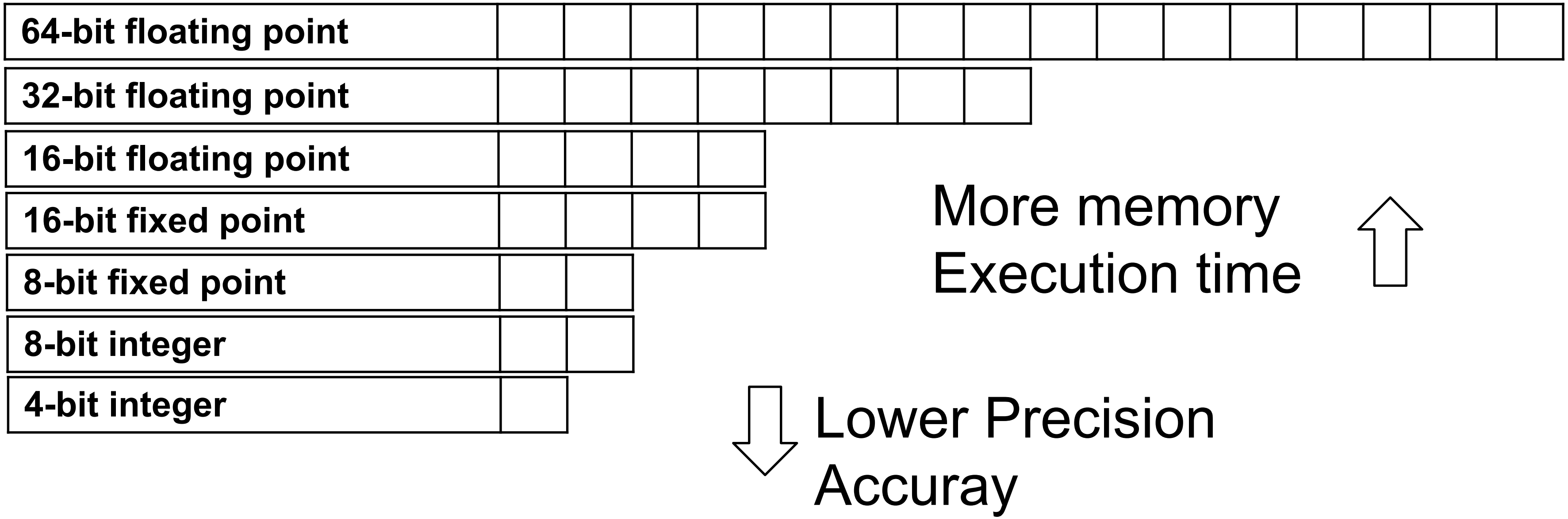
Computer Vision for Embedded Systems

Yung-Hsiang Lu
Purdue University
yunglu@purdue.edu

Reduce Network Sizes

- Reduce #bits for each parameter
- Remove unused connections between layers
- Remove inactive neurons



Benefits of Quantization

- Reduced memory footprint
- Sparse weights
- Faster inference time
- Regularization
- Better cache performance

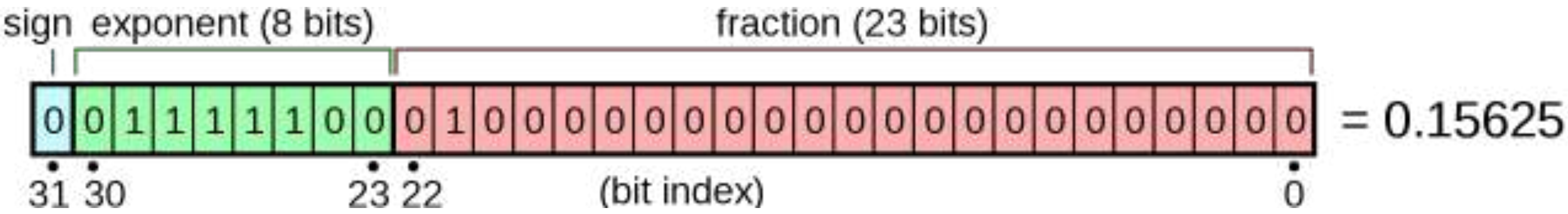
Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, Xiaotong Zhang, Pruning and quantization for deep neural network acceleration: A survey, Neurocomputing, Volume 461, 2021, Pages 370-403, ISSN 0925-2312, https://doi.org/10.1016/j.neucom.2021.07.04

