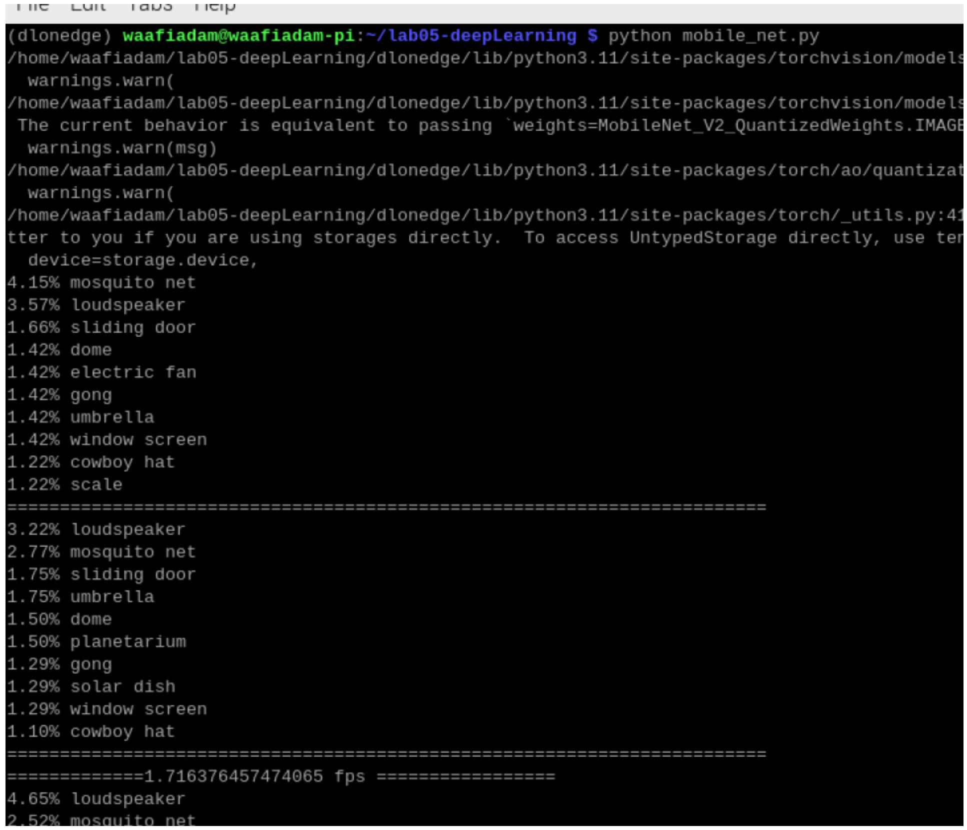
<table>
<tr>
<td>Without quantization</td>
<td><pre>
(dlonedge) waafiadam@waafiadam-pi:~/lab05-deepLearning $ nano mobile_net.py
(dlonedge) waafiadam@waafiadam-pi:~/lab05-deepLearning $ python mobile_net.py
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torchvi
sion/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated
 since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torchvi
sion/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `N
one` for 'weights' are deprecated since 0.13 and may be removed in the future. T
he current behavior is equivalent to passing `weights=MobileNet_V2_Weights.IMAGE
NET1K_V1`. You can also use `weights=MobileNet_V2_Weights.DEFAULT` to get the mo
st up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to
/home/waafiadam/.cache/torch/hub/checkpoints/mobilenet_v2-b0353104.pth
100.0%
=============0.3087357114270822 fps =================
=============0.6692172942956153 fps =================
=============0.6870554795583048 fps =================
=============0.5559767275519126 fps =================
=============0.6837964221160319 fps =================
=============0.6644812510733272 fps =================
=============0.731341847018699 fps =================
=============0.7907629593054308 fps =================
</pre></td>
</tr>
<tr>
<td>With quantization</td>
<td><pre>
(dlonedge) waafiadam@waafiadam-pi:~/lab05-deepLearning $ python mobile_net.py
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torchvisio
  warnings.warn(
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torchvisio
 The current behavior is equivalent to passing `weights=MobileNet_V2_QuantizedWeigh
  warnings.warn(msg)
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torch/ao/q
  warnings.warn(
Downloading: "https://download.pytorch.org/models/quantized/mobilenet_v2_qnnpack_37
100.0%
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torch/_uti
tter to you if you are using storages directly. To access UntypedStorage directly,
  device=storage.device,
=============2.7850482390195888 fps =================
=============6.954545045405686 fps =================
=============7.171428357703352 fps =================
=============6.789512838390181 fps =================
=============7.137562764498985 fps =================
=============7.510490803382073 fps =================
=============7.294203884655935 fps =================
=============7.291220921540745 fps =================
=============7.230424192066417 fps =================
=============7.389799063656637 fps =================
=============7.436308727167398 fps =================
=============7.62210440881067 fps =================
=============7.452623676667707 fps =================
</pre></td>
</tr>
</table>

