

▼ Lab 3: Quantization

Team Jobless

Jiyang Tang (jiyangta), Ran Ju (ranj), Tinglong Zhu (tinglongz), Xinyu Lu (xinyulu2)

▼ Exercise 0: Simple Manual Quantization (Ran Ju and Xinyu Lu)

Here, you are provided a 4x3 tensor of float32 values.

```
1 import torch

1 small_tensor = torch.Tensor([[ 0.22,  0.52,  0.13],
2                               [ 0.29,  1.00,  0.73],
3                               [ 0.44,  0.00,  0.33],
4                               [ 0.59,  0.97,  0.78]])
```

▼ A. Affine Quantization

First, quantize the tensor into two-bit integer precision, with a quantized range from 0 to 3, inclusive. Provide the values of the result below:

You should be able to do this step by hand!

```
1 answer_1a = torch.Tensor([[1, 2, 0],
2                             [1, 3, 2],
3                             [1, 0, 1],
4                             [2, 3, 2]])
```

Hint: note the min and max values of the provided tensor, and try calculating the scaling factor and zero point rather than relying solely on your intuitions

▼ B. Dequantization

Now, dequantize your quantized values back into the original float32 space of (0.0, 1.0). Provide the values of the result below:

You should be able to do this step by hand!

```

1 answer_1b = torch.Tensor([[0.33, 0.67, 0.0],
2     [0.33, 1.0, 0.67],
3     [0.33, 0.0, 0.33],
4     [0.67, 1.0, 0.67]])

```

▼ C. Discussion

Discuss any interesting observations you made. Was anything unintuitive? Surprising?

Although quantization helps in compressing data, it is evident that some precision is lost in the process. Our original values do not perfectly match the dequantized tensor. When quantizing, values are rounded to the nearest integer which introduce errors and it can be carried over when dequantizing.

▼ D. Quantization Error

Calculate and report quantization error as a 4x3 tensor, as simply the absolute difference between the dequantized values and the original.

```

1 answer_1d = torch.Tensor([[0.11, 0.15, 0.13],
2     [0.04, 0.00, 0.06],
3     [0.11, 0.00, 0.00],
4     [0.08, 0.03, 0.11]])
5

```

▼ Exercise 1: Implement Quantization (Ran Ju and Xinyu Lu)

Now, we will implement quantization of PyTorch Tensors.

```
1 ! pip3 install torch torchvision torchaudio --index-url https://download.pyt
```

```

Looking in indexes: https://download.pytorch.org/whl/cu118
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.1
Requirement already satisfied: torchvision in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: torchaudio in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-packa
- . . . . .

```

Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from

1. Calculate Scale and Zero Point for a single Tensor.

In this section, implement affine quantization as described in Lecture 6 where we will be quantizing from `torch.Tensor` in float32 to 8-bit uint8 `torch.Tensors`.

Linear mapping: $Q(r) = \text{round}(\frac{r}{S} + Z)$ **inverse:** $\tilde{r} = (Q(r) - Z) \cdot S$

r is the input, ***S*** is the scaling factor and ***Z*** is the zero point.

S is ratio of input range $[\alpha, \beta]$ to output range $[\alpha_q, \beta_q]$: $S = \frac{\beta - \alpha}{\beta_q - \alpha_q}$

Z is bias mapping zero between input and output: $Z = -(\frac{\alpha}{S} - \alpha_q)$

First implement a function which calculates the float scale and int zeropoint values.

```
1 import torch
2 from typing import Tuple
3
4 def _calculate_scale_and_zeropoint(
5     min_val: float, max_val: float, num_bits: int) -> Tuple[float, int]:
6     qmin = 0
7     qmax = 2**num_bits - 1
8
9     # min_val_scalar = min_val.item()
10    # max_val_scalar = max_val.item()
11
12    scale = (max_val - min_val) / float(qmax - qmin)
13    zero_point = int(qmin - (min_val / scale))
14
15    return scale, zero_point
16
```

2. Quantization and Dequantization

2. Quantization and Dequantization

1. Implement quantization to convert a `torch.float32` Tensor to unsigned `torch.uint8`.
2. Implement dequantization to convert the `torch.uint8` tensor to `torch.float32`.

