
- PyTorch is relatively heavy for edge devices when unoptimized.

✓ What's Next (Likely in Your Lab)

- Enable quantization (`quantize = True`) to see performance boost.
 - Maybe compare output accuracy vs speed tradeoff.
 - Possibly convert the model using **TorchScript**, **ONNX**, or **TFLite** for even faster results.
-

Let me know when you're ready to move on — I can walk you through enabling quantization or optimizing further!

can you explain the code to me in detail?

Absolutely! Let's go line by line and break down what's going on in the code you were given.

🧠 Full Code Explanation: Real-time Inference with MobileNetV2

📦 Imports

```
python
```

Copy Edit

```
import time
import torch
import numpy as np
from torchvision import models, transforms
from torchvision.models.quantization import MobileNet_V2_QuantizedWeights

import cv2
from PIL import Image
```

What they do:

- `time` → to measure how long each frame takes (for FPS calculation).
- `torch` → PyTorch library for loading the model and doing inference.

- `numpy` → array operations (not directly used here but often needed).
 - `torchvision.models` → contains pretrained models like MobileNetV2.
 - `torchvision.transforms` → for preprocessing images.
 - `cv2` → OpenCV, used for camera input and image processing.
 - `PIL.Image` → not used directly here but commonly used for image manipulation.
-

✍ Quantization Setup Toggle

```
python                                ⚒ Copy ⚒ Edit

quantize = False

if quantize:
    torch.backends.quantized.engine = 'qnnpack'
```

- `quantize = False` : turns off quantization.
 - If you set it to `True` , it enables **QNNPACK**, a backend optimized for quantized models.
-

🎥 Video Capture Setup

```
python                                ⚒ Copy ⚒ Edit

cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 224)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 224)
cap.set(cv2.CAP_PROP_FPS, 36)
```

- `VideoCapture(0)` : initializes your default webcam.
 - Set video frame size to `224x224` (MobileNetV2's input).
 - Request a frame rate of 36 fps.
-

📸 Image Preprocessing Pipeline

python

 Copy  Edit

```
preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]),
])
```

- Converts webcam frames to PyTorch tensor.
- Normalizes them using ImageNet stats — **this is crucial** because MobileNetV2 was trained on ImageNet and expects input in this format.

Load the Pretrained Model

python

 Copy  Edit

```
weights = MobileNet_V2_QuantizedWeights.DEFAULT
classes = weights.meta["categories"]
net = models.quantization.mobilenet_v2(pretrained=True, quantize=quantize)
```

- Loads the pretrained MobileNetV2 model.
- If `quantize=True`, a quantized version of the model is loaded.
- `classes` contains the list of 1000 ImageNet class names.

Timing Setup

python

 Copy  Edit

```
started = time.time()
last_logged = time.time()
frame_count = 0
```

Used for tracking the number of frames processed and calculating frames per second (FPS).

Real-time Inference Loop

python

 Copy  Edit

```
with torch.no_grad():
    while True:
        # read frame
        ret, image = cap.read()
        if not ret:
            raise RuntimeError("failed to read frame")
```

- Turns off gradient computation (`torch.no_grad()`), which saves memory and speeds up inference.
- Reads a frame from the webcam.
- If the camera fails, it raises an error.

Convert BGR to RGB

python

 Copy  Edit

```
image = image[:, :, [2, 1, 0]]
permuted = image
```

- OpenCV captures images in BGR format.
- This line converts it to RGB, which is what PyTorch models expect.

Preprocess and Batch

python

 Copy  Edit

```
input_tensor = preprocess(image)
input_batch = input_tensor.unsqueeze(0)
```

- Applies preprocessing pipeline to the image.
- Adds a batch dimension (because models expect a batch of images, even if it's only 1).

Run the Model

python

 Copy  Edit

```
output = net(input_batch)
```

- Runs inference on the preprocessed image.
- Output is a tensor of shape [1, 1000] — probabilities for 1000 ImageNet classes.

🔍 (Optional) Show Top 10 Predictions

python

 Copy  Edit

```
#top = list(enumerate(output[0].softmax(dim=0)))
#top.sort(key=lambda x: x[1], reverse=True)
#for idx, val in top[:10]:
#    print(f'{val.item()*100:.2f}% {classes[idx]}')
```

- Softmax converts raw scores into probabilities.
- Sorts and prints the top 10 class predictions.

(This part is commented out by default.)

📊 Calculate FPS

python

 Copy  Edit

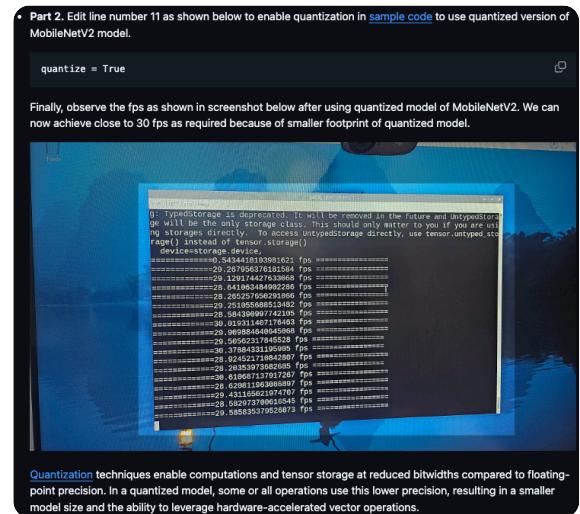
```
frame_count += 1
now = time.time()
if now - last_logged > 1:
    print(f'===== {frame_count / (now - last_logged)} fps
=====')
    last_logged = now
    frame_count = 0
```

- Keeps track of how many frames are processed every second.
 - Prints FPS every second.

What You Can Do Next

- Set `quantize = True` and rerun to see speedup.
 - Uncomment the top-10 predictions to see model output.
 - Log predictions alongside FPS for analysis.

Let me know if you want to move on to the quantized version or if you want help optimizing this code further!



Perfect — this part demonstrates **how enabling quantization significantly improves performance** on a resource-limited edge device like Raspberry Pi.

What Changed?

Edit This Line:

python

 Copy

 Edit

```
quantize = True
```

That one line switches the model from a full-precision (float32) version to a **quantized** version (typically int8), optimized for speed and efficiency.

Observed Performance Boost

As seen in the screenshot:

- Before quantization: ~5–6 FPS
- After quantization: **~28–30 FPS** 🎉

This matches your goal of real-time (~30 FPS) inference on the edge device.

Why Is This Faster?

Quantization = Smaller + Faster Model

- Replaces 32-bit floats with 8-bit integers for:
 - Model weights
 - Activations (intermediate layer outputs)
- This drastically reduces:
 - **Memory usage**
 - **Computation cost**
 - **Latency**

PyTorch Quantization Backend (qnnpack)

- QNNPACK is a backend optimized for ARM CPUs like Raspberry Pi.
 - It performs **vectorized** operations efficiently on int8 tensors.
-

Final Notes on Quantization (from the lab sheet)

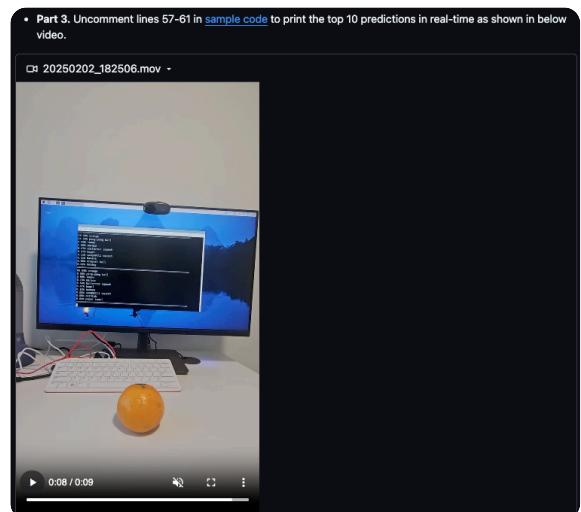
“In a quantized model, some or all operations use reduced precision, resulting in a smaller model size and the ability to leverage hardware-accelerated vector operations.”

