Neural networks quantization

Today we will deal with neural networks quantization!

Our goal is to reduce network size while keeping the accuracy high! Using smaller number of bits for numbers representation allows for faster inference on edge - embedded devices.

For the purpose of we will Neural networks quantization use Xilinx (now AMD) Brevitas framework and PyTorch. Be aware that there are other frameworks to choose from: build-in PyTorch quantization, Intel's OpenVINO, NVIDIA's TensorRT and others.

Use this link for Brevitas reference and documentation: https://xilinx.github.io/brevitas/index.html.

First, install and import necessary libraries.

In [ ]:
```
!pip3 install brevitas
```

In [1]:
```
import numpy as np
import torch
import brevitas
from torch import nn
import torch
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, RandomRotation
import matplotlib.pyplot as plt
from abc import ABC, abstractmethod
from typing import Any
from typing import Tuple
import tqdm
from brevitas import nn as qnn
from brevitas.core.quant import QuantType
from brevitas.quant import (
    Int8ActPerTensorFloat,
    Int8WeightPerTensorFixedPoint,
    Uint8ActPerTensorFloat,
)
from brevitas.quant import Int16Bias
from brevitas import config
from brevitas.graph.calibrate import bias_correction_mode, calibration_mo
import utils as u
```

In [2]:
```
config.IGNORE_MISSING_KEYS = True
device = torch.device('cuda') if torch.cuda.is_available() else torch.dev
```

```
print(device)
torch.manual_seed(0)
```

```
cpu
```

Out[2]:
```
<torch._C.Generator at 0x10fe23850>
```

Let's start with...

Post-training quantization (PTQ)

Post-training model optimization is the process of applying transforming the model's parameters and values into a more hardware-friendly representation without retraining or fine-tuning. It is the easiest method of quantization, so let's start there.

The use of quantization carries certain implications. These are most easily observed for more complex problems and larger neural networks. However, working with such cases is time-consuming (long training).

As an alternative approach, we will test quantization for a simple task (MNIST classification) but with additional augmentation (more difficult cases) and for a very small network. In addition, we will focus on quantizing floating-point values to INT4 (rather than the more common INT8 used often in practice). In this way, the effects of quantization should be noticeable.

4-bit integer quantization lowers the precision of weights and activations to 4 bits, which leads to significant reduction in the model footprint and significant improvements in inference speed.

So, first, we need a model to quantize. Reuse metric, loss function, train_test_pass and training functions from prevoius exercises.

Train the model for 3 epochs, save the weights (`CNN_epoch3.pth` file) and then train it again for another 3 epochs and save inproved weights (`CNN_epoch6.pth` file). You should get around ~85% accuracy.

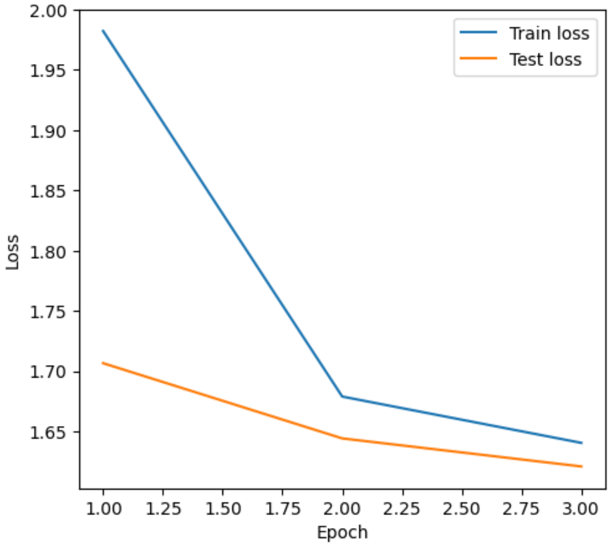
Name the final trained model `fp_model`.

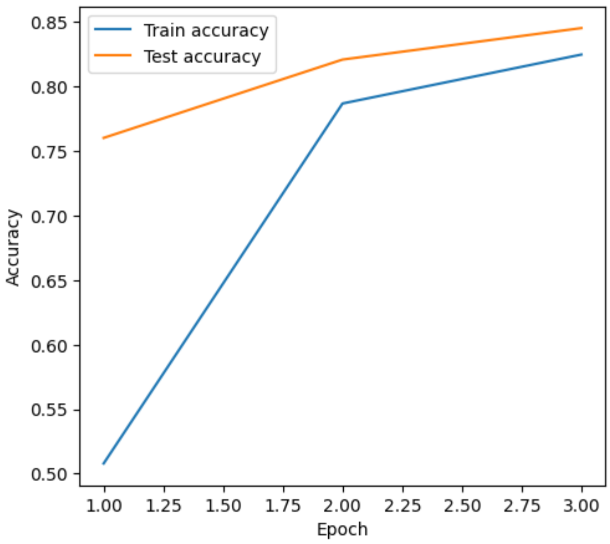
In [3]:
```
class CNN(nn.Module):
    def __init__(self, input_shape, num_of_cls) -> None:
        super().__init__()
        ch_in = input_shape[0]
        self.conv1 = nn.Conv2d(ch_in,8,3,padding=(1,1))
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2,2)

        self.conv2 = nn.Conv2d(8,16,3,padding=(1,1))
        self.relu2 = nn.ReLU()
```

```
        self.pool2 = nn.MaxPool2d(2,2)

        CNN_out_size = 16*7*7

        self.linear = nn.Linear(CNN_out_size, num_of_cls)

    def forward(self, 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)
        x = x.flatten(1)
        x = self.linear(x)
        y = torch.softmax(x,dim=1)
        return y

aug = transforms.Compose([transforms.RandomHorizontalFlip(1), transforms.

train_dataset = datasets.MNIST('data',
                              train=True,
                              download=True,
                              transform=aug)
test_dataset = datasets.MNIST('data',
                              train=False,
                              download=True,
                              transform=aug)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=True)

input_shape = (1, 28, 28)
output_size = 10

FP_MODEL = CNN(input_shape, output_size)
metric = u.AccuracyMetric()
loss_fcn = nn.CrossEntropyLoss()
fpt_optimizer = torch.optim.SGD(FP_MODEL.parameters(), lr=0.01, momentum=

FP_MODEL, history = u.test_or_train(
    model=FP_MODEL,
    train_loader=train_loader,
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=fpt_optimizer,
    update_period=1,
    epoch_max=3,
    device=device,
    mode='both',
    early_stopping_accuracy=0.99
)
```

```
sd = {'model': FP_MODEL.state_dict(), 'opt': fpt_optimizer.state_dict()}
torch.save(sd, './modelsCNN_epoch3.pth')
u.draw_acc_loss(range(1, len(history["loss_train"]) + 1), history)

FP_MODEL, history = u.test_or_train(
    model=FP_MODEL,
    train_loader=train_loader,
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=fpt_optimizer,
    update_period=1,
    epoch_max=3,
    device=device,
    mode="both",
    early_stopping_accuracy=0.99,
)
sd2 = {'model': FP_MODEL.state_dict(), 'opt': fpt_optimizer.state_dict()}
torch.save(sd2, './models/CNN_epoch6.pth')
u.draw_acc_loss(range(1, len(history["loss_train"]) + 1), history)
```