Print prediction	

```
File  Edit  Tabs  Help
(dlonedge) waafiadam@waafiadam-pi:~/lab05-deepLearning $ python mobile_net.py
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torchvision/models
  warnings.warn(
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torchvision/models
 The current behavior is equivalent to passing `weights=MobileNet_V2_QuantizedWeights.IMAGE
  warnings.warn(msg)
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torch/ao/quantizat
  warnings.warn(
/home/waafiadam/lab05-deepLearning/dlonedge/lib/python3.11/site-packages/torch/_utils.py:41
tter to you if you are using storages directly.  To access UntypedStorage directly, use ter
  device=storage.device,
4.15% mosquito net
3.57% loudspeaker
1.66% sliding door
1.42% dome
1.42% electric fan
1.42% gong
1.42% umbrella
1.42% window screen
1.22% cowboy hat
1.22% scale
=========================================================================
3.22% loudspeaker
2.77% mosquito net
1.75% sliding door
1.75% umbrella
1.50% dome
1.50% planetarium
1.29% gong
1.29% solar dish
1.29% window screen
1.10% cowboy hat
=========================================================================
=============1.716376457474065 fps =================
4.65% loudspeaker
2.52% mosquito net
```

Quantization tutorial

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.

```
[1] !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, 2.2.0, 2.2.1, 2.2.2, 2.3.0, 2.3.1, 2.4.0, 2.4.1, 2.5.0, 2.5.
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, 537kB/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, 4.39MB/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, 10.2MB/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:617: UserWarning: This DataLoader will create 16 worker processes in total. Our suggested max number of worker in current system is
  warnings.warn(
```

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

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

```python
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
```