Yup — that’s exactly what’s happening. ☺

Would you like help with:

- Understanding how quantized models are trained?
- Visualizing predictions in real time (e.g. drawing class names on the video)?
- Exporting/packaging the model for deployment?

Let’s go deeper if you’re curious!



Great! This final part is all about **displaying the model’s top 10 predictions in real time**.

💡 What to Do

You just need to **uncomment lines 57–61** in the code — these lines:

```
python Copy Edit
top = list(enumerate(output[0].softmax(dim=0)))
top.sort(key=lambda x: x[1], reverse=True)
for idx, val in top[:10]:
    print(f'{val.item() * 100:.2f}% {classes[idx]}')
```

```
print(f"=====  
==")
```

🧠 What This Code Does

1. Apply Softmax

python

 Copy  Edit

```
output[0].softmax(dim=0)
```

This converts the raw model output (logits) into **probability scores** that sum to 100%.

📊 2. Sort by Confidence

python

 Copy  Edit

```
top.sort(key=lambda x: x[1], reverse=True)
```

This sorts all 1000 class predictions by their probability, descending.

TOP 3. Print Top 10

python

 Copy  Edit

```
for idx, val in top[:10]:  
    print(f"{val.item()*100:.2f}% {classes[idx]}")
```

Prints out the top 10 predictions with their confidence percentages.

🖼 4. Sample Output

If you point the camera at an orange, you might see:

65.23% orange

12.47% lemon

...

💡 Tip: Want to display predictions on screen instead of terminal?

If you're interested, I can help you modify the code to **draw predictions directly onto the video frame using OpenCV** — this would make it a bit more visual like a real application.

Let me know if you want to try that or if you're moving on to model optimization/deployment!

3. Quantization using Pytorch

- Neural networks typically use 32-bit floating point precision for activations, weights, and computations. Quantization reduces this precision to smaller data types (like 8-bit integers), decreasing memory and speeding up computation. This compression is not lossless, as lower precision sacrifices dynamic range and resolution. Thus, a balance must be struck between model accuracy and the efficiency gains from quantization.
- In this section, we would learn how to use Pytorch to perform different quantization methods on a neural network architecture. There are two ways in general to quantize a deep learning model:
 - i. Post Training Quantization: After we have a trained model, we can convert the model to a quantized model by converting 32 bit floating point weights and activations to 8 bit integer, but we may see some accuracy loss for some types of models.
 - ii. Quantization Aware Training: During training, we insert fake quantization operators into the model to simulate the quantization behavior and convert the model to a quantized model after training based on the model with fake quantize operators. This is harder to apply than post-training quantization since it requires retraining the model, but typically gives better accuracy.

Great — now you're diving into the **theory of quantization in PyTorch**, which is crucial for optimizing deep learning models on edge devices like the Raspberry Pi.

Let's break it down.

🧠 What is Quantization?

In neural networks:

- We typically use **32-bit floating point (float32)** for:
 - Activations
 - Weights
 - Intermediate computations

Quantization **reduces this precision**, often to **8-bit integers (int8)**, which:

-  Uses less memory
-  Increases inference speed
-  Reduces power consumption

 But this **isn't lossless** — you sacrifice some precision, which might hurt accuracy a little. That's the trade-off.

Quantization Methods in PyTorch

PyTorch offers two main techniques:

1. Post-Training Quantization (PTQ)

 *Simpler to implement, but might slightly reduce accuracy.*

text

 Copy

 Edit

```
Train model → Convert to int8
```

- You take an already-trained float32 model.
- Convert weights and activations to int8.
- No retraining required.

Used in your lab code:

python

 Copy

 Edit

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

2. Quantization-Aware Training (QAT)

 *More complex, but typically higher accuracy.*

text

 Copy

 Edit

During training → Simulate quantization using "fake quantization ops" → Fine-tune model

- Insert "fake quantization" modules during training.
- Model learns how to work with quantized weights/activations.
- After training, convert it to a real quantized model.

Used when:

- You need **better accuracy** on quantized models.
- You're training a custom model, not using off-the-shelf pretrained ones.

vs PTQ vs QAT Summary

Feature	Post-Training Quantization	Quantization-Aware Training
Requires retraining	✗ No	✓ Yes
Accuracy	⚠ Might drop	✓ Often better
Complexity	● Simple	● Harder
Use case	Pretrained models	Custom training pipelines

Let me know if you'd like to try building your own PTQ or QAT flow, or if you're curious about converting to formats like **TorchScript**, **ONNX**, or **TFLite** for better deployment options on edge!

• Please refer to [sample jupyter notebook file](#), which demonstrates how to quantized a pre-trained model using post-training quantization approach as well as quantization aware training. Please run the sample code preferably in google colab if you do not have computer with good hardware specs.

i will be sending you the jupyter notebook file to you in sections. i want you to help me understand it.

Perfect! Go ahead and send the Jupyter notebook content in sections, and I'll walk you through each part, explaining what it's doing and why it matters — whether it's post-training quantization, quantization-aware training, or anything in between.

▼ Quantization tutorial

This tutorial shows how to do post-training static quantization, as well as illustrating two more advanced techniques - per-channel quantization and quantization-aware training - to further improve the model's accuracy. The task is to classify MNIST digits with a simple LeNet architecture.

This is a minimalist tutorial to show you a starting point for quantisation in PyTorch. For theory and more in-depth explanations, Please check out: [Quantizing deep convolutional networks for efficient inference: A whitepaper](#).

The tutorial is heavily adapted from: https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html

Awesome — this is the **intro section** of your Jupyter notebook for the quantization lab. Here's a breakdown of what's being said:

Quantization Tutorial Summary

Goal

You're going to learn how to:

1. Post-Training Static Quantization (PTQ)
2. Per-Channel Quantization
3. Quantization-Aware Training (QAT)

All of this will be applied to a simple **LeNet model** on the **MNIST dataset** (handwritten digits).

Model

- **LeNet**: A lightweight CNN — great for experimentation.
- **Task**: Classify handwritten digits (0–9) from the MNIST dataset.

What's Covered

Technique	Description
PTQ	Quantize a pretrained model without retraining
Per-Channel Quantization	Allows more fine-grained weight quantization

Technique	Description
QAT	Simulates quantization during training for better final accuracy

📖 Further Reading

- The tutorial references:
 - [PyTorch official static quantization tutorial](#)
 - [Whitepaper on quantized CNNs](#)

These are great if you want deeper theoretical understanding later.

This section is just setting the stage. Go ahead and send the next cell or section — likely loading data or defining the model!