Sparsity

- **Definition:** % of weights that are equal to 0

- **Benefit:** Specialized hardware/software can make computations faster given sparse data
  Fast Sparse Matrix Multiplication
  https://www.cs.tau.ac.il/~zwick/papers/sparse.pdf

32-bit Floating Point



- sign: 0 positive, 1 negative
- 8-bit exponent: e - 127. 01111100 = 124 ⇒ $2^{124-127} = 2^{-3}$
- 1 + fraction. 0100 = 2^{-2}
- 2^{-3} x 1.25 = 0.15625

wikipedia.org

Yung-Hsiang Lu, Purdue University

Adding Two Floating Point Numbers

How to add 3.75 and 5.125 to get 8.875?

$3.75 = 1.875 \times 2 = (1 + 0.875) \times 2$

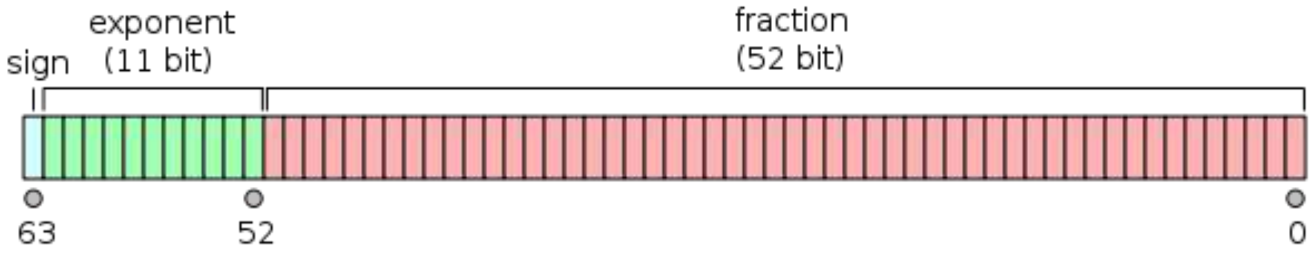
$0.875 = 0.5 + 0.25 + 0.125 = 2^{-1} + 2^{-2} + 2^{-3}$

