Singapore Institute of Technology

BEng (Hons) Information and Communications Technology majoring in Software Engineering

INF2009 Edge Computing and Analytics

Academic Year 2024/2025 Trimester 2

Week 5 Lab – Deep Learning on Edge

Name: Lim Chee Hean

Student ID: 2201529

2. Running Deep Learning Model on Raspberry Pi

```
(dlonedge) cheehean@raspberrypi:~/labs/dlonedge $ python mobile_net.py
/home/cheehean/labs/dlonedge/dlonedge/lib/python3.11/site-packages/torchvision/models/_utils.py:208: UserWarning: The par
ameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/cheehean/labs/dlonedge/dlonedge/lib/python3.11/site-packages/torchvision/models/_utils.py:223: UserWarning: Argumen
ts other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The curre
nt behavior is equivalent to passing `weights=MobileNet_V2_Weights.IMAGENET1K_V1`. You can also use `weights=MobileNet_V2
_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /home/cheehean/.cache/torch/hub/checkpoin
ts/mobilenet_v2-b0353104.pth
100.0%
=============0.5404524601883303 fps =================
=============15.782611599604522 fps =================
=============15.849261547944769 fps =================
=============15.898833284016344 fps =================
=============15.862155980271124 fps =================
=============15.933955237435622 fps =================
=============15.880608740942918 fps =================
=============15.872747183609945 fps =================
=============15.864240837956247 fps =================
=============15.84014097987718 fps =================
=============15.81071862829151 fps =================
=============15.770505764521754 fps =================
=============15.849924114635318 fps =================
=============15.863172092447149 fps =================
=============15.874695883603463 fps =================
=============15.831635923300075 fps =================
=============15.882747319223302 fps =================
```

Without Quantisation

```
(dlonedge) cheehean@raspberrypi:~/labs/dlonedge $ python mobile_net.py
/home/cheehean/labs/dlonedge/dlonedge/lib/python3.11/site-packages/torchvision/models/_utils.py:208: UserWarning: The par
ameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/cheehean/labs/dlonedge/dlonedge/lib/python3.11/site-packages/torchvision/models/_utils.py:223: UserWarning: Argumen
ts other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The curre
nt behavior is equivalent to passing `weights=MobileNet_V2_QuantizedWeights.IMAGENET1K_QNNPACK_V1`. You can also use `wei
ghts=MobileNet_V2_QuantizedWeights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
/home/cheehean/labs/dlonedge/dlonedge/lib/python3.11/site-packages/torch/ao/quantization/utils.py:408: UserWarning: must
run observer before calling calculate_qparams. Returning default values.
  warnings.warn(
Downloading: "https://download.pytorch.org/models/quantized/mobilenet_v2_qnnpack_37f702c5.pth" to /home/cheehean/.cache/t
orch/hub/checkpoints/mobilenet_v2_qnnpack_37f702c5.pth
100.0%
/home/cheehean/labs/dlonedge/dlonedge/lib/python3.11/site-packages/torch/_utils.py:410: UserWarning: TypedStorage is depr
ecated. It will be removed in the future and UntypedStorage will be the only storage class. This should only matter to yo
u if you are using storages directly.  To access UntypedStorage directly, use tensor.untyped_storage() instead of tensor.
storage()
  device=storage.device,
=============0.5541189961581754 fps ================
=============28.824823202270064 fps ================
=============29.75790991730729 fps ================
=============29.760894152840986 fps ================
=============29.760288811396546 fps ================
=============29.87880513388092 fps ================
=============29.68271379700307 fps ================
=============29.839982773650895 fps ================
=============29.758634806281357 fps ================
```

