INF2009 - Lab 5

Lab done on raspberrypi model 3b+

[Part 2. Edit line number 11 as shown below to enable quantization in sample code to use quantized version of MobileNetV2 model.]

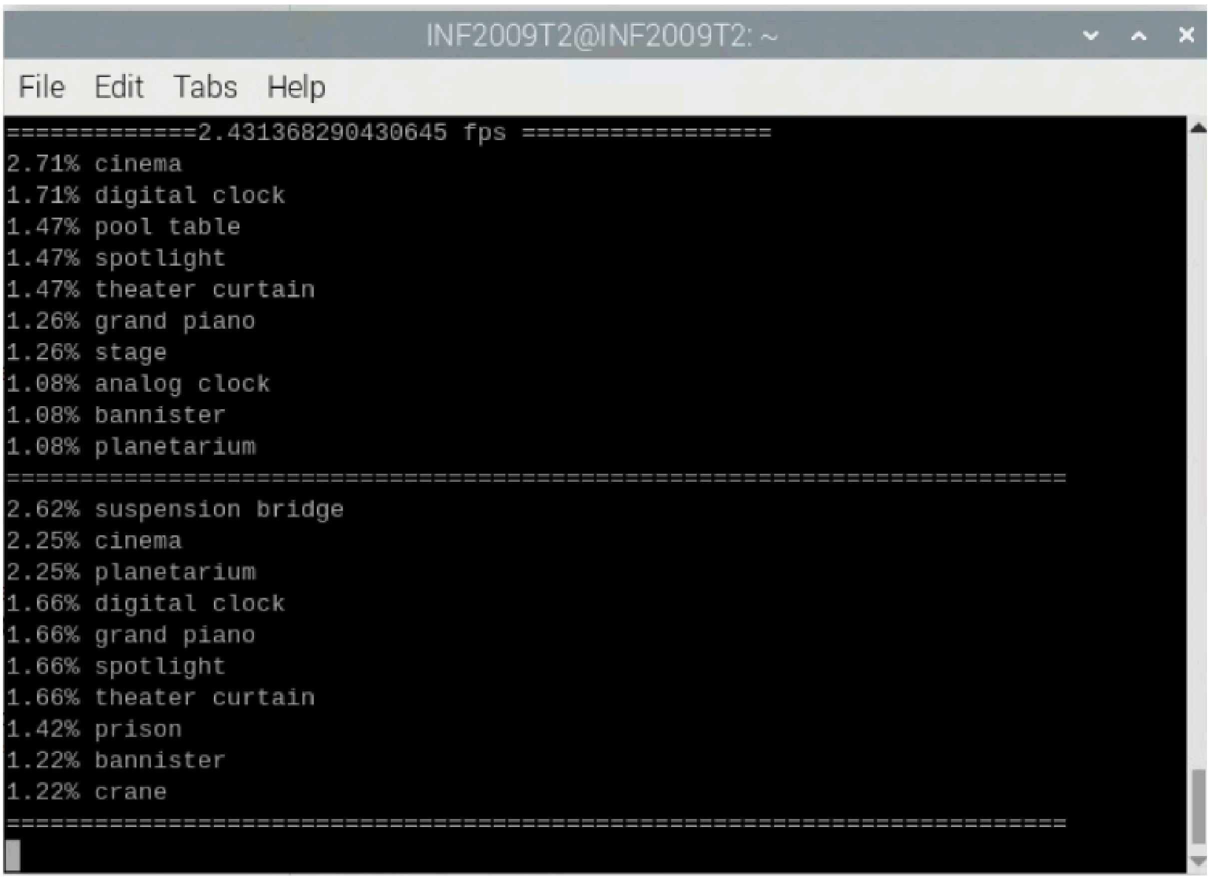
[quantized = False]

```
(dlonedge) INF2009T2@INF2009T2:~ $ python mobile_net.py
/home/INF2009T2/dlonedge/lib/python3.11/site-packages/torchvision/models/_utils.
py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may
 be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/INF2009T2/dlonedge/lib/python3.11/site-packages/torchvision/models/_utils.
py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights'
are deprecated since 0.13 and may be removed in the future. The current 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 weigh
ts.
  warnings.warn(msg)
=============0.39883871604541943 fps =================
=============1.9858312639226556 fps =================
=============2.2185346884068013 fps =================
=============2.242731835090773 fps =================
Illegal instruction
(dlonedge) INF2009T2@INF2009T2:~ $
```