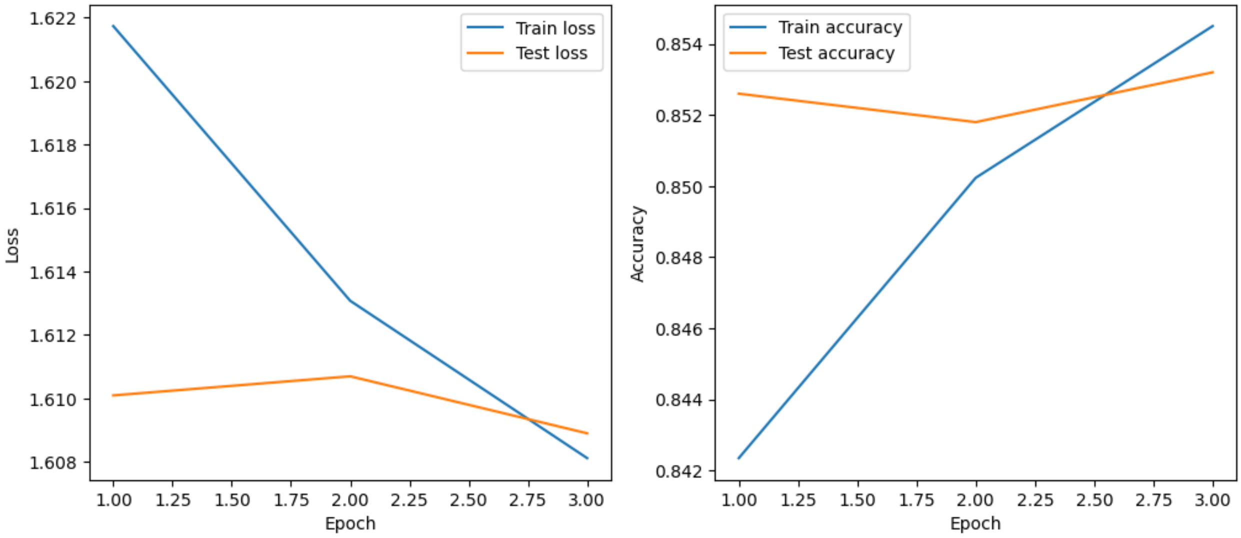
```
Epoch: 1/3
100%|██████████| 938/938 [00:08<00:00, 105.66it/s, accuracy=0.5079, loss=1.9823]
100%|██████████| 157/157 [00:00<00:00, 191.17it/s, accuracy=0.7602, loss=1.7065]
Epoch: 2/3
100%|██████████| 938/938 [00:09<00:00, 99.33it/s, accuracy=0.7869, loss=1.6788]
100%|██████████| 157/157 [00:01<00:00, 146.52it/s, accuracy=0.8209, loss=1.6441]
Epoch: 3/3
100%|██████████| 938/938 [00:09<00:00, 103.36it/s, accuracy=0.8248, loss=1.6403]
100%|██████████| 157/157 [00:00<00:00, 191.20it/s, accuracy=0.8453, loss=1.6208]
```






```
Epoch: 1/3
```

```
100%|██████████| 938/938 [00:09<00:00, 99.71it/s, acc
uracy=0.8424, loss=1.6217]
100%|██████████| 157/157 [00:01<00:00, 136.14it/s, acc
uracy=0.8526, loss=1.6101]
Epoch: 2/3
100%|██████████| 938/938 [00:08<00:00, 109.15it/s, acc
uracy=0.8502, loss=1.6131]
100%|██████████| 157/157 [00:00<00:00, 175.38it/s, acc
uracy=0.8518, loss=1.6107]
Epoch: 3/3
100%|██████████| 938/938 [00:09<00:00, 103.64it/s, acc
uracy=0.8545, loss=1.6081]
100%|██████████| 157/157 [00:00<00:00, 204.22it/s, acc
uracy=0.8532, loss=1.6089]
```



Now - we will quantize this model's weights to INT4.

For this purpose we need to create a new `QuantCNN` class and redefine the model. Instead of `nn.Conv2d` layers use `qnn.QuantConv2d` and instead of `nn.Linear` use `qnn.QuantLinear`. Those layers take the same inputs as previous ones, but additionally we need to pass:

- `weight_bit_width=?` to set the number of bits for weights representation
- `weight_quant=?` to set the method of quantization. Brevitas exposes various pre-made quantizers. We'll use `Int8WeightPerTensorFixedPoint` to represent weights with FixedPoint (i.e. restricting the scale to a power of two).

Define the class, create the model and load the `./CNN_epoch6.pth` weights.