$5.125 = 1.28125 \times 2^2 = (1 + 0.28125) \times 2^2$

$0.28125 = 0.25 + 0.03125 = 2^{-2} + 2^{-5}$

A. make them have the same exponent
B. add the fraction
C. convert back to the correct format

64-bit Floating Point

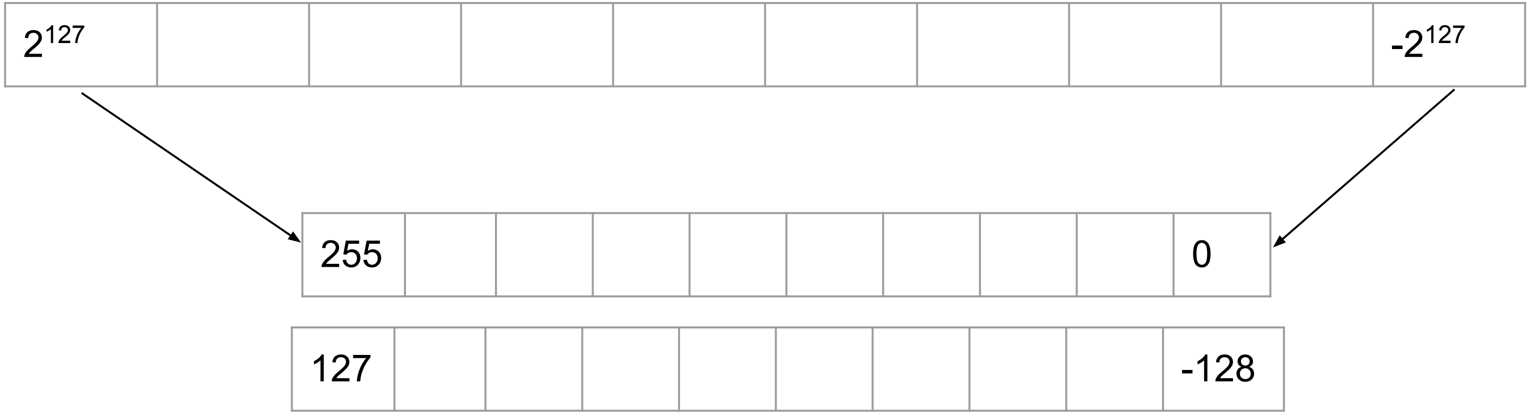


- sign: 0 positive, 1 negative
- 11-bit exponent: e - 1023.
- 1 + fraction

wikipedia.org

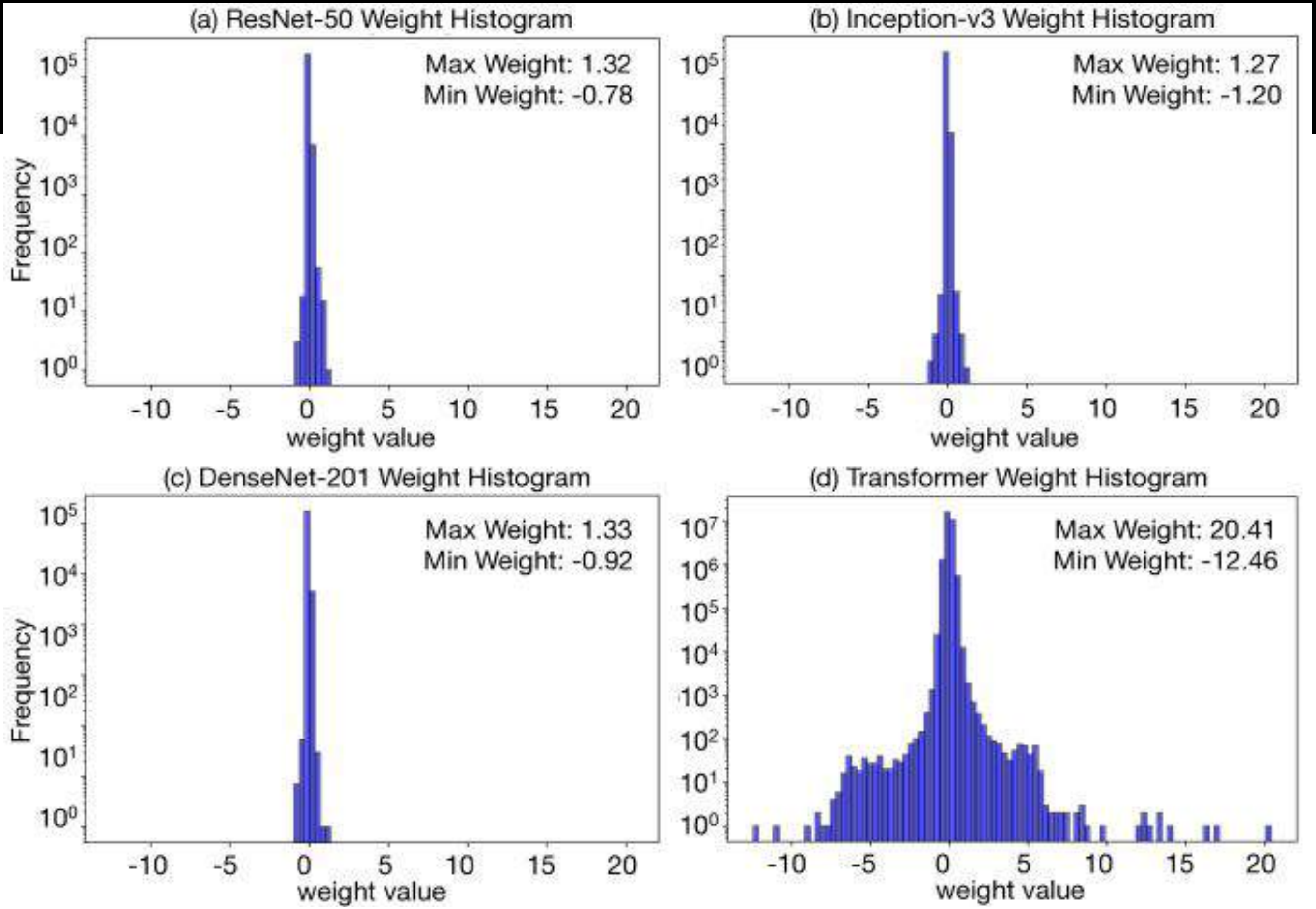
Quantization is lossy mapping

32-bit floating point as large as 2^{127} >> 255 in 8-bit integer



In reality, neural networks' weights are rarely too large

Yung-Hsiang Lu, Purdue University



AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference, https://arxiv.org/abs/1909.13271

Quantization as Linear Mapping

floating point $x \in [\alpha, \beta]$ integer $x_q \in [\alpha_q, \beta_q]$

$$x = c(x_q + d) \qquad x_q = [\frac{x}{c} - d]$$

$$\alpha = c(\alpha_q + d) \qquad \beta = c(\beta_q + d)$$

$$c = \frac{\beta - \alpha}{\beta_q - \alpha_q} \qquad d = \frac{\alpha\beta_q - \beta\alpha_q}{\beta - \alpha}$$

scale

zero point

https://leimao.github.io/article/Neural-Networks-Quantization/

Method	Inference Latency	Accuracy Lose	Training Data
Dynamic Quantization	Usually faster	Small	No need
Static Quantization	Faster	Larger	Unlabeled
Quantization Aware Training	Faster	Small	Labeled

Post-training Static Quantization Overview

trained
32-bit floating point model

Calibrate layer outputs using a dataset representative of the target domain

Map weights, biases to 8-bit integers

Per-layer calibration data

trained
8-bit integer model

During inference, the smaller 8-bit model and the calibration data are used to perform quantized operations