Question

A. Given, a number of quantization bits q , minimum weight value a , and maximum weight value b ; what is the largest possible quantization for a single weight during quantization? Assume there is no numeric overflow.

The largest possible quantization is $2^q - 1$.

```
1 def quantize(x: torch.Tensor, scale: float, zero_point: int, dtype=torch.uint8):
2     # raise NotImplementedError()
3     inv_scale = 1.0 / scale
4     return torch.clamp(torch.round(x * inv_scale) + zero_point,
5                             0, 255).to(dtype)
6
7 def dequantize(x: torch.Tensor, scale: float, zero_point: int):
8     # raise NotImplementedError()
9     return (x.to(torch.float32) - zero_point) * scale
10
```

3. Validate the Implementation

1. Now run your quantization implementation, the dequantized values from your implementation should exactly match the test cases.
2. Empirically, what is the average quantization error? maximum quantization error?
3. Save the original fp32 tensor and quantized tensor to disk with `torch.save`. Report the difference in disk utilization, does this meet expectations?

Hint: If you observe a small difference in the dequantized values, consider what happens to values when rounding or the effects of limited dynamic range of the `uint8` datatype.

```
1 from copy import deepcopy
2
3 def test_case_0():
4     torch.manual_seed(999)
5     test_input = torch.randn((4,4))
6
7     min_val, max_val = torch.min(test_input), torch.max(test_input)
```

```

8  scale, zero_point = _calculate_scale_and_zeropoint(min_val, max_val, 8)
9
10 your_quant = quantize(test_input, scale, zero_point)
11 your_dequant = dequantize(your_quant, scale, zero_point)
12
13 test_case_0 = torch.Tensor([
14     [-0.2623,  1.3991,  0.2842,  1.0275],
15     [-0.9838, -3.4104,  1.4866,  0.2405],
16     [ 1.4866, -0.3716,  0.0874,  2.1424],
17     [ 0.6340, -1.1587, -0.7870,  0.0656]])
18
19 assert torch.allclose(your_dequant, test_case_0, atol=1e-4)
20 assert torch.allclose(your_dequant, test_input, atol=5e-2)
21
22
23 ### Test Case 1
24 def test_case_1():
25     torch.manual_seed(999)
26     test_input = torch.randn((8,8))
27
28     min_val, max_val = torch.min(test_input), torch.max(test_input)
29     scale, zero_point = _calculate_scale_and_zeropoint(min_val, max_val, 8)
30
31     your_quant = quantize(test_input, scale, zero_point)
32     your_dequant = dequantize(your_quant, scale, zero_point)
33
34     test_case_1 = torch.Tensor(
35         [[-0.2623,  1.3991,  0.2842,  1.0275, -0.9838, -3.4104,  1.4866,  0.2405],
36          [ 1.4866, -0.3716,  0.0874,  2.1424,  0.6340, -1.1587, -0.7870,  0.0656],
37          [ 0.0000, -0.6558, -1.0056,  0.3061,  0.6340, -1.0931, -1.6178,  1.5744],
38          [-1.7927,  0.6121, -0.7214,  0.6121,  0.3279, -1.5959, -0.5247,  0.3498],
39          [-1.3773,  1.1149, -0.7870,  0.2842,  0.9182, -1.1805, -0.7433, -1.5524],
40          [ 1.0056, -0.1093,  1.3991, -0.9182, -1.1805, -0.6777, -0.3061,  0.9838],
41          [ 0.2186,  1.6396,  1.0712,  1.7489,  0.0874,  0.3498,  0.9838,  1.2024],
42          [-0.3935, -0.6340,  1.9238,  1.2898,  0.0219,  0.3935,  1.4866, -0.9405]])
43
44     assert torch.allclose(your_dequant, test_case_1, atol=1e-4)
45     assert torch.allclose(your_dequant, test_input, atol=5e-2)
46
47 test_case_0()
48 test_case_1()
49