In [4]:
```
class QuantCNN(nn.Module):
    def __init__(self, input_shape, num_of_cls) -> None:
        super().__init__()
        ch_in = input_shape[0]
        self.conv1 = qnn.QuantConv2d(ch_in, 8, 3, padding=(1, 1), weight_
        self.relu1 = nn.ReLU()
```

```
        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv2 = qnn.QuantConv2d(8, 16, 3, padding=(1, 1), weight_bit
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, 2)

        CNN_out_size = 16 * 7 * 7

        self.linear = qnn.QuantLinear(CNN_out_size, num_of_cls, weight_bi

    def forward(self, 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)
        x = x.flatten(start_dim=1)
        x = self.linear(x)
        y = torch.softmax(x, dim=1)
        return y


PTQ_MODEL = QuantCNN(input_shape, output_size)
ptq_optimizer = torch.optim.SGD(PTQ_MODEL.parameters(), lr=0.01, momentum
load_epoch6_file = torch.load('./models/CNN_epoch6.pth')
PTQ_MODEL.load_state_dict(load_epoch6_file['model'])
```

```
/var/folders/0b/brzkvl1j0tn9xzynh5pr39sw0000gn/T/ipykernel_68089/176277123
1.py:32: FutureWarning: You are using `torch.load` with `weights_only=Fals
e` (the current default value), which uses the default pickle module impli
citly. It is possible to construct malicious pickle data which will execut
e arbitrary code during unpickling (See https://github.com/pytorch/pytorc
h/blob/main/SECURITY.md#untrusted-models for more details). In a future re
lease, the default value for `weights_only` will be flipped to `True`. Thi
s limits the functions that could be executed during unpickling. Arbitrary
objects will no longer be allowed to be loaded via this mode unless they a
re explicitly allowlisted by the user via `torch.serialization.add_safe_gl
obals`. We recommend you start setting `weights_only=True` for any use cas
e where you don't have full control of the loaded file. Please open an iss
ue on GitHub for any issues related to this experimental feature.
  load_epoch6_file = torch.load('./models/CNN_epoch6.pth')
```

Out[4]: <All keys matched successfully>

After loading the weights we can inspect the original weight tensor and the quantized version to see the effect:

In [5]:
```
print(f"Original float weight tensor:\n {PTQ_MODEL.conv1.weight} \n")
print(f"Quantized weight QuantTensor:\n {PTQ_MODEL.conv1.quant_weight()}
```

```
Original float weight tensor:
 Parameter containing:
tensor([[[[-0.0795,  0.1402, -0.2875],
```

```
          [-0.3353, -0.1620,  0.1341],
          [ 0.0134,  0.3762,  0.0715]]],


        [[[ 0.0204, -0.2419, -0.2370],
          [-0.3860, -0.3549, -0.2530],
          [-0.1168, -0.0206,  0.1358]]],


        [[[-0.0795, -0.0268,  0.4344],
          [ 0.9703,  0.7768,  1.1522],
          [ 0.6147,  1.0495,  1.2671]]],


        [[[-0.4660, -0.3548, -0.2075],
          [-0.2463,  0.0058, -0.3329],
          [-0.1953, -0.3335, -0.4379]]],


        [[[ 0.0830,  0.6473,  0.3849],
          [ 0.5214,  0.1787, -0.2781],
          [ 0.1019, -0.4364, -0.5207]]],


        [[[ 0.1590,  0.7707,  0.8919],
          [ 0.5647,  1.0712,  1.1912],
          [ 1.1603,  0.9467,  0.5538]]],


        [[[ 0.2382, -0.1793,  0.0526],
          [-0.2941, -0.2341, -0.1737],
          [ 0.1504,  0.1637, -0.1821]]],


        [[[ 0.4105,  0.5792,  0.2914],
          [ 0.4600,  0.4365,  0.4215],
          [ 0.5710, -0.3064, -0.2674]]]], requires_grad=True)

Quantized weight QuantTensor:
 IntQuantTensor(value=tensor([[[[-0.0000,  0.2500, -0.2500],
          [-0.2500, -0.2500,  0.2500],
          [ 0.0000,  0.5000,  0.0000]]],


        [[[ 0.0000, -0.2500, -0.2500],
          [-0.5000, -0.2500, -0.2500],
          [-0.0000, -0.0000,  0.2500]]],


        [[[-0.0000, -0.0000,  0.5000],
          [ 1.0000,  0.7500,  1.2500],
          [ 0.5000,  1.0000,  1.2500]]],
```

```
          [[[-0.5000, -0.2500, -0.2500],
            [-0.2500,  0.0000, -0.2500],
            [-0.2500, -0.2500, -0.5000]]],


          [[[ 0.0000,  0.7500,  0.5000],
            [ 0.5000,  0.2500, -0.2500],
            [ 0.0000, -0.5000, -0.5000]]],


          [[[ 0.2500,  0.7500,  1.0000],
            [ 0.5000,  1.0000,  1.2500],
            [ 1.2500,  1.0000,  0.5000]]],


          [[[ 0.2500, -0.2500,  0.0000],
            [-0.2500, -0.2500, -0.2500],
            [ 0.2500,  0.2500, -0.2500]]],


          [[[ 0.5000,  0.5000,  0.2500],
            [ 0.5000,  0.5000,  0.5000],
            [ 0.5000, -0.2500, -0.2500]]]], grad_fn=<MulBackward0>), scale=
0.25, zero_point=0.0, bit_width=4.0, signed_t=True, training_t=True)
```

Verify the quantized weights. Calculate a maximum, minimum and average difference between the float and int4 weights.

In [6]:
```
quantized_weights = PTQ_MODEL.conv1.quant_weight()
difference = PTQ_MODEL.conv1.weight - quantized_weights
max_diff = difference.abs().max()
min_diff = difference.abs().min()
avg_diff = difference.abs().mean()
print(f"Maximum difference: {max_diff}")
print(f"Minimum difference: {min_diff.item()}")
print(f"Average difference: {avg_diff.item()}")
```

```
Maximum difference: 0.12383553385734558
Minimum difference: 0.0030280351638793945
Average difference: 0.06253571808338165
```

```
/var/folders/0b/brzkvl1j0tn9xzynh5pr39sw0000gn/T/ipykernel_68089/31429215
7.py:2: UserWarning: Defining your `__torch_function__` as a plain method
is deprecated and will be an error in future, please define it as a classm
ethod. (Triggered internally at /Users/runner/work/pytorch/pytorch/pytorc
h/torch/csrc/utils/python_arg_parser.cpp:315.)
  difference = PTQ_MODEL.conv1.weight - quantized_weights
```

Before we evaluate the quantized model we need to calibrate it. The idea of calibration-based quantization is to perform forward passes only with a small set of data and collect statistics to determine scale factors and zero-points. In this way we can acheve much better accuracy.