Unfortunately, this can result in significant loss of accuracy

Vincent Vanhoucke, Andrew Senior, & Mark Z. Mao (2011). Improving the speed of neural networks on CPUs. In Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011.

Quantization-Aware Training Overview

Forward pass

Round all floating-point weights, biases, and activations to the nearest quantized integer

Backwards propagation

Use floating-point arithmetic as usual

This way, weights can still be adjusted incrementally, while influenced by the quantized forward pass

This still produces a 32-bit floating-point model. However, the weights and biases should now be much more amenable to static quantization, resulting in higher-accuracy than post-training static quantization.

B. Jacob et al., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704-2713, doi: 10.1109/CVPR.2018.00286.

32 bits to 8 bits: Example MatMul Layer

Let's say we have a **matrix multiplication layer** for fp32 tensors input **X** and weight **W.** It'll give us output **O** (note, SF: scale factor, ZP: zero point). You could do similar stuff for Conv2D and Linear layers.

$$O_{fp32} = X_{fp32} \times W_{fp32}$$

$$SF_{O,fp32}(O_{int8} - ZP_{O,int8}) = SF_{X,fp32}(X_{int8} - ZP_{X,int8}) \times SF_{W,fp32}(W_{int8} - ZP_{W,int8})$$

$$O_{int8} = \boxed{\left(\frac{1}{SF_{O,fp32}}\right)}\left(SF_{X,fp32}(X_{int8} - ZP_{X,int8}) \times SF_{W,fp32}(W_{int8} - ZP_{W,int8})\right) + \boxed{ZP_{O,int8}}$$

All this math is done using only quantized tensors, scale factors, and zero points. No massive FP32 tensors