```

```

1 import torch
2 from typing import Tuple
3 import os

```

```

3 import os
4
5 def _calculate_scale_and_zeropoint(min_val: float, max_val: float, num_bits:
6     qmin = 0
7     qmax = 2**num_bits - 1
8
9     scale = (max_val - min_val) / float(qmax - qmin)
10    zero_point = int(qmin - (min_val / scale))
11
12    return scale, zero_point
13
14 def quantize(x: torch.Tensor, scale: float, zero_point: int, dtype=torch.uint8):
15     inv_scale = 1.0 / scale
16     return torch.clamp(torch.round(x * inv_scale) + zero_point, 0, 255).to(dtype)
17
18 def dequantize(x: torch.Tensor, scale: float, zero_point: int):
19     return (x.to(torch.float32) - zero_point) * scale
20
21 def calculate_quantization_error(original: torch.Tensor, quantized: torch.Tensor):
22     return torch.abs(original - quantized)
23
24 def test_case_0():
25     torch.manual_seed(999)
26     test_input = torch.randn((4, 4))
27
28     min_val, max_val = torch.min(test_input), torch.max(test_input)
29     scale, zero_point = _calculate_scale_and_zeropoint(min_val, max_val, 8)
30
31     your_quant = quantize(test_input, scale, zero_point)
32     your_dequant = dequantize(your_quant, scale, zero_point)
33
34     test_case_0 = torch.Tensor([
35         [-0.2623,  1.3991,  0.2842,  1.0275],
36         [-0.9838, -3.4104,  1.4866,  0.2405],
37         [ 1.4866, -0.3716,  0.0874,  2.1424],
38         [ 0.6340, -1.1587, -0.7870,  0.0656]])
39
40     assert torch.allclose(your_dequant, test_case_0, atol=1e-4)
41     assert torch.allclose(your_dequant, test_input, atol=5e-2)
42
43     # Calculate and report quantization error
44     quantization_error = calculate_quantization_error(test_input, your_dequant)
45     avg_quantization_error = quantization_error.mean().item()
46     max_quantization_error = quantization_error.max().item()
47
48     print(f"Average Quantization Error for Test Case 0: {avg_quantization_error}")

```

```

49     print(f"Maximum Quantization Error for Test Case 0: {max_quantization_er
50
51     # Save original and quantized tensors to disk
52     torch.save(test_input, "original_tensor0.pt")
53     torch.save(your_quant, "quantized_tensor0.pt")
54
55 def test_case_1():
56     torch.manual_seed(999)
57     test_input = torch.randn((8, 8))
58
59     min_val, max_val = torch.min(test_input), torch.max(test_input)
60     scale, zero_point = _calculate_scale_and_zerpoint(min_val, max_val, 8)
61
62     your_quant = quantize(test_input, scale, zero_point)
63     your_dequant = dequantize(your_quant, scale, zero_point)
64
65     test_case_1 = torch.Tensor(
66         [[-0.2623, 1.3991, 0.2842, 1.0275, -0.9838, -3.4104, 1.4866, 0.
67         [ 1.4866, -0.3716, 0.0874, 2.1424, 0.6340, -1.1587, -0.7870, 0.6
68         [ 0.0000, -0.6558, -1.0056, 0.3061, 0.6340, -1.0931, -1.6178, 1.5
69         [-1.7927, 0.6121, -0.7214, 0.6121, 0.3279, -1.5959, -0.5247, 0.3
70         [-1.3773, 1.1149, -0.7870, 0.2842, 0.9182, -1.1805, -0.7433, -1.5
71         [ 1.0056, -0.1093, 1.3991, -0.9182, -1.1805, -0.6777, -0.3061, 0.9
72         [ 0.2186, 1.6396, 1.0712, 1.7489, 0.0874, 0.3498, 0.9838, 1.2
73         [-0.3935, -0.6340, 1.9238, 1.2898, 0.0219, 0.3935, 1.4866, -0.9
74
75     assert torch.allclose(your_dequant, test_case_1, atol=1e-4)
76     assert torch.allclose(your_dequant, test_input, atol=5e-2)
77
78     # Calculate and report quantization error
79     quantization_error = calculate_quantization_error(test_input, your_dequant)
80     avg_quantization_error = quantization_error.mean().item()
81     max_quantization_error = quantization_error.max().item()
82
83     print(f"Average Quantization Error for Test Case 1: {avg_quantization_er
84     print(f"Maximum Quantization Error for Test Case 1: {max_quantization_er
85
86     # Save original and quantized tensors to disk
87     torch.save(test_input, "original_tensor1.pt")
88     torch.save(your_quant, "quantized_tensor1.pt")
89
90 # Run the test cases
91 test_case_0()
92 test_case_1()
93
94 # Report disk utilization difference