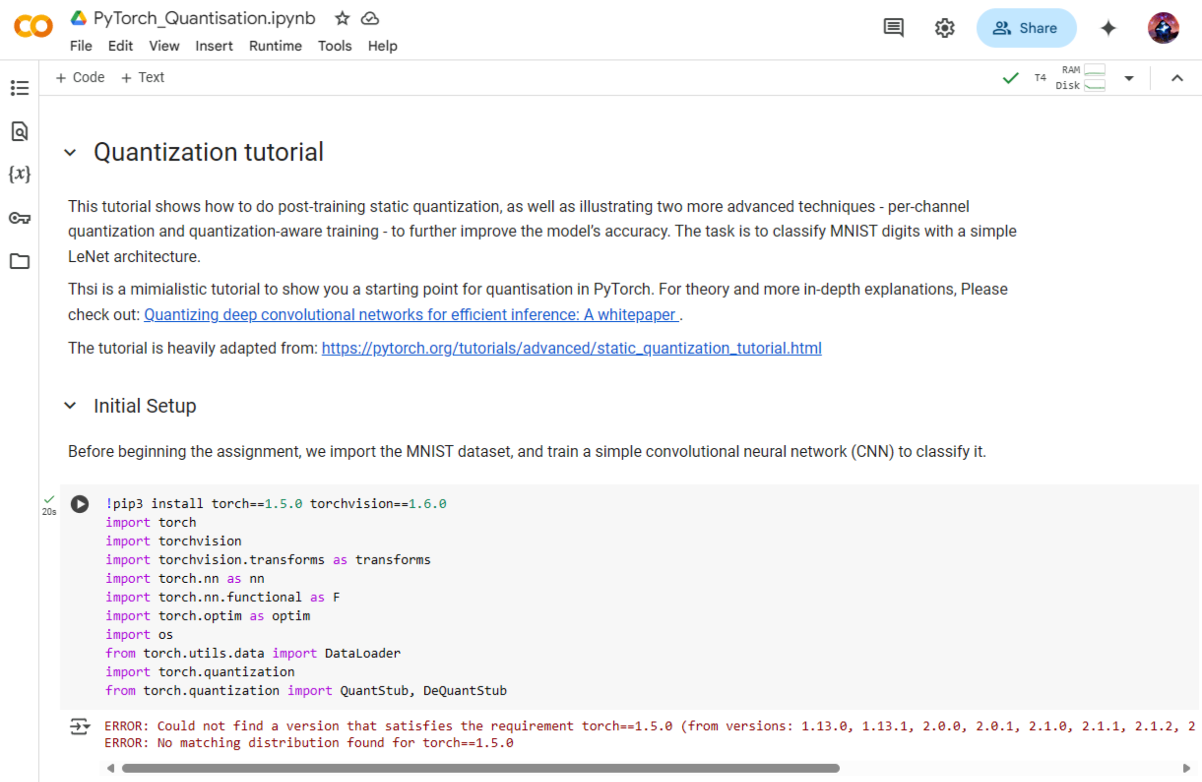
With Quantisation

```
===========================
19.83% laptop
17.02% monitor
14.60% desktop computer
12.53% notebook
6.79% space bar
4.29% computer keyboard
4.29% screen
4.29% typewriter keyboard
2.33% modem
2.00% printer
===========================
17.89% desktop computer
17.89% monitor
9.70% laptop
8.32% notebook
8.32% space bar
7.14% typewriter keyboard
5.26% photocopier
4.51% screen
3.87% desk
3.32% computer keyboard
```

Top 10 Predictions

3. Quantisation using PyTorch

Notebook run on Google Colab: https://colab.research.google.com/drive/1EYOulUvDOVy2BfuOOd0NwE_GlwsH4dwj?usp=sharing


PyTorch_Quantisation.ipynb

File Edit View Insert Runtime Tools Help

Share

+ Code + Text

T4 RAM Disk

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.

Thsi is a mimialistic 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

Initial Setup

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

20s

```
!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
```

```
ERROR: Could not find a version that satisfies the requirement torch==1.5.0 (from versions: 1.13.0, 1.13.1, 2.0.0, 2.0.1, 2.1.0, 2.1.1, 2.1.2, 2
ERROR: No matching distribution found for torch==1.5.0
```

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

[2]
```
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)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9.91M/9.91M [00:00<00:00, 16.1MB/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28.9k/28.9k [00:00<00:00, 497kB/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1.65M/1.65M [00:00<00:00, 3.82MB/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4.54k/4.54k [00:00<00:00, 2.15MB/s]
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:617: UserWarning: This DataLoader will create 16 worker processes in tota
  warnings.warn(
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

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

```
[3] class AverageMeter(object):
        """Computes and stores the average and current value"""
        def __init__(self, name, fmt=':f'):
            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):
            fmtstr = '{name} {val' + self.fmt + '} ({avg' + self.fmt + '})'
            return fmtstr.format(**self.__dict__)

    def accuracy(output, target):
        """ Computes the top 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_one = correct[:1].view(-1).float().sum(0, keepdim=True)
            return correct_one.mul_(100.0 / batch_size).item()

    def print_size_of_model(model):
        """ Prints the real 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)
```

Define a simple CNN that classifies MNIST images.

[4]
```
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
```

[5]
```
net = Net(q=False).cuda()
print_size_of_model(net)
```

```
Size (MB): 0.179057
```

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