Calibrate the model and then evaluate it

In [9]:
```
def calibrate_model(calibration_loader, quant_model):
    with torch.no_grad():
        # Put the model in calibration mode to collect statistics
        # Quantization is automatically disabled during the calibration,
        with calibration_mode(quant_model):
            for i, (images, _) in enumerate(calibration_loader):
                if i > 5:
                    break
                quant_model(images)

        # Apply bias correction
        with bias_correction_mode(quant_model):
            for i, (images, _) in enumerate(calibration_loader):
                if i > 5:
                    break
                quant_model(images)
    return quant_model

PTQ_MODEL = calibrate_model(test_loader, PTQ_MODEL)

PTQ_MODEL, history = u.test_or_train(
    model=PTQ_MODEL,
    train_loader=train_loader,
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=ptq_optimizer,
    update_period=1,
    epoch_max=1,
    device=device,
    mode="test",
    early_stopping_accuracy=0.99,
)
print(f'PTQ: loss={history['loss_test'][0]} acc={history['acc_test'][0]}'
```

```
Epoch: 1/1
100%|████████████████████████████████| 157/157 [00:01<00:00, 156.98it/s, acc
uracy=0.8252, loss=1.6363]
PTQ: loss=1.636268610572815 acc=0.8252
```

Compare the resulted accuracy with the accuracy of floating point model.

Try to answer the following questions: How much memory did we save with reducing the precision to just 4 bits?

Now, let's focus on another (much better) approach to quantisation:

Quantization-aware Training (QAT)

Training-time model compression improves model performance by applying optimizations (such as quantization) during the training. The training process minimizes the loss associated with the lower-precision optimizations, so it is able to maintain the model's accuracy while reducing its latency and memory footprint. Generally, training-time model optimization results in better model performance and accuracy than post-training optimization.

Quantization-aware Training is a popular method that allows quantizing a model and applying fine-tuning to restore accuracy degradation caused by quantization. In fact, this is the most accurate quantization method.

Create another model `qat_model` as on object of `QuantCNN` class. This time however load `./CNN_epoch3.pth` weights. In this way we initialise the model with pre-trained weights.

Then, use your `training()` function to train it for 3 more epochs (you just pass the quantised model, you don't have to change anything else).

That's it! You just did a Quantization-aware Treining. Evaluate the model and compare it's result with PTQ and fully-precision models.

Note that we ran as many iterations of training for QAT as we did for PTQ. However, the second half of the training for QAT was conducted with quantization included. The most common practice is to run first part of the learning for floating-point model and then to fine-tune the model after quantization.

In [10]:

```
QAT_MODEL = QuantCNN(input_shape, output_size)
qat_optimizer = torch.optim.SGD(QAT_MODEL.parameters(), lr=0.01, momentum
load_epoch3_file = torch.load("./models/CNN_epoch3.pth")
QAT_MODEL.load_state_dict(load_epoch3_file["model"])

QAT_MODEL, history = u.test_or_train(
    model=QAT_MODEL,
    train_loader=train_loader,
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=qat_optimizer,
    update_period=1,
    epoch_max=3,
    device=device,
    mode="train",
    early_stopping_accuracy=0.99,
)
```

```
QAT_MODEL, history = u.test_or_train(
    model=QAT_MODEL,
    train_loader=train_loader,
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=qat_optimizer,
    update_period=1,
    epoch_max=1,
    device=device,
    mode="test",
    early_stopping_accuracy=0.99,
)
print(f"QAT: loss={history['loss_test'][0]} acc={history['acc_test'][0]}"
```

```
/var/folders/0b/brzkvl1j0tn9xzynh5pr39sw0000gn/T/ipykernel_68089/24909503
2.py:3: FutureWarning: You are using `torch.load` with `weights_only=False
` (the current default value), which uses the default pickle module implic
itly. It is possible to construct malicious pickle data which will execute
arbitrary code during unpickling (See https://github.com/pytorch/pytorch/b
lob/main/SECURITY.md#untrusted-models for more details). In a future relea
se, the default value for `weights_only` will be flipped to `True`. This l
imits the functions that could be executed during unpickling. Arbitrary ob
jects will no longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via `torch.serialization.add_safe_globa
ls`. We recommend you start setting `weights_only=True` for any use case w
here you don't have full control of the loaded file. Please open an issue
on GitHub for any issues related to this experimental feature.
  load_epoch3_file = torch.load("./models/CNN_epoch3.pth")
Epoch: 1/3
100%|██████████████████████████████| 938/938 [00:09<00:00, 99.98it/s, acc
uracy=0.8364, loss=1.6280]
Epoch: 2/3
100%|██████████████████████████████| 938/938 [00:09<00:00, 102.94it/s, acc
uracy=0.8464, loss=1.6169]
Epoch: 3/3
100%|██████████████████████████████| 938/938 [00:09<00:00, 103.53it/s, acc
uracy=0.8515, loss=1.6114]
Epoch: 1/1
100%|██████████████████████████████| 157/157 [00:00<00:00, 169.52it/s, acc
uracy=0.8555, loss=1.6087]
QAT: loss=1.6087489765167236 acc=0.8555
```

Quantisation of activations and bias values

So far we only quantised weights. For embedded applications and real-time processing it is very often necessary to quantise also activation maps and bias values - we need to store all of those in hardware, right?

For feature maps quantization we can use `qnn.QuantIdentity` (among other things). This function takes `bit_width=?` as input, but also `act_quant` (to set the method of quantization). This time we'll use `Int8ActPerTensorFloat` where values are quantized to signed integer with a per-tensor floating-point scale factor. In order to output quanized values along with quantization parameters we need to set `return_quant_tensor=True` parameter.