```

```

95 original_size_0 = os.path.getsize("original_tensor0.pt")
96 quantized_size_0 = os.path.getsize("quantized_tensor0.pt")
97 original_size_1 = os.path.getsize("original_tensor1.pt")
98 quantized_size_1 = os.path.getsize("quantized_tensor1.pt")
99
100 print(f"Difference in disk utilization for Test Case 0: {original_size_0 - c
101 print(f"Difference in disk utilization for Test Case 1: {original_size_1 - c
102

```

```
























Average Quantization Error for Test Case 0: 0.0063123637810349464
Maximum Quantization Error for Test Case 0: 0.011491775512695312
Average Quantization Error for Test Case 1: 0.0058608632534742355
Maximum Quantization Error for Test Case 1: 0.011491775512695312
Difference in disk utilization for Test Case 0: 59 bytes
Difference in disk utilization for Test Case 1: 187 bytes

















```

The difference in disk utilization is as expected. Since there are 16 entries for test case 0, and 64 entries in test 1, and the difference in disk utilization for test case 1 is approximately 4 times large as that of test case 0.

Exercise 2: MNIST and SST (Tinglong Zhu)

Details of the computing platform used in this section is listed below.

| | |
|---|--|
|  Operating System Properties | |
|  OS Name | Microsoft Windows 11 Pro |
|  OS Language | English (United States) |
|  OS Installer Language | Chinese (Simplified, China) |
|  OS Kernel Type | Multiprocessor Free (64-bit) |
|  OS Version | 10.0.22621.2506 |
|  OS Service Pack | - |
|  OS Installation Date | 6/22/2022 |
|  OS Root | C:\Windows |
|  Physical Memory | |
|  Total | 65157 MB |
|  Used | 23394 MB |
|  Free | 41763 MB |
|  Utilization | 36 % |
|  CPU Properties | |
|  CPU Type | 8C+8c , 3900 MHz (39 x 100) |
|  CPU Alias | Alder Lake-HX |
|  CPU Stepping | C0 |
|  Instruction Set | x86, x86-64, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA, AES, SHA |
|  Min / Max CPU Multiplier | 8x / 28x |
|  Engineering Sample | No |
|  L1 Code Cache | 32 KB per core |
|  L1 Data Cache | 48 KB per core |

| | |
|--|--|
|  L2 Cache | 1280 KB per core (On-Die, ECC, Full-Speed) |
|  L3 Cache | 30 MB (On-Die, ECC, Full-Speed) |
|  CPUID Properties | |
|  CPUID Manufacturer | GenuineIntel |
|  CPUID CPU Name | 12th Gen Intel(R) Core(TM) i9-12950HX |
|  CPUID Revision | 00090672h |
|  IA Brand ID | 00h (Unknown) |
|  Platform ID | 58h / MC 02h (LGA1700) |
|  Microcode Update Revision | 1Eh |
|  SMT / CMP Units | 2 / 16 |
|  Tjmax Temperature | 100 癈 (212 癈) |
|  CPU Thermal Design Power (TDP) | 55 W |
|  CPU Max Power Limit | Unlimited Power / Unlimited Time |
|  CPU Power Limit 1 (Long Duration) | 175 W / 56.00 sec (Unlocked) |
|  CPU Power Limit 2 (Short Duration) | 175 W / 2.44 ms (Unlocked) |
|  Max Turbo Boost Multipliers | 1C: 53x, 2C: 53x, 3C: 53x, 4C: 53x, 5C: 53x, 6C: 53x, 7C: 53x, ... |

We use PyTorch 1.13 and Python 3.9.18 for experiments in this section.

For torch qconfig mapping, we set it to “x86” since we are using an Intel CPU.