```
[6] 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, cuda=False) -> 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
```

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

```
[5,   301] loss 0.036345 (0.108369) train_acc 98.437500 (96.807517)
[5,   401] loss 0.065963 (0.108113) train_acc 96.875000 (96.808759)
[5,   501] loss 0.051236 (0.108233) train_acc 98.437500 (96.768962)
[5,   601] loss 0.129429 (0.109293) train_acc 93.750000 (96.695611)
[5,   701] loss 0.088946 (0.108332) train_acc 96.875000 (96.718973)
[5,   801] loss 0.168593 (0.107481) train_acc 96.875000 (96.767712)
[5,   901] loss 0.069065 (0.105559) train_acc 95.312500 (96.796962)
[6,     1] loss 0.067282 (0.067282) train_acc 98.437500 (98.437500)
[6,   101] loss 0.112153 (0.097384) train_acc 96.875000 (97.246287)
[6,   201] loss 0.040557 (0.093763) train_acc 98.437500 (97.147077)
[6,   301] loss 0.135155 (0.091971) train_acc 96.875000 (97.238372)
[6,   401] loss 0.018335 (0.090673) train_acc 100.000000 (97.249065)
[6,   501] loss 0.354826 (0.092473) train_acc 92.187500 (97.199351)
[6,   601] loss 0.044738 (0.092209) train_acc 98.437500 (97.197379)
[6,   701] loss 0.096501 (0.091547) train_acc 96.875000 (97.193741)
[6,   801] loss 0.045191 (0.091416) train_acc 98.437500 (97.200765)
[6,   901] loss 0.139336 (0.090707) train_acc 96.875000 (97.225305)
[7,     1] loss 0.115988 (0.115988) train_acc 96.875000 (96.875000)
[7,   101] loss 0.098593 (0.074701) train_acc 95.312500 (97.602104)
[7,   201] loss 0.127600 (0.076514) train_acc 96.875000 (97.605721)
[7,   301] loss 0.065016 (0.076335) train_acc 98.437500 (97.669228)
[7,   401] loss 0.107845 (0.079162) train_acc 95.312500 (97.642612)
[7,   501] loss 0.066593 (0.080711) train_acc 98.437500 (97.554890)
[7,   601] loss 0.077146 (0.080138) train_acc 96.875000 (97.553557)
[7,   701] loss 0.128316 (0.080207) train_acc 95.312500 (97.541459)
[7,   801] loss 0.127779 (0.079945) train_acc 95.312500 (97.530431)
[7,   901] loss 0.008587 (0.079945) train_acc 100.000000 (97.518382)
[8,     1] loss 0.146177 (0.146177) train_acc 96.875000 (96.875000)
[8,   101] loss 0.025474 (0.070408) train_acc 100.000000 (97.988861)
[8,   201] loss 0.027232 (0.066126) train_acc 100.000000 (97.986629)
[8,   301] loss 0.156648 (0.065108) train_acc 93.750000 (97.991071)
[8,   401] loss 0.160867 (0.066723) train_acc 95.312500 (97.907575)
[8,   501] loss 0.073245 (0.067451) train_acc 96.875000 (97.929142)
[8,   601] loss 0.125441 (0.070091) train_acc 96.875000 (97.857737)
[8,   701] loss 0.019409 (0.071096) train_acc 98.437500 (97.815621)
[8,   801] loss 0.050740 (0.071536) train_acc 98.437500 (97.826935)
[8,   901] loss 0.094728 (0.071899) train_acc 98.437500 (97.825333)
[9,     1] loss 0.135984 (0.135984) train_acc 93.750000 (93.750000)
[9,   101] loss 0.079476 (0.075561) train_acc 98.437500 (97.694926)
[9,   201] loss 0.028070 (0.073862) train_acc 98.437500 (97.675684)
[9,   301] loss 0.038277 (0.070601) train_acc 98.437500 (97.767857)
[9,   401] loss 0.016259 (0.068794) train_acc 100.000000 (97.864713)
[9,   501] loss 0.025112 (0.067307) train_acc 100.000000 (97.947854)
[9,   601] loss 0.117574 (0.066609) train_acc 98.437500 (97.956531)
[9,   701] loss 0.004474 (0.066432) train_acc 100.000000 (97.958274)
[9,   801] loss 0.048276 (0.066176) train_acc 98.437500 (97.973237)
[9,   901] loss 0.018555 (0.065586) train_acc 100.000000 (98.002220)
[10,     1] loss 0.030428 (0.030428) train_acc 100.000000 (100.000000)
[10,   101] loss 0.013638 (0.055391) train_acc 100.000000 (98.329208)
[10,   201] loss 0.015486 (0.060047) train_acc 100.000000 (98.227612)
[10,   301] loss 0.039359 (0.058705) train_acc 98.437500 (98.266196)
[10,   401] loss 0.105518 (0.060443) train_acc 96.875000 (98.203709)
[10,   501] loss 0.009181 (0.060370) train_acc 100.000000 (98.169286)
[10,   601] loss 0.025167 (0.060403) train_acc 100.000000 (98.156718)
[10,   701] loss 0.096805 (0.059579) train_acc 98.437500 (98.181170)
[10,   801] loss 0.046563 (0.059738) train_acc 98.437500 (98.176108)
[10,   901] loss 0.062853 (0.059433) train_acc 98.437500 (98.203385)
Finished Training
```

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

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