how do we know the scale factor and zero point for **O?**

32 bits to 8 bits: Quantizing an entire NN

1. Do calibration on multiple images from dataset

fp32 input

fp32 Conv2d

Store output A min, MAX

Use avg mins, maxes to get A's SF and ZP

output A

fp32 Linear 1

Store output B min, MAX

Use avg mins, maxes to get B's SF and ZP

output B

fp32 Linear 2

fp32 output

2. Quantize weights and biases

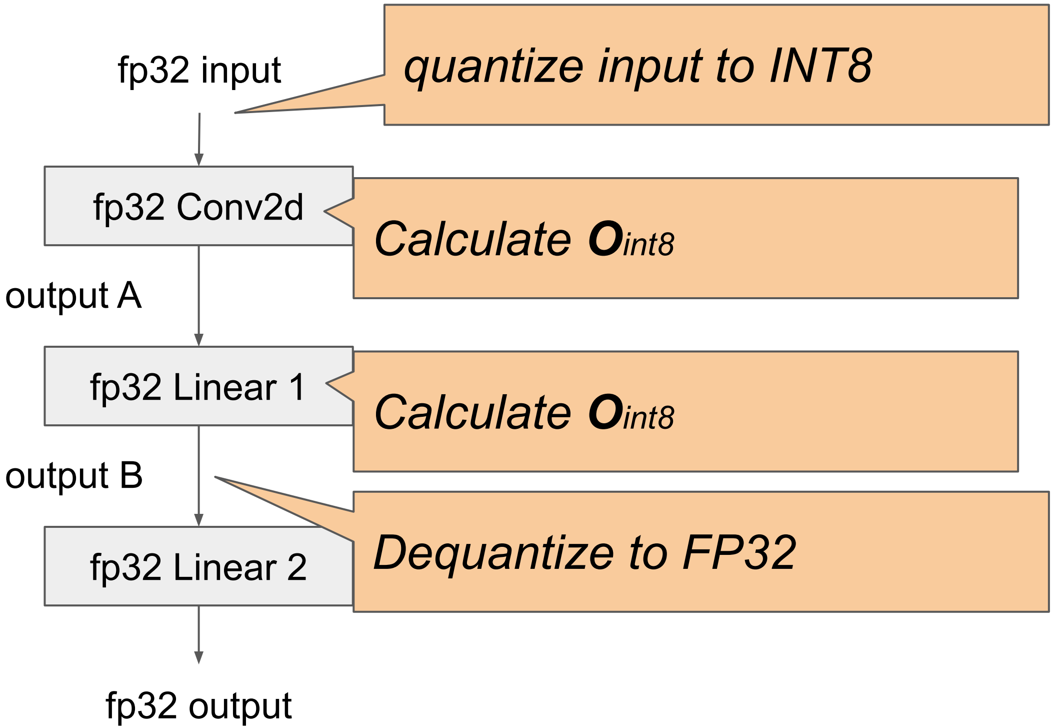
Use weight/bias mins and maxes to calculate SF's and ZP's to get quantized weight/bias