The training hyperparameters are listed below (same as provided in the instructions):

| Hyperparameter | Value |
|-------------------------|-------|
| Learning Rate | 1e-3 |
| Batch size | 64 |
| Hidden size | 256 |
| Number of hidden layers | 2 |
| Training epochs | 2 |
| Optimizer | Adam |
| Input Size | 400 |

And the following table shows our experiment Results

| Task | Q | dtype | Size (MB) | (Params) | Accuracy(%) | Latency B1(ms) | B64(ms) | B1 (std) | B64 (std) |
|-----------|----|---------|-----------|----------|-------------|----------------|---------|----------|-----------|
| MNIST/SST | -- | float32 | 5.883967 | 1470474 | 97.34 | 0.323 | 1.30 | 0.00550 | 0.0380 |
| MNIST/SST | D | float16 | 5.885847 | 1470474 | 97.34 | 0.512 | 1.72 | 0.00510 | 0.0549 |
| MNIST/SST | D | qint8 | 1.480727 | 1470474 | 97.34 | 0.503 | 1.11 | 0.00369 | 0.0256 |
| MNIST/SST | S | qint8 | 1.515239 | 1470474 | 96.67 | 0.593 | 0.955 | 0.00971 | 0.0385 |

Observation 1:

It seems that dynamically quantizing the model to float16 did not reduce the model size. After doing some research. <https://discuss.pytorch.org/t/float16-dynamic-quantization-has-no-model-size-benefit/99675> pointed out that the parameters are quantized to float16 but still stored in a float32 format. Thus, considering some other parameters for quantization (scaling factor, etc.)

the model size would be a little larger than the original.

Observation 2:

Whether quantized to float16 or qint8 under a dynamic quantization scheme, there is no accuracy loss for this task. If we quantize the model in a static way, there is little accuracy degradation. We think that it might be due to that the task is simple and does not need a large model space. Thus quantizing may not lead to a accuracy loss, since high precision is not necessary. The dynamic quantization has a slightly better accuracy compared to the static PTQ model. This met our expectation since the scaling factors of dynamic PTQ model are computed dynamically for each input and layer, while the statically quantized model's clipping range is pre-calibrated and fixed using a validation set. And the statically quantized model's B64 inference latency is lower than the dynamic PTQ model, since it does not have computational overheads (converting floating-> int). The B1 inference latency of the static PTQ model is much higher than that of the dynamic PTQ model. We consider it was affected by the underlying implementation of PyTorch.

Exercise 3: Quantize Your Class Project (Jiyang Tang)

A. Reporting preliminaries

Hardware Specs

- Laptop: Dell G15 5511
- CPU
 - 11th Gen Intel i7-11800H, 2.30GHz
 - Speed 3.35 GHz
 - 8 cores, 16 logical processors
 - L1 cache: 640 KB
 - L2 cache: 10.0 MB
 - L3 cache: 24.0 MB
- Memory: 32GB, dual-channel DDR4
- OS: Windows 10 Education 22H2, OS Build 19045.3570
- Python environment:
 - Python==3.11
 - PyTorch==2.1
 - ESPnet==202210

- Test data is loaded into memory beforehand

NOTE: our python environment is updated and this causes the results from Lab2 to be outdated. We updated all baseline results in this lab report to reflect this change.

Model and Data

We use the same model architecture (multi-decoder encoder-decoder architecture) and data (subsampling MuST-C) as in Lab2. Please refer to our Lab2 report for more details.

Benchmarking

We measure the latency in seconds and BLEU scores for each experiment and visualize the results. We chose not to include FLOP as it is not affected much by quantization. For latency, we measure the time it takes to perform inference on every test sample and take the average of them except the first one (warm-up). We increase the number of test utterances from 20 to 100 in order to obtain more stable measurement.

All experiments are conducted using the specified hardware above under the same conditions. We stopped as many applications and background services as possible before benchmarking. All experiments are conducted using ESPnet.