Let's try it: Create a Torch Tensor with 10 random values between -1 and 1. Define a qnn.QuantIdentity layer (2 bits) and pass the Tensor through it. Then, print both input and output tensors. Analize it, find the floating-point scale. Calculate difference between tensor values

In [12]:
```
# Let's try it: Create a Torch Tensor with 10 random values between -1 an
tensor = torch.randn(10).uniform_(-1, 1)
layer = qnn.QuantIdentity(bit_width=2, act_quant=Int8ActPerTensorFloat, r
output = layer(tensor)
print(f"Input tensor: {tensor} \n")
print(f"Output tensor: {output} \n")
# Calculate difference between tensor values
difference = tensor - output.tensor
max_diff = difference.abs().max()
min_diff = difference.abs().min()
avg_diff = difference.abs().mean()
print(f"Maximum difference: {max_diff}")
print(f"Minimum difference: {min_diff.item()}")
print(f"Average difference: {avg_diff.item()}")
```

```
Input tensor: tensor([ 0.0024,  0.5135, -0.6011, -0.3494, -0.3690, -0.997
6, -0.8753, -0.4639,
        -0.8027,  0.5567])

Output tensor: IntQuantTensor(value=tensor([ 0.0000,  0.4988, -0.4988, -0.
4988, -0.4988, -0.9976, -0.9976, -0.4988,
        -0.9976,  0.4988], grad_fn=<MulBackward0>), scale=0.49879461526870
73, zero_point=0.0, bit_width=2.0, signed_t=True, training_t=True)

Maximum difference: 0.19490134716033936
Minimum difference: 0.0
Average difference: 0.08086554706096649
```

We'are ready for model with quantized weight's, activations and biases! For this purpose define yet another class `QuantCNN_extended` with:

- `qnn.QuantIdentity` before the first Convolution (4 bits)
- `qnn.QuantReLU` instead of ReLU. We don't need to quantize the outputs of convolutional layers, since we have ReLU activations just after each such layer (we could do that tho). We need to pass `bit_width=4`, `return_quant_tensor=True`, and `act_quant=?`. For ReLU we use `Uint8ActPerTensorFloat` (try to figure out - WHY?).
- for each convolutional and linear layer add `bias_quant=Int16Bias`

parameter (we'll use 16 bits for bias. It is not uncommon to use even 32 bits).

Create an object of `QuantCNN_extended`, load `./CNN_epoch3.pth` weights, and train it for 3 epochs. Evaluate the model and compare it's result with PTQ, weights-only QAT and fully-precision models!

In [14]:
```
# We'are ready for model with quantized weight's, activations and biases!

# *   `qnn.QuantIdentity` before the first Convolution (4 bits)
# *   `qnn.QuantReLU` instead of ReLU. We don't need to quantize the outp
# * for each convolutional and linear layer add `bias_quant=Int16Bias` pa

# Create an object of `QuantCNN_extended`, load `./CNN_epoch3.pth` weight

class QuantCNN_extended(nn.Module):
    def __init__(self, input_shape, num_of_cls) -> None:
        super().__init__()
        ch_in = input_shape[0]
        self.identity = qnn.QuantIdentity(bit_width=4, act_quant=Int8ActP
        self.conv1 = qnn.QuantConv2d(ch_in, 8, 3, padding=(1, 1), weight_
        self.relu1 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True,
        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv2 = qnn.QuantConv2d(8, 16, 3, padding=(1, 1), weight_bit
        self.relu2 = qnn.QuantReLU(bit_width=4, return_quant_tensor=True,
        self.pool2 = nn.MaxPool2d(2, 2)

        CNN_out_size = 16 * 7 * 7

        self.linear = qnn.QuantLinear(CNN_out_size, num_of_cls, weight_bi

    def forward(self, x):
        x = self.identity(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)
        x = x.flatten(start_dim=1)
        x = self.linear(x)
        y = torch.softmax(x, dim=1)
        return y

QAT_EXTENDED_MODEL = QuantCNN_extended(input_shape, output_size)
qat_extended_optimizer = torch.optim.SGD(QAT_EXTENDED_MODEL.parameters(),
load_epoch3_file = torch.load("./models/CNN_epoch3.pth")
QAT_EXTENDED_MODEL.load_state_dict(load_epoch3_file["model"])
#train
QAT_EXTENDED_MODEL, history = u.test_or_train(
    model=QAT_EXTENDED_MODEL,
    train_loader=train_loader,
```

```
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=qat_extended_optimizer,
    update_period=1,
    epoch_max=3,
    device=device,
    mode="train",
    early_stopping_accuracy=0.99,
)
#evaluate
QAT_EXTENDED_MODEL, history = u.test_or_train(
    model=QAT_EXTENDED_MODEL,
    train_loader=train_loader,
    test_loader=test_loader,
    loss_fn=loss_fcn,
    metric=metric,
    optimizer=qat_extended_optimizer,
    update_period=1,
    epoch_max=1,
    device=device,
    mode="test",
    early_stopping_accuracy=0.99,
)
print(f'QAT - extended: loss={history['loss_test'][0]} acc={history['acc_
```

```
/var/folders/0b/brzkvl1j0tn9xzynh5pr39sw0000gn/T/ipykernel_68089/185866385
4.py:41: FutureWarning: You are using `torch.load` with `weights_only=Fals
e` (the current default value), which uses the default pickle module impli
citly. It is possible to construct malicious pickle data which will execut
e arbitrary code during unpickling (See https://github.com/pytorch/pytorc
h/blob/main/SECURITY.md#untrusted-models for more details). In a future re
lease, the default value for `weights_only` will be flipped to `True`. Thi
s limits the functions that could be executed during unpickling. Arbitrary
objects will no longer be allowed to be loaded via this mode unless they a
re explicitly allowlisted by the user via `torch.serialization.add_safe_gl
obals`. We recommend you start setting `weights_only=True` for any use cas
e where you don't have full control of the loaded file. Please open an iss
ue on GitHub for any issues related to this experimental feature.
  load_epoch3_file = torch.load("./models/CNN_epoch3.pth")
Epoch: 1/3
100%|██████████| 938/938 [00:17<00:00, 53.35it/s, acc
uracy=0.8352, loss=1.6291]
Epoch: 2/3
100%|██████████| 938/938 [00:16<00:00, 57.02it/s, acc
uracy=0.8458, loss=1.6177]
Epoch: 3/3
100%|██████████| 938/938 [00:15<00:00, 60.06it/s, acc
uracy=0.8512, loss=1.6117]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 122.13it/s, acc
uracy=0.8578, loss=1.6048]
```

```
QAT - extended: loss=1.6048347339630127 acc=0.8578
```

Perfect! We got familiar with QAT and PTQ. Now - Let's do some **science**!