```
Accuracy of the network on the test images: 98.18% - FP32
```

Post-training quantization

Define a new quantized network architeture, 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.

[9]
```
qnet = Net(q=True)
load_model(qnet, net)
fuse_modules(qnet)
```

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

[10]
```
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)
```

```
QConfig(activation=functools.partial(<class 'torch.ao.quantization.observer.MinMaxObserver'>, quant_min=0, quant_max=127){}, weight=functools.pa
Post Training Quantization Prepare: Inserting Observers

 Conv1: After observer insertion

 ConvReLU2d(
  (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), bias=False)
  (1): ReLU()
  (activation_post_process): MinMaxObserver(min_val=inf, max_val=-inf)
)
Post Training Quantization: Calibration done
Post Training Quantization: Convert done

 Conv1: After fusion and quantization

 QuantizedConvReLU2d(1, 6, kernel_size=(5, 5), stride=(1, 1), scale=0.049503736197948456, zero_point=0, bias=False)
Size of model after quantization
Size (MB): 0.050084
```

[11]
```
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: 98.1% - INT8
```

We can also define a cusom 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.

```
[12] from torch.quantization.observer import MovingAverageMinMaxObserver

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

     qnet.qconfig = torch.quantization.QConfig(
                                         activation=MovingAverageMinMaxObserver.with_args(reduce_range=True),
                                         weight=MovingAverageMinMaxObserver.with_args(dtype=torch.qint8, qscheme=torch.per_tensor_symmetric))
     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)
     score = test(qnet, testloader, cuda=False)
     print('Accuracy of the fused and quantized network on the test images: {}% - INT8'.format(score))
```

```
QConfig(activation=functools.partial(<class 'torch.ao.quantization.observer.MovingAverageMinMaxObserver'>, reduce_range=True){}, weight=functool:
Post Training Quantization Prepare: Inserting Observers

 Conv1: After observer insertion

 ConvReLU2d(
  (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), bias=False)
  (1): ReLU()
  (activation_post_process): MovingAverageMinMaxObserver(min_val=inf, max_val=-inf)
)
/usr/local/lib/python3.11/dist-packages/torch/ao/quantization/observer.py:229: UserWarning: Please use quant_min and quant_max to specify the ra
  warnings.warn(
Post Training Quantization: Calibration done
Post Training Quantization: Convert done

 Conv1: After fusion and quantization

 QuantizedConvReLU2d(1, 6, kernel_size=(5, 5), stride=(1, 1), scale=0.0462433323264122, zero_point=0, bias=False)
Size of model after quantization
Size (MB): 0.050084
Accuracy of the fused and quantized network on the test images: 98.18% - INT8
```

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.

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

```
[14] 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){}, weight=functools.partial
Size of model after quantization
Size (MB): 0.050084
```

```
[15] 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: 98.16% - INT8
```

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.

[16]