Experiment Configurations

We selected experiments that vary input size by increasing MFCC/FBank hop length from Lab2, as they are the only configurations that successfully reduced the inference latency. The configurations are listed below:

| Model | Hop Length |
|-------------|------------|
| Original | 160 |
| input_size1 | 200 |
| input_size2 | 250 |
| input_size3 | 300 |

B. Discussing quantization with respect to your project

We believe there are two types of quantization that are suitable for our project:

- Dynamic PTQ with 8-bit (integer) precision
- Static PTQ with 8-bit (integer) precision

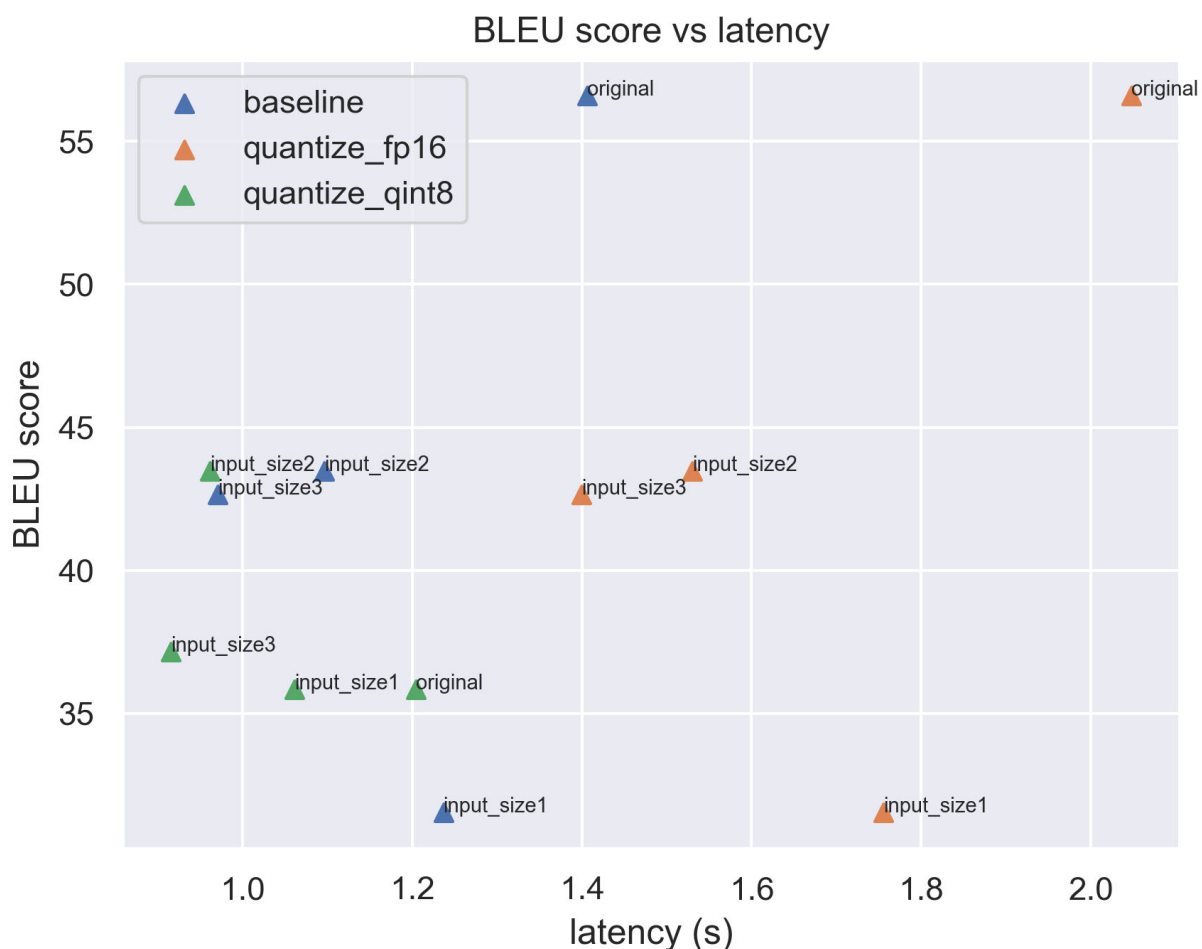
QAT is not realistic for our project because the amount of resources needed to train this model is too much in the current timeline. PTQ is shown to be quite effective for many speech tasks, and we believe with proper implementation and tuning, we can achieve big reduction to model

size and inference latency. We don't plan to use other precision types in our final product, such as 16-bit float, because the X86 PyTorch backend doesn't really provide native support for such data types, and our experiments in later sections show that these types actually increase the latency.