[quantize = True]

```
INF2009T2@INF2009T2: ~
File  Edit  Tabs  Help
 is equivalent to passing `weights=MobileNet_V2_QuantizedWeights.IMAGENET1K_QNNP
ACK_V1`. You can also use `weights=MobileNet_V2_QuantizedWeights.DEFAULT` to get
 the most up-to-date weights.
  warnings.warn(msg)
/home/INF2009T2/dlonedge/lib/python3.11/site-packages/torch/ao/quantization/util
s.py:408: UserWarning: must run observer before calling calculate_qparams. Retur
ning default values.
  warnings.warn(
/home/INF2009T2/dlonedge/lib/python3.11/site-packages/torch/_utils.py:410: UserW
arning: TypedStorage is deprecated. It will be removed in the future and Untyped
Storage will be the only storage class. This should only matter to you if you ar
e using storages directly.  To access UntypedStorage directly, use tensor.untype
d_storage() instead of tensor.storage()
  device=storage.device,
=============0.4585176544925242 fps =================
=============8.956136475686181 fps =================
=============8.718393840242728 fps =================
=============9.376677239441607 fps =================
=============8.718472370871059 fps =================
=============4.084576747179129 fps =================
=============3.826133360821601 fps =================
=============4.271795219643375 fps =================
=============6.896578550888588 fps =================
=============10.321595031031444 fps =================
```

[Part 3. Uncomment lines 57-61 in sample code to print the top 10 predictions in real-time as shown in below video.]



[Please run the sample code preferably in google colab if you do not have computer with good hardware specs.]

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
```

```
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
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.g
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.g
100%|██████████| 28.9k/28.9k [00:00<00:00, 485kB/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, 4.44MB/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):
```

Define some helper functions and classes that help us to track the statistics and accuracy with respect to th

[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.

```
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).

```
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
```

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

```
[5,   301]  loss 0.131167 (0.112947) train_acc 95.312500 (96.397425)
[5,   401]  loss 0.087898 (0.112094) train_acc 96.875000 (96.461970)
[5,   501]  loss 0.060697 (0.111269) train_acc 96.875000 (96.550649)
[5,   601]  loss 0.054562 (0.110437) train_acc 98.437500 (96.591618)
[5,   701]  loss 0.082529 (0.108722) train_acc 96.875000 (96.640959)
[5,   801]  loss 0.097370 (0.107206) train_acc 96.875000 (96.674079)
[5,   901]  loss 0.064080 (0.106664) train_acc 96.875000 (96.696379)
[6,     1]  loss 0.023275 (0.023275) train_acc 100.000000 (100.000000)
[6,   101]  loss 0.038166 (0.085403) train_acc 98.437500 (97.014233)
[6,   201]  loss 0.124207 (0.087366) train_acc 93.750000 (97.092662)
[6,   301]  loss 0.109315 (0.086667) train_acc 95.312500 (97.108596)
[6,   401]  loss 0.129814 (0.090044) train_acc 95.312500 (97.100998)
[6,   501]  loss 0.139331 (0.092687) train_acc 96.875000 (97.049651)
[6,   601]  loss 0.063075 (0.090763) train_acc 96.875000 (97.114185)
[6,   701]  loss 0.030249 (0.090312) train_acc 100.000000 (97.144704)
[6,   801]  loss 0.025079 (0.091385) train_acc 98.437500 (97.157850)
[6,   901]  loss 0.084097 (0.089591) train_acc 96.875000 (97.211432)
[7,     1]  loss 0.163165 (0.163165) train_acc 96.875000 (96.875000)
[7,   101]  loss 0.176784 (0.083949) train_acc 93.750000 (97.400990)
[7,   201]  loss 0.194335 (0.084654) train_acc 95.312500 (97.434701)
[7,   301]  loss 0.110892 (0.087157) train_acc 98.437500 (97.425249)
[7,   401]  loss 0.019905 (0.083416) train_acc 100.000000 (97.502338)
[7,   501]  loss 0.085851 (0.081469) train_acc 98.437500 (97.536178)
[7,   601]  loss 0.141938 (0.080012) train_acc 95.312500 (97.589954)
[7,   701]  loss 0.066964 (0.079655) train_acc 98.437500 (97.601641)
[7,   801]  loss 0.147328 (0.080094) train_acc 95.312500 (97.581149)
[7,   901]  loss 0.033777 (0.080150) train_acc 100.000000 (97.549598)
[8,     1]  loss 0.182345 (0.182345) train_acc 93.750000 (93.750000)
[8,   101]  loss 0.030919 (0.067361) train_acc 100.000000 (97.973391)
[8,   201]  loss 0.032256 (0.069733) train_acc 100.000000 (97.862251)
[8,   301]  loss 0.059822 (0.070260) train_acc 96.875000 (97.824958)
[8,   401]  loss 0.023306 (0.068684) train_acc 100.000000 (97.934850)
[8,   501]  loss 0.036137 (0.069463) train_acc 100.000000 (97.904192)
[8,   601]  loss 0.048703 (0.071087) train_acc 98.437500 (97.849938)
[8,   701]  loss 0.104549 (0.072150) train_acc 95.312500 (97.824536)
[8,   801]  loss 0.078956 (0.071356) train_acc 95.312500 (97.830836)
[8,   901]  loss 0.089153 (0.072547) train_acc 93.750000 (97.792383)
[9,     1]  loss 0.011554 (0.011554) train_acc 100.000000 (100.000000)
[9,   101]  loss 0.042790 (0.068466) train_acc 98.437500 (97.896040)
[9,   201]  loss 0.032765 (0.065282) train_acc 98.437500 (97.955535)
[9,   301]  loss 0.035886 (0.069844) train_acc 98.437500 (97.923588)
[9,   401]  loss 0.209807 (0.069203) train_acc 96.875000 (97.954333)
[9,   501]  loss 0.050870 (0.070198) train_acc 98.437500 (97.894835)
[9,   601]  loss 0.110716 (0.067546) train_acc 95.312500 (97.964330)
[9,   701]  loss 0.039885 (0.066769) train_acc 98.437500 (97.993937)
[9,   801]  loss 0.071653 (0.066144) train_acc 96.875000 (98.020053)
[9,   901]  loss 0.026428 (0.064958) train_acc 98.437500 (98.038638)
[10,     1]  loss 0.017211 (0.017211) train_acc 100.000000 (100.000000)
[10,   101]  loss 0.039179 (0.062078) train_acc 100.000000 (98.081683)
[10,   201]  loss 0.049393 (0.058936) train_acc 98.437500 (98.243159)
[10,   301]  loss 0.031647 (0.057493) train_acc 98.437500 (98.297342)
[10,   401]  loss 0.018421 (0.057240) train_acc 100.000000 (98.281640)
[10,   501]  loss 0.035684 (0.057926) train_acc 98.437500 (98.256612)
[10,   601]  loss 0.051504 (0.057750) train_acc 98.437500 (98.229513)
[10,   701]  loss 0.087400 (0.058567) train_acc 95.312500 (98.205688)
[10,   801]  loss 0.118477 (0.058640) train_acc 96.875000 (98.205368)
[10,   901]  loss 0.040956 (0.059399) train_acc 98.437500 (98.168701)
Finished Training
```