```
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))
```

```
[5,   701]  loss 0.052019 (0.122098) train_acc 100.000000 (96.291013)
[5,   801]  loss 0.044505 (0.119888) train_acc 98.437500 (96.344413)
[5,   901]  loss 0.130819 (0.119043) train_acc 93.750000 (96.359947)
[6,     1]  loss 0.042558 (0.042558) train_acc 98.437500 (98.437500)
[6,   101]  loss 0.117998 (0.095780) train_acc 96.875000 (97.199876)
[6,   201]  loss 0.153724 (0.099767) train_acc 95.312500 (96.976057)
[6,   301]  loss 0.102752 (0.100420) train_acc 96.875000 (96.947674)
[6,   401]  loss 0.114067 (0.101486) train_acc 95.312500 (96.917862)
[6,   501]  loss 0.076434 (0.100548) train_acc 95.312500 (96.903069)
[6,   601]  loss 0.099132 (0.100521) train_acc 96.875000 (96.939996)
[6,   701]  loss 0.124909 (0.099986) train_acc 95.312500 (96.966387)
[6,   801]  loss 0.094857 (0.099538) train_acc 96.875000 (96.958880)
[6,   901]  loss 0.063304 (0.099211) train_acc 96.875000 (96.965178)
[7,     1]  loss 0.155789 (0.155789) train_acc 98.437500 (98.437500)
[7,   101]  loss 0.121592 (0.084016) train_acc 96.875000 (97.246287)
[7,   201]  loss 0.113657 (0.088692) train_acc 96.875000 (97.162624)
[7,   301]  loss 0.080615 (0.089595) train_acc 98.437500 (97.207226)
[7,   401]  loss 0.035767 (0.089689) train_acc 98.437500 (97.229582)
[7,   501]  loss 0.085524 (0.088710) train_acc 96.875000 (97.249251)
[7,   601]  loss 0.218870 (0.088937) train_acc 89.062500 (97.244176)
[7,   701]  loss 0.104015 (0.088812) train_acc 93.750000 (97.242778)
[7,   801]  loss 0.062049 (0.087247) train_acc 98.437500 (97.280743)
[7,   901]  loss 0.137946 (0.087183) train_acc 95.312500 (97.273862)
[8,     1]  loss 0.144765 (0.144765) train_acc 93.750000 (93.750000)
[8,   101]  loss 0.035153 (0.078542) train_acc 100.000000 (97.617574)
[8,   201]  loss 0.073580 (0.077333) train_acc 93.750000 (97.590174)
[8,   301]  loss 0.037076 (0.076602) train_acc 98.437500 (97.689992)
[8,   401]  loss 0.065163 (0.079205) train_acc 98.437500 (97.611440)
[8,   501]  loss 0.124532 (0.077171) train_acc 93.750000 (97.664047)
[8,   601]  loss 0.026846 (0.076975) train_acc 100.000000 (97.675749)
[8,   701]  loss 0.108788 (0.077499) train_acc 96.875000 (97.643991)
[8,   801]  loss 0.082355 (0.077784) train_acc 98.437500 (97.626014)
[8,   901]  loss 0.110730 (0.077616) train_acc 95.312500 (97.615497)
[9,     1]  loss 0.049956 (0.049956) train_acc 100.000000 (100.000000)
[9,   101]  loss 0.016301 (0.067249) train_acc 100.000000 (98.035272)
[9,   201]  loss 0.020118 (0.071308) train_acc 100.000000 (97.877799)
[9,   301]  loss 0.109771 (0.074577) train_acc 96.875000 (97.783430)
[9,   401]  loss 0.023761 (0.072803) train_acc 100.000000 (97.868610)
[9,   501]  loss 0.032846 (0.072097) train_acc 98.437500 (97.866766)
[9,   601]  loss 0.042937 (0.070304) train_acc 98.437500 (97.914933)
[9,   701]  loss 0.062441 (0.071503) train_acc 96.875000 (97.869116)
[9,   801]  loss 0.008451 (0.071378) train_acc 100.000000 (97.863998)
[9,   901]  loss 0.079512 (0.070993) train_acc 98.437500 (97.866953)
[10,     1]  loss 0.131926 (0.131926) train_acc 98.437500 (98.437500)
[10,   101]  loss 0.035047 (0.062974) train_acc 98.437500 (97.880569)
[10,   201]  loss 0.058336 (0.063175) train_acc 98.437500 (98.041045)
[10,   301]  loss 0.033677 (0.064038) train_acc 98.437500 (98.063746)
[10,   401]  loss 0.029126 (0.066189) train_acc 100.000000 (97.973815)
[10,   501]  loss 0.082996 (0.064965) train_acc 96.875000 (98.000873)
[10,   601]  loss 0.306042 (0.065791) train_acc 96.875000 (97.966930)
[10,   701]  loss 0.051159 (0.064802) train_acc 96.875000 (97.967190)
[10,   801]  loss 0.138415 (0.065150) train_acc 95.312500 (97.955680)
[10,   901]  loss 0.062642 (0.065912) train_acc 96.875000 (97.946726)
Finished Training
Size of model after quantization
Size (MB): 0.050084
Accuracy of the fused and quantized network (trained quantized) on the test images: 97.74% - INT8
```