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

```
[5,   301]  loss 0.130226 (0.095148) train_acc 96.875000 (97.098214)
[5,   401]  loss 0.026708 (0.095899) train_acc 100.000000 (97.042550)
[5,   501]  loss 0.032032 (0.097381) train_acc 100.000000 (97.030938)
[5,   601]  loss 0.141492 (0.095322) train_acc 96.875000 (97.054389)
[5,   701]  loss 0.027958 (0.094612) train_acc 98.437500 (97.082293)
[5,   801]  loss 0.061462 (0.093666) train_acc 98.437500 (97.140293)
[5,   901]  loss 0.018378 (0.093634) train_acc 100.000000 (97.126457)
[6,     1]  loss 0.041896 (0.041896) train_acc 98.437500 (98.437500)
[6,   101]  loss 0.028789 (0.080735) train_acc 100.000000 (97.571163)
[6,   201]  loss 0.083980 (0.079551) train_acc 98.437500 (97.636816)
[6,   301]  loss 0.182107 (0.081990) train_acc 93.750000 (97.513497)
[6,   401]  loss 0.032204 (0.079611) train_acc 100.000000 (97.556889)
[6,   501]  loss 0.023446 (0.082139) train_acc 100.000000 (97.480040)
[6,   601]  loss 0.069212 (0.082962) train_acc 96.875000 (97.444364)
[6,   701]  loss 0.037459 (0.082007) train_acc 98.437500 (97.443384)
[6,   801]  loss 0.047428 (0.082163) train_acc 98.437500 (97.454354)
[6,   901]  loss 0.044934 (0.082328) train_acc 98.437500 (97.447281)
[7,     1]  loss 0.027211 (0.027211) train_acc 100.000000 (100.000000)
[7,   101]  loss 0.061318 (0.079967) train_acc 98.437500 (97.555693)
[7,   201]  loss 0.019661 (0.074515) train_acc 100.000000 (97.807836)
[7,   301]  loss 0.071975 (0.075795) train_acc 96.875000 (97.689992)
[7,   401]  loss 0.197348 (0.073915) train_acc 95.312500 (97.767300)
[7,   501]  loss 0.022026 (0.073806) train_acc 98.437500 (97.729541)
[7,   601]  loss 0.138676 (0.073975) train_acc 96.875000 (97.748544)
[7,   701]  loss 0.142781 (0.073317) train_acc 96.875000 (97.748752)
[7,   801]  loss 0.045697 (0.073197) train_acc 98.437500 (97.748908)
[7,   901]  loss 0.029089 (0.072630) train_acc 100.000000 (97.787181)
[8,     1]  loss 0.086880 (0.086880) train_acc 98.437500 (98.437500)
[8,   101]  loss 0.031957 (0.063239) train_acc 98.437500 (97.957921)
[8,   201]  loss 0.055228 (0.064516) train_acc 96.875000 (97.947761)
[8,   301]  loss 0.064809 (0.066964) train_acc 96.875000 (97.882060)
[8,   401]  loss 0.024150 (0.066456) train_acc 100.000000 (97.891989)
[8,   501]  loss 0.052051 (0.067819) train_acc 98.437500 (97.844935)
[8,   601]  loss 0.030726 (0.069865) train_acc 98.437500 (97.777142)
[8,   701]  loss 0.076662 (0.068869) train_acc 95.312500 (97.811163)
[8,   801]  loss 0.062394 (0.068359) train_acc 98.437500 (97.852294)
[8,   901]  loss 0.038201 (0.066778) train_acc 98.437500 (97.913776)
[9,     1]  loss 0.069726 (0.069726) train_acc 96.875000 (96.875000)
[9,   101]  loss 0.089719 (0.058035) train_acc 96.875000 (98.344678)
[9,   201]  loss 0.042227 (0.061537) train_acc 100.000000 (98.041045)
[9,   301]  loss 0.017454 (0.062003) train_acc 100.000000 (98.037791)
[9,   401]  loss 0.031119 (0.059236) train_acc 98.437500 (98.141365)
[9,   501]  loss 0.023392 (0.059887) train_acc 100.000000 (98.122505)
[9,   601]  loss 0.061657 (0.061412) train_acc 98.437500 (98.104721)
[9,   701]  loss 0.153046 (0.062134) train_acc 93.750000 (98.080867)
[9,   801]  loss 0.121500 (0.061349) train_acc 96.875000 (98.115637)
[9,   901]  loss 0.039490 (0.061769) train_acc 100.000000 (98.113208)
[10,     1]  loss 0.015638 (0.015638) train_acc 98.437500 (98.437500)
[10,   101]  loss 0.037577 (0.053899) train_acc 98.437500 (98.468441)
[10,   201]  loss 0.085149 (0.058886) train_acc 98.437500 (98.180970)
[10,   301]  loss 0.027379 (0.058129) train_acc 98.437500 (98.229859)
[10,   401]  loss 0.114032 (0.058585) train_acc 96.875000 (98.211502)
[10,   501]  loss 0.079467 (0.057953) train_acc 95.312500 (98.241018)
[10,   601]  loss 0.060543 (0.059655) train_acc 98.437500 (98.206115)
[10,   701]  loss 0.062714 (0.057902) train_acc 98.437500 (98.250267)
[10,   801]  loss 0.039739 (0.057576) train_acc 98.437500 (98.248283)
[10,   901]  loss 0.027036 (0.057400) train_acc 98.437500 (98.257145)
Finished Training
```

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

```
[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: 98.48% - 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){}, weig
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=0.05162367224693298, 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: 98.49% - INT8
```

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.
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=0.05162367224693298, zero_point=0, bias=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: 98.49% - 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.