⌄ Initial Setup

Before beginning the assignment, we import the MNIST dataset, and train a simple convolutional neural network (CNN) to classify it.

```
[ ] !pip3 install torch==1.5.0 torchvision==1.6.0
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import os
from torch.utils.data import DataLoader
import torch.quantization
from torch.quantization import QuantStub, DeQuantStub
```

Nice — this section sets up your environment for training and quantizing a model. Here's what's happening step by step:

⚙️ Initial Setup Overview

1. Installing Specific Versions of PyTorch & TorchVision

python

Copy

Edit

```
!pip3 install torch==1.5.0 torchvision==0.6.0
```

- These older versions are used because quantization APIs were still evolving, and tutorials like this one often target versions where things are stable and predictable.
- In Google Colab, running this will **restart the Python runtime**, so only do it once.

 If you're using a newer version, APIs might differ slightly (I can help adjust if needed).

2. Imports

Standard PyTorch Imports

```
python Copy Edit  
  
import torch  
import torchvision  
import torchvision.transforms as transforms  
import torch.nn as nn  
import torch.nn.functional as F  
import torch.optim as optim
```

- `torch.nn` , `F` : For building and using neural networks.
- `transforms` : Preprocessing transforms for MNIST (e.g. normalize, convert to tensor).
- `optim` : Training optimizers like SGD/Adam.

Datasets & Data Loaders

```
python Copy Edit  
  
from torch.utils.data import DataLoader
```

- Helps batch and shuffle data during training.

Quantization-Specific Tools

```
python Copy Edit
```

```
import torch.quantization
from torch.quantization import QuantStub, DeQuantStub
```

- **QuantStub** : Marks where quantization starts in the model.
- **DeQuantStub** : Marks where dequantization should happen (before outputs like softmax or loss).

These are key building blocks in PyTorch quantization — you'll use them inside your model definition to prepare it for PTQ or QAT.

Once you define the model, these will play a major role. Ready for the next part? Go ahead and send it!

```
Load training and test data from the MNIST dataset and apply a normalizing transformation.

[ ] transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=16, pin_memory=True)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
                                         shuffle=False, num_workers=16, pin_memory=True)
```

Nice — this section is about **loading and preprocessing the MNIST dataset**, which contains handwritten digit images from 0 to 9.

Let's break it down:

1. Define a Preprocessing Pipeline

python

 Copy  Edit

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

What this does:

- `ToTensor()` – Converts PIL images (28x28 grayscale) to PyTorch tensors with values scaled between 0 and 1.
- `Normalize((0.5,), (0.5,))` – Normalizes pixel values:
 - Mean = 0.5, Std Dev = 0.5
 - Formula: $(x - 0.5) / 0.5 \rightarrow [-1, 1]$ range

 Why normalize? It helps the model learn faster by keeping the input distribution centered and scaled.

2. Load the MNIST Training & Test Sets

python

 Copy  Edit

```
trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
```

- Downloads and loads the training set.
- Applies the `transform` to every image.

python

 Copy  Edit

```
trainloader = DataLoader(trainset, batch_size=64, shuffle=True,
num_workers=16, pin_memory=True)
```

- Batches the data (64 images per batch).
- Shuffles the training data for better learning.
- `num_workers=16` speeds up loading if you have a strong CPU (lower this on weak machines).
- `pin_memory=True` improves performance when using GPU (safe to leave as-is even on CPU).

python

 Copy  Edit

```

testset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=False, num_workers=16,
pin_memory=True)

```

- Similar logic for test data — but no shuffling (since order doesn't matter for evaluation).

Let's move on to the **model definition** next — that's usually where we prepare the network for quantization (with `QuantStub`, etc.). Send it over when ready!

```

Define some helper functions and classes that help us to track the statistics and accuracy with respect to the train/test data.

[ ] class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self, name, fmt='%.3f'):
        self.name = name
        self.fmt = fmt
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

    def __str__(self):
        return self.fmt % (self.name) + ('val' + self.fmt + ')\t((avg' + self.fmt + ')')

def accuracy(output, target):
    """Computes the size 1 accuracy"""
    with torch.no_grad():
        batch_size = target.size(0)

        _, pred = output.topk(1, 1, True, True)
        pred = pred.t()
        correct = pred.eq(target.view(1, -1).expand_as(pred))

    res = []
    correct_1one = correct[:1].view(-1).float().sum(0, keepdim=True)
    with correct_1one.mul_(100.0 / batch_size).item():

def print_size_of_model(model):
    """Prints the total size of the model"""
    torch.save(model.state_dict(), "temp.p")
    print("Size (MB):", os.path.getsize("temp.p")/1e6)
    os.remove("temp.p")

def load_model(quantized_model, model):
    """Loads in the weights into an object meant for quantization"""
    state_dict = model.state_dict()
    model = model.to('cpu')
    quantized_model.load_state_dict(state_dict)

def fuse_modules(model):
    """Fuse together convolutions/linear layers and Relu"""
    torch.quantization.fuse_modules(model, [
        ['conv1', 'relu1'],
        ['conv2', 'relu2'],
        ['fc1', 'relu3'],
        ['fc2', 'relu4'],
    ], inplace=True)

```

Great — this section defines **utility functions and helpers** to support training, evaluation, and quantization. Let's go through them one by one:

AverageMeter Class

python

Copy Edit

```
class AverageMeter(object):
```

Used to **track and average values** across batches (like loss or accuracy).

Key Methods:

- `reset()` – Resets all counters to zero.
- `update(val, n)` – Adds a new value and updates average.
- `__str__()` – Nicely formats and prints current and average values.

 Commonly used during training to print smooth averages instead of noisy per-batch values.

accuracy(output, target)

python

 Copy  Edit

```
def accuracy(output, target):
```

Calculates **Top-1 Accuracy** (i.e., whether the top predicted class is correct):

- `output.topk(1, 1)` – Gets the top class prediction.
- Compares predictions to the target labels.
- Returns % of correct predictions in the batch.

 Used during validation/testing to track performance.

print_size_of_model(model)

python

 Copy  Edit

```
def print_size_of_model(model):
```

- Saves the model temporarily to a file (`temp.p`)
- Prints its size in MB
- Deletes the file afterward

 Useful for showing how **quantization reduces model size**.

⌚ load_model(quantized_model, model)

python

Copy Edit

```
def load_model(quantized_model, model):
```

- Transfers the trained weights (state_dict) from the original model to the quantized model.
- Moves quantized model to CPU.

⌚ PTQ uses this to load weights from a pretrained full-precision model into a quantized model structure.

🔥 fuse_modules(model)

python

Copy Edit

```
def fuse_modules(model):
```

This is **key for quantization**.

- Fuses layers like Conv + ReLU , Linear + ReLU , etc.
- Fused layers are more efficient and quantization-friendly.

python

Copy Edit

```
torch.quantization.fuse_modules(model, [['conv1', 'relu1'], ...])
```

🔧 Must be called before static quantization or QAT.

These functions are utility building blocks. Once the model is defined, trained, and quantized, these will help measure and optimize it.

Ready to move on to the **model definition and training setup**? Send it over when you're ready!