We'll focus on QAT. Try to determine what **are the optimal quantization parameters for our model and MNIST classification**. Create a function that creates, inicializes with `./CNN_epoch3.pth` weights, trains and evaluates the network for the following parameters:

- 2 bits for activations, 8 bits for weights
- 2 bits for activations, 4 bits for weights
- 2 bits for activations, 2 bits for weights
- 4 bits for activations, 8 bits for weights
- 4 bits for activations, 4 bits for weights
- 4 bits for activations, 2 bits for weights
- 8 bits for activations, 8 bits for weights
- 8 bits for activations, 4 bits for weights
- 8 bits for activations, 2 bits for weights

For each model, determine **how much memory is needed to store the weights and activations after each layer** (you don't need to store activations between convolution and ReLU). The number of parameters in the model can be determined by the following function, and the number of values in the activations can be determined by analyzing the network model. Ignore the bias.

When the function is finished, it should display a graph of the dependence of the required memory on the accuracy of the model. Based on your analysis of the graph, answer the question, "What is the best option?"

In [21]:
```python
def calculate_memory(model, bits_weights, bits_activations):
    weight_memory = 0
    activation_memory = 0

    # Calculate weights memory
    for name, layer in model.named_modules():
        if isinstance(layer, (nn.Conv2d, nn.Linear)):
            num_params = layer.weight.numel()
            weight_memory += (
                num_params * bits_weights / 8
            )  # Convert bits to bytes

    # Calculate activations memory
    dummy_input = torch.randn(1, 1, 28, 28)
    model.eval()
    with torch.no_grad():
        device = next(model.parameters()).device
        dummy_input = dummy_input.to(device)
```

```
        activations = model(dummy_input)

    for activation in activations:
        activation_memory += (
            activation.numel() * bits_activations / 8
        )  # Convert bits to bytes

    total_memory = weight_memory + activation_memory
    return total_memory / 1024  # Convert to KB


def create_and_train_model(input_shape, output_size, act_bit_width, weigh
    model = QuantCNN_extended(input_shape, output_size)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9
    load_epoch3_file = torch.load("./models/CNN_epoch3.pth")
    model.load_state_dict(load_epoch3_file["model"])

    for layer in model.children():
        if isinstance(layer, qnn.QuantConv2d) or isinstance(layer, qnn.Qu
            layer.weight_bit_width = weight_bit_width
            layer.act_bit_width = act_bit_width

    model, history = u.test_or_train(
        model=model,
        train_loader=train_loader,
        test_loader=test_loader,
        loss_fn=loss_fcn,
        metric=metric,
        optimizer=optimizer,
        update_period=1,
        epoch_max=3,
        device=device,
        mode="train",
        early_stopping_accuracy=0.99,
    )

    model, history = u.test_or_train(
        model=model,
        train_loader=train_loader,
        test_loader=test_loader,
        loss_fn=loss_fcn,
        metric=metric,
        optimizer=optimizer,
        update_period=1,
        epoch_max=1,
        device=device,
        mode="test",
        early_stopping_accuracy=0.99,
    )

    memory = calculate_memory(model, weight_bit_width, act_bit_width)

    print(f'Configuration: Act: {act_bit_width}, Weight: {weight_bit_widt
```