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=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): 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 range
  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=0.04911242797970772, 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: 98.42% - 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(<c
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: 98.39% - 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,   601]  loss 0.104876 (0.113961) train_acc 96.875000 (96.563020)
[5,   701]  loss 0.117084 (0.111989) train_acc 96.875000 (96.605296)
[5,   801]  loss 0.138332 (0.111028) train_acc 95.312500 (96.660424)
[5,   901]  loss 0.051595 (0.110021) train_acc 98.437500 (96.680771)
[6,     1]  loss 0.133016 (0.133016) train_acc 95.312500 (95.312500)
[6,   101]  loss 0.158325 (0.107004) train_acc 96.875000 (96.658416)
[6,   201]  loss 0.051479 (0.103064) train_acc 98.437500 (96.890547)
[6,   301]  loss 0.035492 (0.102322) train_acc 98.437500 (96.906146)
[6,   401]  loss 0.055282 (0.099475) train_acc 98.437500 (97.026964)
[6,   501]  loss 0.054306 (0.098757) train_acc 98.437500 (97.046532)
[6,   601]  loss 0.095016 (0.098169) train_acc 96.875000 (97.028390)
[6,   701]  loss 0.022390 (0.096166) train_acc 100.000000 (97.075606)
[6,   801]  loss 0.201543 (0.095433) train_acc 92.187500 (97.056414)
[6,   901]  loss 0.032593 (0.094473) train_acc 100.000000 (97.088305)
[7,     1]  loss 0.082702 (0.082702) train_acc 96.875000 (96.875000)
[7,   101]  loss 0.061849 (0.083912) train_acc 96.875000 (97.354579)
[7,   201]  loss 0.093935 (0.084946) train_acc 96.875000 (97.349192)
[7,   301]  loss 0.114694 (0.084056) train_acc 93.750000 (97.394103)
[7,   401]  loss 0.053905 (0.083485) train_acc 98.437500 (97.369857)
[7,   501]  loss 0.029696 (0.083732) train_acc 98.437500 (97.386477)
[7,   601]  loss 0.073774 (0.083291) train_acc 96.875000 (97.397567)
[7,   701]  loss 0.101660 (0.082862) train_acc 96.875000 (97.412179)
[7,   801]  loss 0.173280 (0.082633) train_acc 95.312500 (97.425094)
[7,   901]  loss 0.086437 (0.082587) train_acc 96.875000 (97.433407)
[8,     1]  loss 0.091430 (0.091430) train_acc 96.875000 (96.875000)
[8,   101]  loss 0.060509 (0.071013) train_acc 98.437500 (97.741337)
[8,   201]  loss 0.022380 (0.072048) train_acc 100.000000 (97.644590)
[8,   301]  loss 0.110691 (0.075604) train_acc 95.312500 (97.518688)
[8,   401]  loss 0.089144 (0.076106) train_acc 98.437500 (97.588061)
[8,   501]  loss 0.053322 (0.074824) train_acc 98.437500 (97.629741)
[8,   601]  loss 0.031045 (0.075958) train_acc 100.000000 (97.597754)
[8,   701]  loss 0.110432 (0.075953) train_acc 95.312500 (97.612785)
[8,   801]  loss 0.047327 (0.075406) train_acc 98.437500 (97.643571)
[8,   901]  loss 0.030332 (0.075299) train_acc 100.000000 (97.638041)
[9,     1]  loss 0.029518 (0.029518) train_acc 100.000000 (100.000000)
[9,   101]  loss 0.108725 (0.067076) train_acc 96.875000 (97.988861)
[9,   201]  loss 0.013901 (0.069072) train_acc 100.000000 (97.908893)
[9,   301]  loss 0.113233 (0.070484) train_acc 93.750000 (97.866487)
[9,   401]  loss 0.005616 (0.069105) train_acc 100.000000 (97.884196)
[9,   501]  loss 0.127388 (0.068064) train_acc 96.875000 (97.922904)
[9,   601]  loss 0.052610 (0.068528) train_acc 98.437500 (97.886335)
[9,   701]  loss 0.046688 (0.067714) train_acc 98.437500 (97.898092)
[9,   801]  loss 0.027673 (0.067919) train_acc 100.000000 (97.904963)
[9,   901]  loss 0.034674 (0.067362) train_acc 98.437500 (97.927650)
[10,     1]  loss 0.072880 (0.072880) train_acc 98.437500 (98.437500)
[10,   101]  loss 0.061996 (0.061991) train_acc 96.875000 (98.019802)
[10,   201]  loss 0.026138 (0.063950) train_acc 98.437500 (98.056592)
[10,   301]  loss 0.110508 (0.062127) train_acc 96.875000 (98.074128)
[10,   401]  loss 0.034705 (0.063450) train_acc 98.437500 (98.004988)
[10,   501]  loss 0.035081 (0.062326) train_acc 100.000000 (98.038298)
[10,   601]  loss 0.019510 (0.062913) train_acc 100.000000 (97.985129)
[10,   701]  loss 0.025575 (0.063395) train_acc 100.000000 (97.989479)
[10,   801]  loss 0.354063 (0.062254) train_acc 96.875000 (98.047363)
[10,   901]  loss 0.027256 (0.062338) train_acc 100.000000 (98.069853)
Finished Training
Size of model after quantization
Size (MB): 0.050084
Accuracy of the fused and quantized network (trained quantized) on the test images: 97.71% - INT8
```