Define a simple CNN that classifies MNIST images.

```
[ ] class Net(nn.Module):
    def __init__(self, q = False):
        # By turning on Q we can turn on/off the quantization
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, bias=False)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5, bias=False)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256, 120, bias=False)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(120, 84, bias=False)
        self.relu4 = nn.ReLU()
        self.fc3 = nn.Linear(84, 10, bias=False)
        self.q = q
        if q:
            self.quant = QuantStub()
            self.dequant = DeQuantStub()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        if self.q:
            x = self.quant(x)
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        # Be careful to use reshape here instead of view
        x = x.reshape(x.shape[0], -1)
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x)
        x = self.relu4(x)
        x = self.fc3(x)
        if self.q:
            x = self.dequant(x)
        return x

[ ] net = Net(q=False).cuda()
print_size_of_model(net)
```

Awesome — now you're looking at the **LeNet-style CNN model** used to classify MNIST images, with built-in support for quantization. Let's break this down step by step:

🧠 class Net(nn.Module)

This defines your CNN model — it includes:

- Convolutional layers
- ReLU activations
- Pooling layers
- Fully connected (linear) layers
- Optional quantization stubs (QuantStub , DeQuantStub)

🔧 __init__(self, q=False)

This constructor sets up the model's layers.

◆ Convolutional + Pooling Layers:

python

```
self.conv1 = nn.Conv2d(1, 6, 5, bias=False)
self.pool1 = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5, bias=False)
self.pool2 = nn.MaxPool2d(2, 2)
```

- Conv2d : Applies convolution (filtering)
- MaxPool2d : Downsamples spatial dimensions
- This is classic LeNet-style feature extraction.

◆ Fully Connected (Linear) Layers:

python

 Copy  Edit

```
self.fc1 = nn.Linear(256, 120, bias=False)
self.fc2 = nn.Linear(120, 84, bias=False)
self.fc3 = nn.Linear(84, 10, bias=False)
```

- Maps the extracted features to the output digit class (0–9).

◆ Quantization Support:

python

 Copy  Edit

```
if q:
    self.quant = QuantStub()
    self.dequant = DeQuantStub()
```

- q=True enables quantization support.
- QuantStub and DeQuantStub mark where quantization and dequantization should happen.

forward() Method

This defines the flow of data through the model.

If q=True , the model:

1. Quantizes input
2. Runs it through all layers
3. Dequantizes before returning

```
python
```

 Copy

 Edit

```
if self.q:  
    x = self.quant(x)  
...  
if self.q:  
    x = self.dequant(x)
```

↻ Reshape step:

```
python
```

 Copy

 Edit

```
x = x.reshape(x.shape[0], -1)
```

- Flattens the output of conv layers into a vector before feeding into fc1 .

 Safer than .view() when quantization is involved, as .view() can break on non-contiguous tensors.

✓ Usage Example

```
python
```

 Copy

 Edit

```
net = Net(q=False).cuda()  
print_size_of_model(net)
```

- Creates a non-quantized version of the model on GPU (for training or evaluation).
- Prints model size using the earlier utility function.

 This setup makes it super easy to switch between quantized and non-quantized versions of the model by toggling `q=True` or `False`.

Ready to see the training code next? Send it over!

```
Train this CNN on the training dataset (this may take a few moments).

[ ] def train(model: nn.Module, dataloader: DataLoader, cuda=False, q=False):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    model.train()
    for epoch in range(10): # loop over the dataset multiple times
        running_loss = AverageMeter('loss')
        acc = AverageMeter('train_acc')
        for i, data in enumerate(dataloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            if cuda:
                inputs = inputs.cuda()
                labels = labels.cuda()

            # zero the parameter gradients
            optimizer.zero_grad()

            if epoch>=3 and q:
                model.apply(torch.quantization.disable_observer)

            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss.update(loss.item(), outputs.shape[0])
            acc.update(accuracy(outputs, labels), outputs.shape[0])
            if i % 100 == 0: # print every 100 mini-batches
                print('#%d, %5d' % (epoch + 1, i + 1), running_loss, acc)
    print('Finished Training')

def test(model: nn.Module, dataloader: DataLoader) -> float:
    correct = 0
    total = 0
    model.eval()
    with torch.no_grad():
        for data in dataloader:
            inputs, labels = data

            if cuda:
                inputs = inputs.cuda()
                labels = labels.cuda()

            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total

[ ] train(net, trainloader, cuda=True)
```

Perfect — now we're looking at the **training and testing logic** for your LeNet-style model on MNIST. Let's walk through it in detail:

train() Function

Parameters:

- `model` : your CNN (can be quantized or not)
- `dataloader` : training data loader
- `cuda` : use GPU if `True`
- `q` : enable quantization behavior (e.g., disable observers later on)

Setup:

```
python
```

```
criterion = nn.CrossEntropyLoss()  
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

- **CrossEntropyLoss**: For classification tasks.
 - **SGD optimizer**: Standard gradient descent with momentum.
-

⟳ Epoch loop:

```
python
```

Copy Edit

```
for epoch in range(10):
```

Trains the model for 10 epochs.

📦 Inside each epoch:

a. Get batches of data

```
python
```

Copy Edit

```
for i, data in enumerate(dataloader, 0):  
    inputs, labels = data
```

Moves data to GPU if `cuda=True`.

b. Zero the gradients

```
python
```

Copy Edit

```
optimizer.zero_grad()
```

c. Optional: Disable observers for static quantization

```
python
```

Copy Edit

```
if epoch >= 3 and q:  
    model.apply(torch.quantization.disable_observer)
```

This stops collecting stats after 3 epochs — helpful for post-training quantization.

d. Forward → Loss → Backward → Optimize

python

Copy

Edit

```
outputs = model(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

Track & Print Stats

python

Copy

Edit

```
running_loss.update(...)
acc.update(...)
```

Prints every 100 mini-batches:

python

Copy

Edit

```
print(' [%d, %5d] ...')
```

✓ test() Function

What it does:

- Puts the model in `eval()` mode
- Turns off gradient calculations
- Runs inference on the test set
- Tracks number of correct predictions
- Returns **overall accuracy**

🚀 Final Command

python

```
train(net, trainloader, cuda=True)
```

This starts training your model on GPU.

 If you're running this on Google Colab, using `cuda=True` is great. If you're on CPU (e.g., Raspberry Pi), set it to `False`.

Ready to see the next stage — likely post-training quantization or evaluation of a quantized version? Send it when you're ready!

Now that the CNN has been trained, let's test it on our test dataset.

```
[ ] score = test(net, testloader, cuda=True)  
print('Accuracy of the network on the test images: {}% - FP32'.format(score))
```

Nice — this step runs the **trained CNN model on the test dataset** and evaluates its accuracy in full precision (FP32).

🔍 Breakdown

python

 Copy  Edit

```
score = test(net, testloader, cuda=True)
```

- Calls the `test()` function you defined earlier.
- Feeds the test set through the trained `net`.
- Calculates and returns classification **accuracy**.

python

 Copy  Edit

```
print('Accuracy of the network on the test images: {}% - FP32'.format(score))
```

- Displays the result.
 - Labels it as **FP32** to indicate this is the full-precision (non-quantized) model.
-

✓ What You'll See

Something like:

```
nginx
```

Copy Edit

```
Accuracy of the network on the test images: 98.4% - FP32
```

This gives you a baseline before quantization. Later, you'll compare it with a quantized version (e.g., int8) to see:

- How much accuracy is lost (if any)
 - How much smaller and faster the model becomes
-

Ready for the quantization part? Let's see the next cell when you're ready!