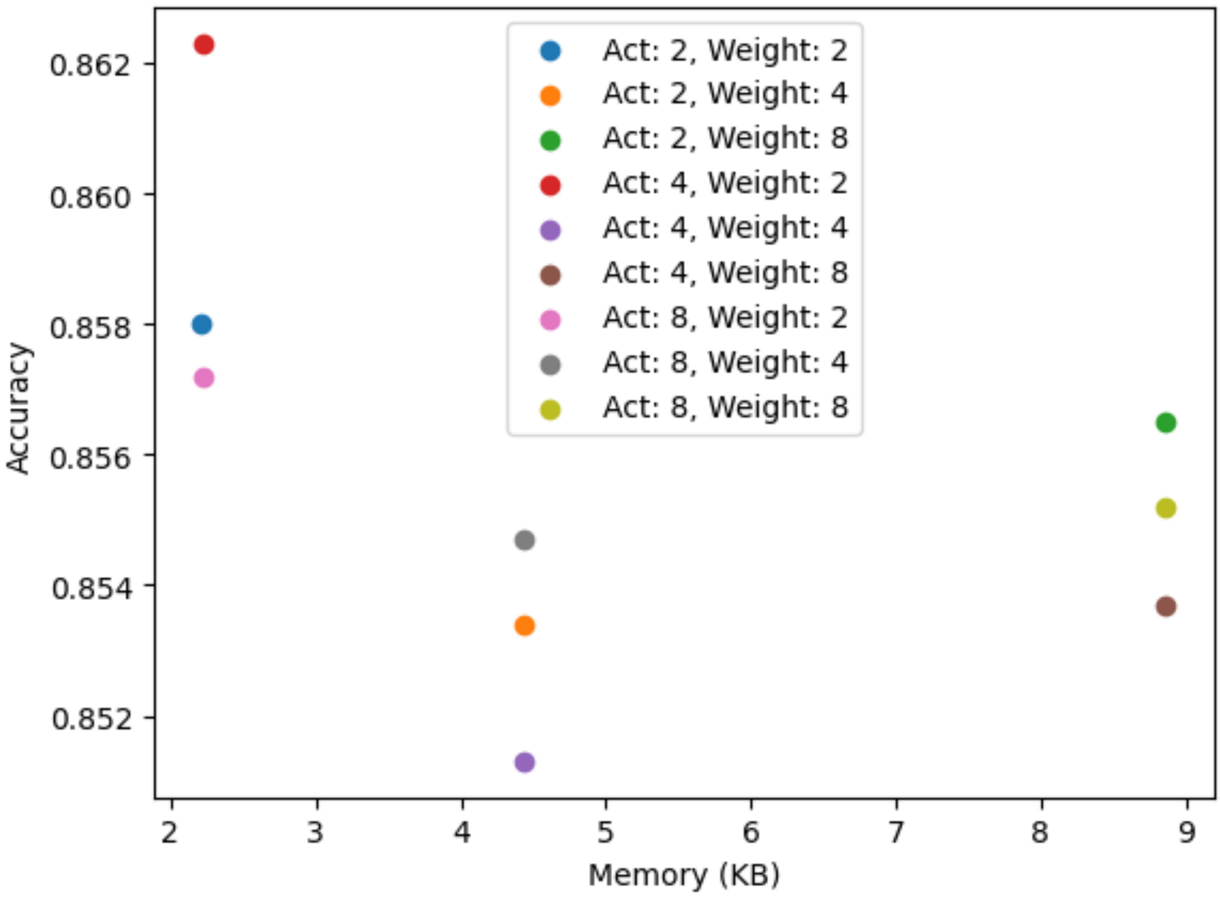
```
    return model, memory, history

act_bit_widths = [2, 4, 8]
weight_bit_widths = [2, 4, 8]
results = []

for act_bit_width in act_bit_widths:
    for weight_bit_width in weight_bit_widths:
        model, memory, history = create_and_train_model(input_shape, outp
        results.append((act_bit_width, weight_bit_width, memory, history)

print(f'Results: {results}')
fig, ax = plt.subplots()
for result in results:
    ax.scatter(result[2], result[3]['acc_test'][0], label=f"Act: {result[
ax.set_xlabel('Memory (KB)')
ax.set_ylabel('Accuracy')
ax.legend()
plt.show()
```

```
/var/folders/0b/brzkvl1j0tn9xzynh5pr39sw0000gn/T/ipykernel_68089/405529085
8.py:51: FutureWarning: You are using `torch.load` with `weights_only=Fals
e` (the current default value), which uses the default pickle module impli
citly. It is possible to construct malicious pickle data which will execut
e arbitrary code during unpickling (See https://github.com/pytorch/pytorc
h/blob/main/SECURITY.md#untrusted-models for more details). In a future re
lease, the default value for `weights_only` will be flipped to `True`. Thi
s limits the functions that could be executed during unpickling. Arbitrary
objects will no longer be allowed to be loaded via this mode unless they a
re explicitly allowlisted by the user via `torch.serialization.add_safe_gl
obals`. We recommend you start setting `weights_only=True` for any use cas
e where you don't have full control of the loaded file. Please open an iss
ue on GitHub for any issues related to this experimental feature.
  load_epoch3_file = torch.load("./models/CNN_epoch3.pth")
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 60.07it/s, acc
uracy=0.8346, loss=1.6294]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 68.50it/s, acc
uracy=0.8456, loss=1.6186]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 68.63it/s, acc
uracy=0.8496, loss=1.6134]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 123.72it/s, acc
uracy=0.8580, loss=1.6057]
Configuration: Act: 2, Weight: 2, Loss: 1.605701612663269, Acc: 0.858, Mem
ory: 2.21533203125 KB
Epoch: 1/3
```

```
100%|██████████| 938/938 [00:15<00:00, 61.89it/s, accuracy=0.8362, loss=1.6286]
Epoch: 2/3
100%|██████████| 938/938 [00:14<00:00, 66.86it/s, accuracy=0.8452, loss=1.6183]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 68.17it/s, accuracy=0.8501, loss=1.6125]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 124.38it/s, accuracy=0.8534, loss=1.6083]
Configuration: Act: 2, Weight: 4, Loss: 1.608261019897461, Acc: 0.8534, Memory: 4.42822265625 KB
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 61.56it/s, accuracy=0.8355, loss=1.6292]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 67.16it/s, accuracy=0.8443, loss=1.6192]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 68.80it/s, accuracy=0.8499, loss=1.6132]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 126.03it/s, accuracy=0.8565, loss=1.6060]
Configuration: Act: 2, Weight: 8, Loss: 1.6059625471115113, Acc: 0.8565, Memory: 8.85400390625 KB
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 62.31it/s, accuracy=0.8360, loss=1.6282]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 68.46it/s, accuracy=0.8461, loss=1.6178]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 68.58it/s, accuracy=0.8505, loss=1.6122]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 127.51it/s, accuracy=0.8623, loss=1.6015]
Configuration: Act: 4, Weight: 2, Loss: 1.6014500715255737, Acc: 0.8623, Memory: 2.2177734375 KB
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 62.27it/s, accuracy=0.8365, loss=1.6281]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 68.76it/s, accuracy=0.8436, loss=1.6197]
Epoch: 3/3
```