fp32 Conv2d

fp32 Linear 1

fp32 Linear 2

32 bits to 8 bits: Quantized Inference

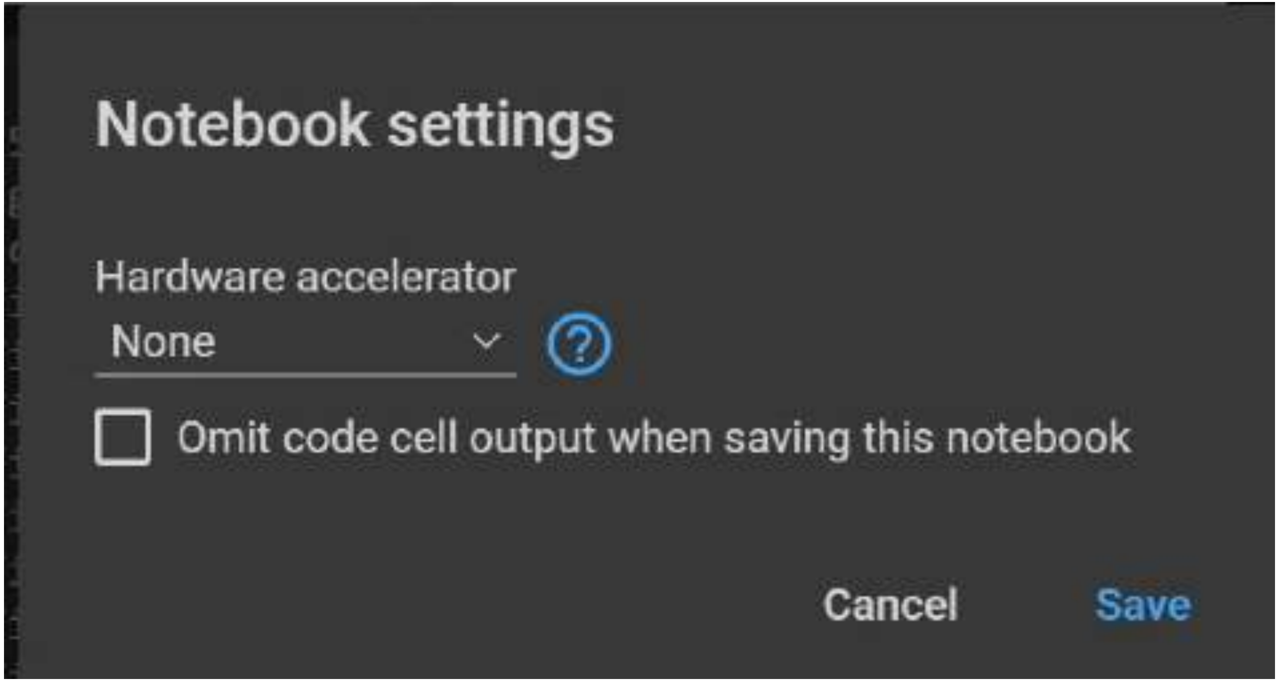


Yung-Hsiang Lu, Purdue University

Post-Training Static Quantization Assignment

Google Colab

Only need to use a CPU runtime in Colab



Get setup

Setup

TASKS:

1. Run these cells to grab the PyPI packages and import the dependencies for the notebook. You can click into the "Files" explorer on the sidebar to confirm that `./ClassyClassifierParams.pt` was appropriately downloaded.

```
[1]
!pip install torchinfo
!pip install gdown
!gdown https://drive.google.com/uc?id=1xr1g4FfhV_znqbg1KTThK44vblAqEvmd -O ./ClassyClassifierParams.pt
```

```
Collecting torchinfo
  Downloading torchinfo-1.5.3-py3-none-any.whl (19 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.5.3
Requirement already satisfied: gdown in /usr/local/lib/python3.7/dist-packages (3.6.4)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from gdown) (1.15.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from gdown) (2.23.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from gdown) (4.62.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests->gdown) (2021.5.30)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests->gdown) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->gdown) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->gdown) (2.10)
Downloading...
From: https://drive.google.com/uc?id=1xr1g4FfhV_znqbg1KTThK44vblAqEvmd
To: /content/ClassyClassifierParams.pt
100% 516k/516k [00:00<00:00, 34.1MB/s]
```

```
[2]
import copy
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import os
```


Yung-Hsiang Lu, Purdue University

Check out the `ClassyClassifier`

```python
CLASSY_CLASSIFIER_PARAMETERS_FILENAME = "./ClassyClassifierParams.pt"

class ClassyClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        # INPUT SHAPE: Bx3x32x32 (B is for "batch size," in this case: 5)
        self.layer1_conv = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1) # output Bx16x28x28
        self.layer2_pool = nn.MaxPool2d(kernel_size=2, stride=2) # output Bx16x14x14
        self.layer3_conv = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1) # output Bx32x10x10
        self.layer4_pool = nn.MaxPool2d(kernel_size=2, stride=2) # output Bx32x5x5
        self.layer5_flat = nn.Flatten() # output Bx(32x5x5=800)
        self.layer6_fc = nn.Linear(in_features=800, out_features=128) # output Bx128
        self.layer7_fc = nn.Linear(in_features=128, out_features=84) # output Bx84
        self.layer8_fc = nn.Linear(in_features=84, out_features=10) # output Bx10

    def forward(self, x: torch.Tensor):
        x = F.relu(self.layer1_conv(x))
        x = self.layer2_pool(x)
        x = F.relu(self.layer3_conv(x))
        x = self.layer4_pool(x)
        x = self.layer5_flat(x)
        x = F.relu(self.layer6_fc(x))
        x = F.relu(self.layer7_fc(x))
        x = self.layer8_fc(x)
        return x
```

Quantizing by hand...