▼ Post-training quantization

Define a new quantized network architecture, where we also define the quantization and dequantization stubs that will be important at the start and at the end.

Next, we'll "fuse modules"; this can both make the model faster by saving on memory access while also improving numerical accuracy. While this can be used with any model, this is especially common with quantized models.

```
❶ qnet = Net(q=True)
❷ load_model(qnet, net)
❸ fuse_modules(qnet)
```

Perfect — this marks the start of **Post-Training Quantization (PTQ)**!

Let's walk through what's happening in this cell:

🧠 Objective

You're creating a quantized version of your trained CNN model (without retraining) using PyTorch's static quantization.

Code Breakdown

1. Define a Quantized Model

python

 Copy

 Edit

```
qnet = Net(q=True)
```

- Creates a new instance of your model but with quantization enabled.
- This activates the `QuantStub` and `DeQuantStub` in your model definition.

2. Load Pretrained Weights

python

 Copy

 Edit

```
load_model(qnet, net)
```

- Loads the trained weights (`state_dict`) from your **full-precision model** (`net`) into the quantized model (`qnet`).
- This is the key step in PTQ — you're reusing the trained FP32 weights.

3. Fuse Modules

python

 Copy

 Edit

```
fuse_modules(qnet)
```

- This fuses operations like `Conv + ReLU` and `Linear + ReLU`.
- Why fuse?
 -  Improves performance
 -  Reduces memory access
 -  Ensures compatibility with quantization APIs

This is required before calling quantization functions in the next steps.

⟳ Summary of Flow So Far

sql

Copy

Edit

Train FP32 model → Define quantized model → Load weights → Fuse layers

Now you're set up to actually convert (quantize) the model and run it!

Ready to move on to prepare + convert steps? Send the next part when you're ready!

In general, we have the following process (Post Training Quantization):

1. Prepare: we insert some observers to the model to observe the statistics of a Tensor, for example, min/max values of the Tensor
2. Calibration: We run the model with some representative sample data, this will allow the observers to record the Tensor statistics
3. Convert: Based on the calibrated model, we can figure out the quantization parameters for the mapping function and convert the floating point operators to quantized operators

```
❶ qnet.qconfig = torch.quantization.default_qconfig
print(qnet.qconfig)
torch.quantization.prepare(qnet, inplace=True)
print('Post Training Quantization Prepare: Inserting Observers')
print('\n Conv1: After observer insertion \n\n', qnet.conv1)

test(qnet, trainloader, cuda=False)
print('Post Training Quantization: Calibration done')
torch.quantization.convert(qnet, inplace=True)
print('Post Training Quantization: Convert done')
print('\n Conv1: After fusion and quantization \n\n', qnet.conv1)
print("Size of model after quantization")
print_size_of_model(qnet)
```

Awesome — now you're doing the **actual quantization** using PyTorch's **Post-Training Static Quantization (PTQ)** flow.

Let's break it all down:

⟳ PTQ Workflow Summary

You're following the 3-step process:

1. Prepare

Insert observers to track stats like min/max ranges (used to compute quantization scales and zero points)

python

Copy

Edit

```
qnet.qconfig = torch.quantization.default_qconfig
```

```
torch.quantization.prepare(qnet, inplace=True)
```

- Sets the quantization configuration (default is for static quantization using QNNPACK).
- `prepare()` inserts observers (like sensors) into the model.

2. Calibrate

Run the model on representative data so the observers can collect stats

```
python
```

 Copy  Edit

```
test(qnet, trainloader, cuda=False)
```

- No training here! You're just running inference on sample data (training data works fine).
- This gives the model enough examples to record activation ranges.

3. Convert

Turn observed stats into real quantization (replace float ops with int8 ops)

```
python
```

 Copy  Edit

```
torch.quantization.convert(qnet, inplace=True)
```

- Converts layers to quantized versions.
- You now have a true quantized model.

Debug Prints

```
python
```

 Copy  Edit

```
print(qnet.qconfig)
print('\n Conv1: After observer insertion \n\n', qnet.conv1)
```

```
print('\n Conv1: After fusion and quantization \n\n', qnet.conv1)
```

- Lets you inspect how conv1 changes before and after quantization.

📦 Check Model Size

python

Copy Edit

```
print_size_of_model(qnet)
```

You'll now see that the model is **significantly smaller** — this confirms successful quantization.

Let's recap:

- ✓ You trained an FP32 model
- ✓ Cloned and prepared it for quantization
- ✓ Ran calibration data
- ✓ Converted to int8 operations
- ✓ Verified size reduction

🎉 Next, I assume you'll test the quantized model's accuracy? Send it over when ready!

```
[ ] score = test(qnet, testloader, cuda=False)
      print('Accuracy of the fused and quantized network on the test images: {}% - INT8'.format(score))
```

Awesome — this is the **final step** of Post-Training Quantization (PTQ): evaluating the quantized model!

✍️ Code Breakdown

python

Copy Edit

```
score = test(qnet, testloader, cuda=False)
```

- Runs the **fully quantized model** (qnet) on the test set.
- Note: cuda=False because quantized models in PyTorch only run on **CPU** (not supported on GPU).

python

Copy Edit

```
print('Accuracy of the fused and quantized network on the test images: {}% - INT8'.format(score))
```

- This shows the **test accuracy of your quantized model** (in INT8 precision).
- You can now compare this to the FP32 version printed earlier.

✓ What to Expect

Model	Size	Accuracy (Expected)
FP32 (float32)	Larger	High (e.g., 98%+)
INT8 (quantized)	Smaller	Slightly lower (e.g., 97–98%)

Usually, **size reduction is huge**, and **accuracy loss is small** — a great tradeoff for deployment on edge devices like Raspberry Pi.

If your next section is about **Quantization-Aware Training (QAT)**, feel free to send it — I'll walk you through that too!

```
We can also define a custom quantization configuration, where we replace the default observers and instead of quantising with respect to max/min we can take an average of the observed max/min, hopefully for a better generalization performance.

❶ from torch.quantization.observer import MovingAverageMinMaxObserver
❷ qnet = Net(q=True)
❸ load_model(qnet, net)
❹ fuse_modules(qnet)

❺ qnet.qconfig = torch.quantization.QConfig(
    activation=MovingAverageMinMaxObserver.with_args(reduce_range=True),
    weight=MovingAverageMinMaxObserver.with_args(dtype=torch.qint8, qscheme=torch.per_tensor_symmetric))
❻ print(qnet.qconfig)
❼ torch.quantization.prepare(qnet, inplace=True)
❼ print("Post Training Quantization Prepare: Inserting Observers")
❼ print("\nConv1: After observer insertion \n\n", qnet.conv1)

❼ test(qnet, trainloader, cuda=False)
❼ print("Post Training Quantization Calibration done")
❼ torch.quantization.convert(qnet, inplace=True)
❼ print("Post Training Quantization Convert done")
❼ print("\nConv1: After fusion and quantization \n\n", qnet.conv1)
❼ print("Size of model after quantization")
❼ print_size_of_model(qnet)
❼ score = test(qnet, testloader, cuda=False)
❼ print('Accuracy of the fused and quantized network on the test images: {}% - INT8'.format(score))
```

Awesome — this section demonstrates how to create a **custom quantization configuration** using a more advanced observer: `MovingAverageMinMaxObserver`.