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))
```

```
Accuracy of the network on the test images: 98.24% - 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

```
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=funct
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=0.055636387318372726, 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: 98.2% - 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.

```
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=f
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
  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=0.05271691083908081, 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: 98.21% - 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=
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: 97.73% - 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.

```
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,   601]  loss 0.221678 (0.120936) train_acc 93.750000 (96.201643)
[5,   701]  loss 0.092243 (0.119598) train_acc 98.437500 (96.215228)
[5,   801]  loss 0.198002 (0.118149) train_acc 92.187500 (96.278090)
[5,   901]  loss 0.129455 (0.116718) train_acc 95.312500 (96.335669)
[6,     1]  loss 0.134675 (0.134675) train_acc 95.312500 (95.312500)
[6,   101]  loss 0.072376 (0.100153) train_acc 98.437500 (96.875000)
[6,   201]  loss 0.052070 (0.103848) train_acc 98.437500 (96.851679)
[6,   301]  loss 0.061826 (0.101115) train_acc 98.437500 (96.895764)
[6,   401]  loss 0.153503 (0.099452) train_acc 93.750000 (96.956827)
[6,   501]  loss 0.046255 (0.101139) train_acc 98.437500 (96.856287)
[6,   601]  loss 0.138749 (0.099121) train_acc 96.875000 (96.916597)
[6,   701]  loss 0.147611 (0.099054) train_acc 93.750000 (96.928495)
[6,   801]  loss 0.100683 (0.099622) train_acc 98.437500 (96.873049)
[6,   901]  loss 0.094831 (0.099388) train_acc 96.875000 (96.868063)
[7,     1]  loss 0.084378 (0.084378) train_acc 98.437500 (98.437500)
[7,   101]  loss 0.044826 (0.080451) train_acc 98.437500 (97.447401)
```

```
[10,   501]  loss 0.138968 (0.067275) train_acc 95.312500 (97.938498)
[10,   601]  loss 0.090875 (0.066074) train_acc 95.312500 (97.974730)
[10,   701]  loss 0.043586 (0.065935) train_acc 98.437500 (97.976106)
[10,   801]  loss 0.153884 (0.066267) train_acc 96.875000 (97.965434)
[10,   901]  loss 0.033105 (0.066287) train_acc 98.437500 (97.957131)
Finished Training
Size of model after quantization
Size (MB): 0.050084
Accuracy of the fused and quantized network (trained quantized) on the test images: 95.93% - INT8
```