Quantization formula
To quantize a tensor, you need to **use its min and max** values to calculate the **scale factor** and **zero point.**

$$Value_{fp32} = ScaleFactor_{fp32} \times (Value_{int8} - ZeroPoint_{int8})$$

Generate this

And this

To calculate this

Quantization helper functions

Helper functions to calculate scalefactor/zeropoint, quantize/dequan

```python
def calculate_scalefactor_and_zeropoint(min_val: float, max_val: float) -> Tuple[
    """Calculates the scale factor and zero point to quantize from a [`min_val`,

    Follows the quantization formula: FP32 VALUE = scalefactor * (QUANT INT8 VALU

    Using the formula, float value `min_val` quantizes to `INT8_MIN` of 0.

    Args:
        min_val (float): Minimum value in range of interest.
        max_val (float): Maximum value in range of interest.

    Returns:
        tuple[float, int]: Scale factor, zero point
    """
    INT8_MIN = 0
    INT8_MAX = 255

    scalefactor =  # TODO: fill this in such that you can scale from [min_val, max_val] to [0, 255]
    zeropoint =  # TODO: fill this in

    # This clamps the zero point appropriately into the range [0, 255]
    if zeropoint < INT8_MIN:
        zeropoint = INT8_MIN
    elif zeropoint > INT8_MAX:
        zeropoint = INT8_MAX

    return scalefactor, int(zeropoint)
```

Read the docstrings
Complete the `#TODO`s
- Four 1-line `#TODO`s

Technically, this is *pseudo*-quantization

Yes, we still store as `FloatTensors.`
But these tensors are still integers, in range [0, 255]. The point is to learn quantization arithmetic!

You will use `torch.fx` **later** to perform real quantization.

Fill out `QuantizedLayer`

1. Constructor should call quantize_tensor() to get int8 weights/biases with scale factors and zero points
2. run_quantized_layer uses the int8 tensors along with their scale factors and zero points to calculate the floating point output. The floating point output is converted to int8 using some precalculated scalefactor and zero point

Again, remember that this is to learn the math. Real hardware implements this symbolically and much quicker!

Tip: refer to the "Example MatMul Layer" slide in the Quantization lecture slide deck to see how the math works

Read the docstrings
Complete the `#TODO`s

Fill out `QuantizerForClassyClassifier`

Calibration requires that, for each image batch, you store the min/max of each layer's output (i.e. the preceding layer's input) in lists.

Once this is done, you can average the lists and use the average min/max to calculate scalefactors and zero points (remember, these are needed as parameters for QuantizedLayer.run_quantized_layer())

We've done the first layer for you!

Fill out `QuantizerForClassyClassifier`

```python
    def run_calibrated_quantized(self, x: torch.Tensor) -> torch.Tensor:
        """Runs the pseudo-quantized ClassyClassifier on a 32-bit float tensor.

        Quantizes the tensor to [0, 255] prior to forward pass, and then dequantizes back to float before return.

        Args:
            x (torch.Tensor): Input 32-bit float tensor.

        Returns:
            torch.Tensor: Output 32-bit float tensor.
        """
        x = copy.deepcopy(x)

        # Use the calibration input stats for layer 1 to quantize the tensor and run the layer
        x, x_scalefactor, x_zeropoint = quantize_tensor(x, self.calibration_input_stats["layer1_conv"]["avg_min"], self.calibration_input_stats["l
        x = self.int8_layer1_conv.run_quantized_layer(x, x_scalefactor, x_zeropoint, self.calibration_input_stats["layer3_conv"]["input_scalefacto
        x = F.relu(x)
        x = self.fp32_model.layer2_pool(x)

        # TODO: As above, use the calibration input stats for the appropriate layers to complete the rest of the forward pass: layer 3-7 (DON'T DO
```


Yung-Hsiang Lu, Purdue University

Accuracy should be similar!

Run the cells.
Your output should look like this image

Trying out a real quantization library

Run the cells.
Your output should look like this image

Note that accuracy remains similar, but the FX-Quantized network is much smaller than the original!

```
ACCURACY:
ORIGINAL ClassyClassifier:
Accuracy: 0.687, 6874 correc
Model Size (kB): 539.1
FX-QUANTIZED ClassyClassifier:
Accuracy: 0.690, 6901 correct/10000 total images
Model Size (kB): 160.4
PSEUDO-QUANTIZED ClassyClassifier:
Accuracy: 0.689, 6890 correct/10000 total images
```