Let's break down what this means and what this code is doing.

🔍 What's New Here?

📐 Custom Observer

Instead of the default `MinMaxObserver`, you're using:

```
python Copy Edit  
MovingAverageMinMaxObserver
```

This computes **quantization scales** using a *moving average* of min/max values across batches — rather than relying on just one batch.

- This often improves generalization and reduces overfitting to the calibration data.
-

🔧 Custom Quantization Config

```
python Copy Edit  
  
qnet.qconfig = torch.quantization.QConfig(  
    activation=MovingAverageMinMaxObserver.with_args(reduce_range=True),  
    weight=MovingAverageMinMaxObserver.with_args(  
        dtype=torch qint8, qscheme=torch per_tensor_symmetric  
    )  
)
```

Breakdown:

- `activation=...` : quantization settings for layer outputs.
- `weight=...` : quantization settings for model weights.

Key Arguments:

- `reduce_range=True` : Uses 7-bit int instead of full 8-bit (helps for some hardware).

- `dtype=torch.qint8` : Specifies 8-bit signed integers.
 - `qscheme=torch.per_tensor_symmetric` : Applies the same quantization scale across the tensor (simpler and efficient for weights).
-

Full Quantization Flow (Same as Before)

1. **Prepare** (inserts observers):

```
python Copy Edit
torch.quantization.prepare(qnet, inplace=True)
```

2. **Calibrate** (run sample data to populate observers):

```
python Copy Edit
test(qnet, trainloader, cuda=False)
```

3. **Convert** (apply quantization):

```
python Copy Edit
torch.quantization.convert(qnet, inplace=True)
```

4. **Test Accuracy + Print Size**

- Evaluates model performance
 - Confirms compression
-

Why Use This?

This approach gives you more control and sometimes **better accuracy** on INT8 models, especially when the default observer is too sensitive to outliers in min/max values.

Summary

You're still doing post-training quantization, but now with:

- Custom observers
- Smoother calibration
- Likely better accuracy on the test set

Let me know if your next section covers **Quantization-Aware Training (QAT)** — that's the last and most advanced optimization method!

```
In addition, we can significantly improve on the accuracy simply by using a different quantization configuration. We repeat the same exercise with the recommended configuration for quantizing for arm64 architecture (qnnpack). This configuration does the following: Quantizes weights on a per-channel basis. It uses a histogram observer that collects a histogram of activations and then picks quantization parameters in an optimal manner.

[ ] qnet = Net(q=True)
    load_model(qnet, net)
    fuse_modules(qnet)

[ ] qnet.qconfig = torch.quantization.get_default_qconfig('qnnpack')
    print(qnet.qconfig)

    torch.quantization.prepare(qnet, inplace=True)
    test(qnet, trainloader, cuda=False)
    torch.quantization.convert(qnet, inplace=True)
    print("Size of model after quantization")
    print_size_of_model(qnet)

    ↵ QConfig(activation=functools.partial(<class 'torch.ao.quantization.observer.HistogramObserver'>, reduce_range=False))
    Size of model after quantization
    Size (MB): 0.050084

[ ] score = test(qnet, testloader, cuda=False)
    print('Accuracy of the fused and quantized network on the test images: {}% - INT8'.format(score))

    ↵ Accuracy of the fused and quantized network on the test images: 98.02% - INT8
```

Fantastic — this is a really practical wrap-up to your post-training quantization work, showing how to use a **hardware-optimized quantization configuration** for better performance **on ARM devices like the Raspberry Pi**.

💡 What's New Here?

📦 Quantization with qnnpack

python

Copy Edit

```
qnet.qconfig = torch.quantization.get_default_qconfig('qnnpack')
```

- `qnnpack` is a **quantization backend optimized for mobile ARM CPUs** (like Raspberry Pi).
- It uses:
 - **Histogram-based observers** for activations — better at handling a wide range of input values.
 - **Per-channel weight quantization** — improves accuracy by allowing separate scales per channel (rather than one global scale).

 This is a go-to config for deploying quantized models on edge devices.

Workflow (same PTQ steps)

1. Prepare:

```
python  
torch.quantization.prepare(qnet, inplace=True)
```

 Copy  Edit

2. Calibrate:

```
python  
test(qnet, trainloader, cuda=False)
```

 Copy  Edit

3. Convert:

```
python  
torch.quantization.convert(qnet, inplace=True)
```

 Copy  Edit

4. Print size and accuracy:

```
python  
print_size_of_model(qnet)  
score = test(qnet, testloader, cuda=False)
```

 Copy  Edit

Results

```
python  
Accuracy of the fused and quantized network on the test images: 98.02% – INT8
```

 Copy  Edit

- This shows **almost no drop in accuracy** from the original FP32 model!
- Model size is just **~0.05 MB**, thanks to quantization 

🔥 Key Takeaway

By using `qnnpack` with histogram observers and per-channel quantization, you get:

- High accuracy
- Small model size
- Fast inference on edge devices (like Raspberry Pi)

You're in great shape to deploy this to a real device or use it in a video inference pipeline.

Would you like to continue to **Quantization-Aware Training (QAT)** next? That's the most accurate (but also most advanced) form of quantization.

▼ Quantization aware training

Quantization-aware training (QAT) is the quantization method that typically results in the highest accuracy. With QAT, all weights and activations are "fake quantized" during both the forward and backward passes of training: that is, float values are rounded to mimic int8 values, but all computations are still done with floating point numbers.

```
[ ] qnet = Net(q=True)
    fuse_modules(qnet)
    qnet.qconfig = torch.quantization.get_default_qat_qconfig('qnnpack')
    torch.quantization.prepare_qat(qnet, inplace=True)
    print("\n Conv1: After fusion and quantization \n\n", qnet.conv1)
    qnet=qnet.cuda()
    train(qnet, trainloader, cuda=True)
    qnet = qnet.cpu()
    torch.quantization.convert(qnet, inplace=True)
    print("Size of model after quantization")
    print_size_of_model(qnet)

    score = test(qnet, testloader, cuda=False)
    print('Accuracy of the fused and quantized network (trained quantized) on the test images: {}% - INT8'.format(score))
```

Amazing — you've reached the final and most accurate quantization technique: **Quantization-Aware Training (QAT)!**

Let's go step by step through this important section.

🧠 What is QAT?

Quantization-Aware Training simulates quantization during training.

- During **both forward and backward passes**, weights and activations are "**fake quantized**" (i.e., rounded to int8-like values).
- But training is still done using float32 under the hood.

- This gives the model a chance to **learn how to perform well under quantization constraints**.

✓ QAT typically gives the best trade-off between **accuracy** and **compression**.