For the first method, it's quite simple to implement. After loading the pre-trained model from file, we use `torch.quantization.quantize_dynamic` API to quantize the model. Then we use the quantized modules as if they're regular pytorch modules.

Unfortunately, we failed to implement the second method in this lab. During beam search, we need to quantize and dequantize the input and output of the MT decoder for static PTQ to work. This requires us to change ESPnet's internal code, and we simply didn't have enough time. However, this method is still included in our project plan and we'll present more results in future report(s).

C. Quantizing your model, on your device



| Configuration | Q | dtype | Size (MB) | Params | BLEU | Latency(s) |
|----------------|---|-------|-----------|--------|------|------------|
| baseline | | | 100 | 100 | 56 | 1.4 |
| quantize_fp16 | | | 100 | 100 | 56 | 2.05 |
| quantize_qint8 | | | 100 | 100 | 56 | 1.25 |

| | | | | | | |
|-------------|---|---------|-----|------------|------|------|
| Original | - | float32 | 220 | 57,592,032 | 56.6 | 1.41 |
| input_size1 | - | float32 | 220 | 57,592,032 | 31.5 | 1.24 |
| input_size2 | - | float32 | 220 | 57,592,032 | 43.5 | 1.10 |
| input_size3 | - | float32 | 220 | 57,592,032 | 42.7 | 0.97 |
| Original | D | qint8 | 70 | 57,592,032 | 35.8 | 1.20 |
| input_size1 | D | qint8 | 70 | 57,592,032 | 35.8 | 1.06 |
| input_size2 | D | qint8 | 70 | 57,592,032 | 43.5 | 0.96 |
| input_size3 | D | qint8 | 70 | 57,592,032 | 37.2 | 0.92 |
| Original | D | float16 | 220 | 57,592,032 | 56.6 | 2.05 |
| input_size1 | D | float16 | 220 | 57,592,032 | 31.5 | 1.76 |
| input_size2 | D | float16 | 220 | 57,592,032 | 43.5 | 1.53 |
| input_size3 | D | float16 | 220 | 57,592,032 | 42.6 | 1.40 |

We performed three sets of experiments:

- Baselines without quantization
- Dynamic PTQ with 8-bit (integer) precision
- Dynamic PTQ with 16-bit (float) precision

Above shows our results with each set of experiments marked with different colors:

There are several interesting conclusions (with and without PTQ, and using different precisions) we can draw from this plot:

Like in Lab2, decreasing the input size reduces the inference latency regardless of whether quantization is enabled.

Dynamic PTQ using 8-bit integer significantly reduces the inference latency. However, it also reduces the BLEU score, especially when the input size isn't reduced (comparing `original` in blue and green). Moreover, the difference in latency is less significant when the input size becomes smaller (for example, comparing `original` and `input_size3` in blue and green). Strangely, `input_size2` with 8-bit integer PTQ achieves higher BLEU score than other reduced input size experiments even without PTQ. We believe somehow this configuration introduces the least precision loss, showing that calibration is very important for PTQ. Overall, this level of decrease in BLEU score is acceptable for our project. But we will try to improve this using static PTQ in the future.

Another interesting finding is that dynamic PTQ using 16-bit float drastically increase the inference latency. Meanwhile, the BLEU scores stay the same as the baselines. The reason is that X86 PyTorch backend doesn't natively support FP16, therefore needs to emulate 16-bit precision using FP32. Compared to the model used in Exercise 2, speech translation models have more diverse layers and that requires more value conversions, causing the latency to increase without the benefit of reduced model size. Since our target platform is X86 CPUs, we

will not use this type of quantization in the final product.

Follow-up question and future work

The follow-up question to our results is that, does static quantization work introduce more latency reduction? Exercise 2 uses a feed-forward network which contains mostly linear layers. But the speech translation model consists of many other modules, like attention, convolution, and various activation functions. Dynamic quantization cannot quantize these modules, and we believe static quantization can achieve lower latency. And we aim to perform the following experiments to verify this hypothesis in the future:

| Configuration | Q | dtype |
|---------------|---|-------|
| Original | S | qint8 |
| input_size1 | S | qint8 |
| input_size2 | S | qint8 |
| input_size3 | S | qint8 |