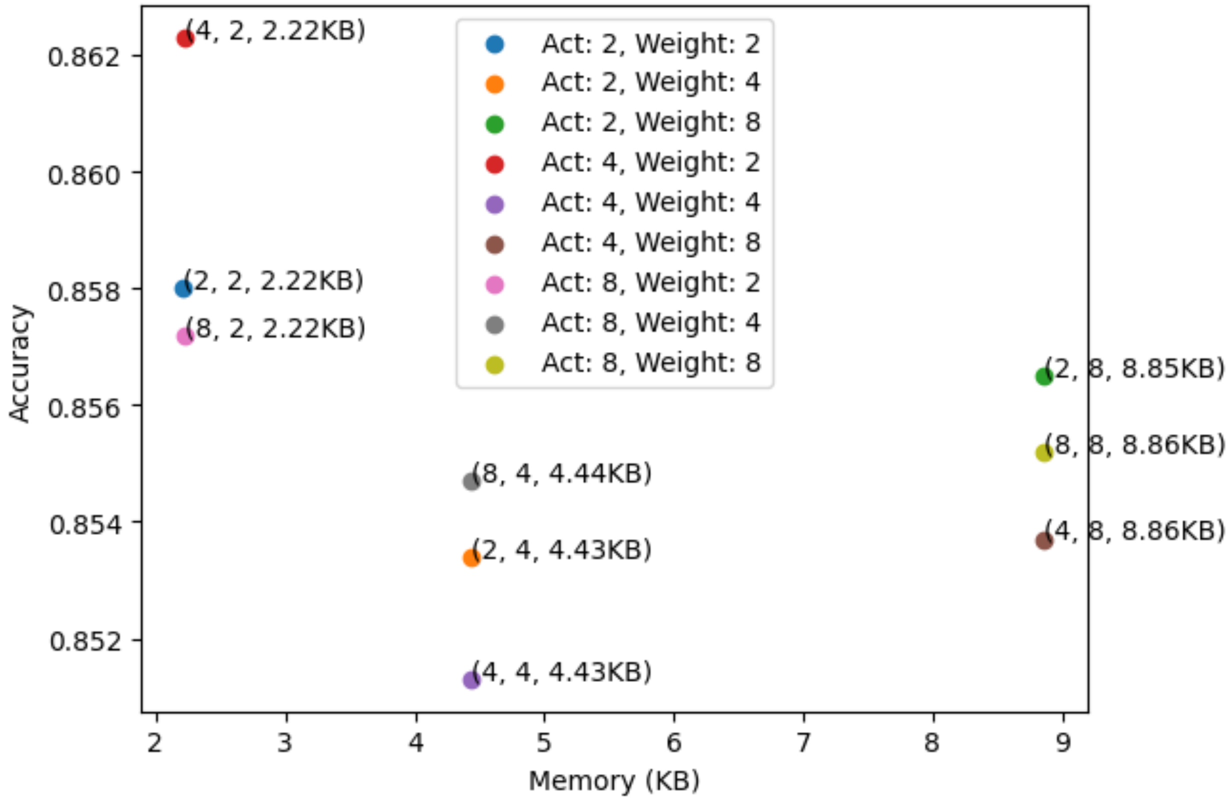
```
100%|██████████| 938/938 [00:13<00:00, 68.49it/s, accuracy=0.8508, loss=1.6122]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 122.57it/s, accuracy=0.8513, loss=1.6137]
Configuration: Act: 4, Weight: 4, Loss: 1.613733044052124, Acc: 0.8513, Memory: 4.4306640625 KB
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 61.92it/s, accuracy=0.8350, loss=1.6297]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 68.34it/s, accuracy=0.8435, loss=1.6193]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 68.61it/s, accuracy=0.8490, loss=1.6143]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 124.15it/s, accuracy=0.8537, loss=1.6078]
Configuration: Act: 4, Weight: 8, Loss: 1.6077985324859618, Acc: 0.8537, Memory: 8.8564453125 KB
Epoch: 1/3
100%|██████████| 938/938 [00:14<00:00, 62.67it/s, accuracy=0.8361, loss=1.6281]
Epoch: 2/3
100%|██████████| 938/938 [00:14<00:00, 66.19it/s, accuracy=0.8445, loss=1.6189]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 68.01it/s, accuracy=0.8497, loss=1.6131]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 122.76it/s, accuracy=0.8572, loss=1.6057]
Configuration: Act: 8, Weight: 2, Loss: 1.6057231525421143, Acc: 0.8572, Memory: 2.22265625 KB
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 61.78it/s, accuracy=0.8358, loss=1.6284]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 68.68it/s, accuracy=0.8439, loss=1.6192]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 67.79it/s, accuracy=0.8501, loss=1.6128]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 118.42it/s, accuracy=0.8547, loss=1.6072]
```

```
Configuration: Act: 8, Weight: 4, Loss: 1.6072398246765136, Acc: 0.8547, M
emory: 4.435546875 KB
Epoch: 1/3
100%|██████████| 938/938 [00:15<00:00, 61.39it/s, acc
uracy=0.8351, loss=1.6293]
Epoch: 2/3
100%|██████████| 938/938 [00:13<00:00, 69.13it/s, acc
uracy=0.8445, loss=1.6190]
Epoch: 3/3
100%|██████████| 938/938 [00:13<00:00, 70.55it/s, acc
uracy=0.8508, loss=1.6124]
Epoch: 1/1
100%|██████████| 157/157 [00:01<00:00, 124.74it/s, acc
uracy=0.8552, loss=1.6078]
Configuration: Act: 8, Weight: 8, Loss: 1.6078474243164063, Acc: 0.8552, M
emory: 8.861328125 KB
Results: [(2, 2, 2.21533203125, {'loss_train': [], 'acc_train': [], 'loss_
test': [1.605701612663269], 'acc_test': [0.858]}), (2, 4, 4.42822265625,
{'loss_train': [], 'acc_train': [], 'loss_test': [1.608261019897461], 'acc
_test': [0.8534]}), (2, 8, 8.85400390625, {'loss_train': [], 'acc_train':
[], 'loss_test': [1.6059625471115113], 'acc_test': [0.8565]}), (4, 2, 2.21
77734375, {'loss_train': [], 'acc_train': [], 'loss_test': [1.601450071525
5737], 'acc_test': [0.8623]}), (4, 4, 4.4306640625, {'loss_train': [], 'ac
c_train': [], 'loss_test': [1.613733044052124], 'acc_test': [0.8513]}), (
4, 8, 8.8564453125, {'loss_train': [], 'acc_train': [], 'loss_test': [1.60
77985324859618], 'acc_test': [0.8537]}), (8, 2, 2.22265625, {'loss_train':
[], 'acc_train': [], 'loss_test': [1.6057231525421143], 'acc_test': [0.857
2]}), (8, 4, 4.435546875, {'loss_train': [], 'acc_train': [], 'loss_test':
[1.6072398246765136], 'acc_test': [0.8547]}), (8, 8, 8.861328125, {'loss_t
rain': [], 'acc_train': [], 'loss_test': [1.6078474243164063], 'acc_test':
[0.8552]})]
```



In [23]:

```
fig, ax = plt.subplots()
for result in results:
    ax.scatter(
        result[2],
        result[3]["acc_test"][0],
        label=f"Act: {result[0]}, Weight: {result[1]}",
    )
    ax.text(result[2], result[3]["acc_test"][0], f"({result[0]}, {result[
ax.set_xlabel("Memory (KB)")
ax.set_ylabel("Accuracy")
ax.legend()
plt.show()
```



Act: 2, Weight: 2
Act: 2, Weight: 4
Act: 2, Weight: 8
Act: 4, Weight: 2
Act: 4, Weight: 4
Act: 4, Weight: 8
Act: 8, Weight: 2
Act: 8, Weight: 4
Act: 8, Weight: 8
(4, 2, 2.22KB)
(2, 2, 2.22KB)
(8, 2, 2.22KB)
(8, 4, 4.44KB)
(2, 4, 4.43KB)
(4, 4, 4.43KB)
(2, 8, 8.85KB)
(8, 8, 8.86KB)
(4, 8, 8.86KB)
Accuracy
0.862
0.860
0.858
0.856
0.854
0.852
2
3
4
5
6
7
8
9
Memory (KB)