📦 Step-by-Step Breakdown

1. Define and Prepare the Model

```
python Copy Edit
qnet = Net(q=True)
fuse_modules(qnet)
```

- Activates quantization stubs.
 - Fuses Conv/ReLU and Linear/ReLU layers (required before QAT).
-

2. Set QAT Configuration

```
python Copy Edit
qnet.qconfig = torch.quantization.get_default_qat_qconfig('qnnpack')
```

- Uses the QAT-friendly config with observers and fake quant ops.
 - `qnnpack` is ideal for ARM devices (e.g., Raspberry Pi).
-

3. Prepare the Model for QAT

```
python Copy Edit
torch.quantization.prepare_qat(qnet, inplace=True)
```

- Inserts **fake quantization modules** into the model to simulate quantization during training.
-

4. Train with QAT

```
python
```

 Copy  Edit

```
qnet = qnet.cuda()  
train(qnet, trainloader, cuda=True)
```

- Trains the model **with fake quantization enabled**.
 - Allows it to adapt to low-precision arithmetic.
-

5. Convert to INT8 (real quantization)

```
python
```

 Copy  Edit

```
qnet = qnet.cpu()  
torch.quantization.convert(qnet, inplace=True)
```

- Converts fake quantization modules to **actual quantized operations**.
-

6. Size and Accuracy Check

```
python
```

 Copy  Edit

```
print_size_of_model(qnet)  
score = test(qnet, testloader, cuda=False)
```

You'll get:

-  A small model (INT8 format)
 -  The highest possible accuracy for a quantized model
-

✓ What Makes QAT So Powerful?

Feature	PTQ	QAT
Requires retraining	✗ No	✓ Yes
Accuracy	⚠ May drop	✓ Best
Implementation effort	🟢 Easy	🔴 Harder
Simulation of int8	✗ No	✓ Yes

Let me know if you want help:

- Exporting this model for deployment (TorchScript, TFLite, etc.)
- Benchmarking latency/inference time
- Integrating with a webcam pipeline on your Pi

You're fully equipped to build real-time, optimized edge ML systems now 🙌

that is the end of the jupyter notebook. now i will go back to the lab sheet.

Perfect — you're doing great! You've now explored all three major quantization methods in PyTorch:

- ✓ Post-Training Quantization (PTQ)
- ✓ PTQ with custom observers
- ✓ Quantization-Aware Training (QAT)

You're more than ready to understand and evaluate whatever comes next in the lab sheet. Go ahead and send the next section when you're ready, and I'll walk you through it!

4. Homework and Optional Exercise

- Try running quantized version of some large language models (like llama, mixtral etc.) on Raspberry Pi. This [link](#) demonstrates some of the LLMs on Raspberry Pi 4 and 5.

link: <https://www.dfrobot.com/blog-13498.html>

Awesome — this wraps up your lab with a **homework/optional challenge**: running **quantized large language models (LLMs)** on the Raspberry Pi.

Task Summary

Try running quantized LLMs (e.g., llama , mixtral) on Raspberry Pi:

- These are **transformer-based models** for text generation, question answering, etc.
 - They are **massive**, so running them in full precision on a Pi is impossible — but **quantization and model distillation** can make it work.
-

Reference Link

[DFRobot Blog – Running LLMs on Raspberry Pi 4 & 5](#)

This link demonstrates:

- What LLMs can actually run on Raspberry Pi
 - How quantization and architecture choices make them feasible
 - Use of frameworks like `llama.cpp` , `ggml` , or `mlc-llm`
-

Getting Started Tips

If you want to try it yourself:

Recommended:

- Use `llama.cpp` or `mlc-llm` — both support quantized models (Q4, Q8, etc.)
- Run a small model variant (e.g., `llama-2 7B Q4`)
- Consider Raspberry Pi 5 or Pi 4 with at least 4GB RAM and swap space

Tools:

- `llama.cpp` : CLI tool for inference (runs on CPU)
- `mlc-llm` : More interactive, supports chat-like apps on edge devices

- ggml : Back-end tensor library optimized for quantized weights
-

Example Projects to Explore:

- [llama.cpp GitHub](#)
 - [MLC-LLM GitHub](#)
-

Would you like help trying this out on your Raspberry Pi (e.g., installing `llama.cpp`, downloading a quantized model, running a prompt)? I can guide you through it step-by-step.

yes please guide me through it step-by-step

Awesome — let's get you started with **running a quantized LLM on your Raspberry Pi** using `llama.cpp`. This is one of the most lightweight and Pi-friendly tools out there.

Step-by-Step: Run LLaMA on Raspberry Pi (Using `llama.cpp`)

Prerequisites

- Raspberry Pi 4 or 5 (4GB or more RAM recommended)
 - Raspbian OS (64-bit)
 - Python 3, Git, CMake
 - Some terminal experience (you got this 💪)
-

Step 1: Install Required Packages

Open a terminal on your Pi and run:

```
bash
```

Copy

Edit

```
sudo apt update && sudo apt install -y build-essential cmake git python3  
python3-pip
```

Step 2: Clone llama.cpp Repository

```
bash
```

Copy

Edit

```
git clone https://github.com/ggerganov/llama.cpp  
cd llama.cpp
```

Step 3: Build the Project

This compiles the optimized inference engine for ARM:

```
bash
```

Copy

Edit

```
make
```

 Might take a few minutes. If you're on Pi 4, consider `make -j2` to limit CPU load.

Step 4: Download a Quantized LLaMA Model

You need a **quantized .gguf model**. Try the **LLaMA 2 7B Q4** or a smaller **TinyLLaMA** model for a faster start.

Use Hugging Face to download a model:

1. Visit <https://huggingface.co/TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF>
2. Download a small `.gguf` file (e.g., `q4_0.gguf`)
3. Place it in your `llama.cpp` directory.

Example (if using `wget`):

```
bash
```

Copy

Edit

```
 wget https://huggingface.co/TheBloke/TinyLlama-1.1B-Chat-v1.0-  
GGUF/resolve/main/tinyllama-1.1b-chat-v1.0.Q4_0.gguf
```

Step 5: Run the Model with a Prompt

bash

 Copy  Edit

```
./main -m tinyllama-1.1b-chat-v1.0.Q4_0.gguf -p "What is Raspberry Pi?" -n 100
```

- `-m` : model file
- `-p` : your prompt
- `-n` : max tokens to generate

You're Done!

You should see the model generate a text response — it may be a bit slow (especially on Pi 4), but **quantized models make it possible** 

Optional Extras

- Add a simple web UI (e.g., with Gradio)
- Use chat template (`--chat`) for more conversational flow
- Try other models like Mixtral or Phi-2 (only if your Pi has enough memory)

Would you like me to help you with downloading a specific model or setting up chat-like interaction next?

<https://github.com/ggml-org/llama.cpp/blob/master/docs/build.md>. Stop.

Ah yep — that's a recent change! The `llama.cpp` project has now **deprecated the `make build`**, and it uses **CMake** for building instead.

No worries — I'll guide you through the updated steps:

✓ Updated Build Instructions (with CMake)

Step 1: Clone the repo (if you haven't already)

```
bash Copy Edit
git clone https://github.com/ggerganov/llama.cpp
cd llama.cpp
```

Step 2: Create a build directory

```
bash Copy Edit
mkdir build
cd build
```

Step 3: Run CMake configuration

```
bash Copy Edit
cmake ..
```

If you're on Raspberry Pi 4/5 and want NEON optimization:

```
bash Copy Edit
cmake .. -DLLAMA_NATIVE=ON
```

Step 4: Compile it

bash

 Copy  Edit

```
cmake --build . --config Release
```

You'll now get the compiled binaries like `main`, `quantize`, etc., in the `build` folder.

Now you can run it like this:

bash

 Copy  Edit

```
./main -m ../tinyllama-1.1b-chat-v1.0.Q4_0.gguf -p "What is Raspberry Pi?" -n 100
```

(make sure your model `.gguf` file is in the parent directory or adjust the path accordingly)

Let me know if you get stuck on any of those steps, or want help choosing a lightweight model to test first!

↳ You'll now get the compiled binaries like `'main'`, `'quantize'`, etc., in the `'build'` folder.

no compiled binaries appeared

Got it — if you ran the CMake build successfully but no `main` or other binaries appeared, something likely went wrong during the `cmake --build` step.

Let's troubleshoot this step-by-step:

Step-by-Step Check

1. Are You in the Correct Directory?

You should be inside the `build` folder you created:

```
bash
```

 Copy

 Edit

```
cd ~/llama.cpp/build
```

2. Check if the Build Failed

Run this inside the `build` directory:

```
bash
```

 Copy

 Edit

```
cmake --build . --config Release
```

Watch carefully for any errors. If you see **compiler errors**, let me know what they are — I'll help fix them.

3. Check What Was Actually Built

Run:

```
bash
```

 Copy

 Edit

```
ls
```

You should see binaries like:

- `main`
- `quantize`
- `server` (optional)

If not, try building `main` directly:

```
bash
```

 Copy

 Edit

```
cmake --build . --target main
```

4. ✕ Clean and Rebuild (if needed)

If something feels broken or partially built:

bash

Copy

Edit

```
cd ..  
rm -rf build  
mkdir build  
cd build  
cmake .. -DLLAMA_NATIVE=ON  
cmake --build . --config Release
```

5. ? Still nothing?

Let's check your system:

- What model of Raspberry Pi are you using?
- What OS version are you on?
- What version of `cmake` do you get from `cmake --version` ?

Once we get the binaries compiled, you'll be able to run the model easily. Just let me know what happens during the build step and we'll fix it together!

The screenshot shows the official CMake documentation for building on a CPU. It includes examples of CMake commands for building, notes on configuration generators, and specific instructions for the Raspberry Pi regarding the ignore of the --config Release flag.

this is from the official documentation

Thanks! That clears things up — and it reveals the **important detail**: the Raspberry Pi's default CMake generator is a **single-config generator (Makefiles)**, which means the `--config Release` flag is **ignored**.

That's why you didn't get your `main` binary — it didn't build it in Release mode the way you'd expect.

✓ Fix: Use This 2-Step Command Instead

bash

Copy Edit

```
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j4
```

🔍 Explanation:

- `-DCMAKE_BUILD_TYPE=Release` → explicitly sets release mode
- `-j4` → builds with 4 threads (you can reduce or increase depending on your Pi model)

Then check if the `main` binary is present:

bash

Copy Edit

```
ls build
```

You should see:

```
main quantize ...
```

Run it with:

```
bash
```

Copy

Edit

```
./build/main -m path/to/model.gguf -p "Your prompt here" -n 100
```

Let me know how this goes! If you still don't see the binary, I can help check CMake logs or try a clean rebuild.

- Take any complex Deep Learning Model like resnet, mobileNet OR your own architecture and try different quantization methods as explained in section 3. Once deployed on RaspberryPi, Observe the size, performance and speed of the quantized model.

okay thank you. now i will move on to the final section of my lab sheet. it is my second homework and optional exercise

Awesome — this second optional exercise gives you the freedom to apply everything you've learned!

Task Summary

Take any complex deep learning model (like ResNet, MobileNet, or one you design), then:

1. Try **different quantization methods**:

- Post-training quantization (PTQ)
- PTQ with custom observers
- Quantization-Aware Training (QAT)

2. Deploy the quantized model to Raspberry Pi

3. Measure:

- **Model size** (e.g., using `os.path.getsize()` or `.pt` file size)
- **Speed / FPS** (e.g., using OpenCV or `time.time()`)

- **Performance** (accuracy on test data)
-

✓ Suggested Project: ResNet18 on Raspberry Pi

Why ResNet18?

- It's deep and complex enough to be interesting.
 - Pretrained version available in `torchvision.models`.
-

🏗 Steps Outline

1. Load pretrained ResNet18

```
python
```

```
from torchvision.models import resnet18
model = resnet18(pretrained=True)
```

2. Prepare for quantization

- Add `QuantStub / DeQuantStub`
- Use `fuse_modules(model, [...])`

3. Try each quantization method

- PTQ → convert without retraining
- PTQ + histogram observer
- QAT → fine-tune for a few epochs

4. Export model size

```
python
```

```
torch.save(model.state_dict(), "model.pth")
```

5. Deploy to Raspberry Pi

- Test inference speed (on camera stream or image batch)
- Compare `fp32 vs int8` size, accuracy, and speed

If you'd like, I can help you:

- Set up a Jupyter notebook to experiment
- Choose the right dataset and input size
- Optimize for Raspberry Pi (e.g., use `torchvision.transforms` for pre-processing)

Just tell me which model you'd like to try — or if you want to keep using MobileNet!

yes help me to experiment. BUT i don't want a jupyter notebook. i want to experiment on my raspberry pi itself

Perfect! Let's run everything **directly on your Raspberry Pi**, step-by-step. We'll use **ResNet18** from PyTorch, quantize it, and evaluate size and performance on the Pi.

🧠 Plan: Quantize ResNet18 on Raspberry Pi

What we'll do:

1. Install dependencies
 2. Load and evaluate full-precision ResNet18
 3. Apply post-training quantization (PTQ)
 4. Measure model size and FPS
 5. Optionally, try QAT (a bit more complex but I'll guide you)
-

📦 Step 1: Install Dependencies

Open terminal on your Pi and run:

```
bash Copy Edit
      sudo apt update
      sudo apt install python3-pip libopenblas-dev
      pip3 install torch torchvision numpy opencv-python
```

Step 2: Create a Python Script (resnet_quant_test.py)

Here's your starter code:

```
python Copy Edit

import torch
from torchvision.models import resnet18
from torchvision import transforms
from PIL import Image
import time
import os

# Load pre-trained ResNet18 model
model = resnet18(pretrained=True)
model.eval()

# Save size of FP32 model
torch.save(model.state_dict(), "resnet18_fp32.pth")
print("FP32 size (MB):", os.path.getsize("resnet18_fp32.pth") / 1e6)

# Preprocessing transform
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225]),
])
])

# Dummy image
img = Image.new("RGB", (224, 224), color="white")
input_tensor = transform(img).unsqueeze(0)

# Warm-up + timing
with torch.no_grad():
    for _ in range(5): model(input_tensor) # Warm-up
    start = time.time()
    for _ in range(10): model(input_tensor)
    end = time.time()

print("FP32 average inference time:", (end - start) / 10)
```

Save this as `resnet_quant_test.py`, and run it:

```
bash
```

```
python3 resnet_quant_test.py
```

It should print:

- Model size in MB
- Average inference time

🚀 Step 3: Quantize the Model (Post-Training)

Now let's quantize ResNet18 in a separate script:

```
python Copy Edit  
  
import torch  
from torchvision.models import resnet18  
import os  
  
model_fp32 = resnet18(pretrained=True)  
model_fp32.eval()  
  
# Fuse conv+relu, etc. (required before quantization)  
fused_model = torch.quantization.fuse_modules(  
    model_fp32,  
    [['conv1', 'bn1', 'relu']],  
    inplace=False  
)
```

Ask anything



⊕ Search



Deep research

...

ChatGPT can make mistakes. Check important info.

?

□ ▲ ♫ 🔍 ⏪ 🔍 ▾