HW9.

Mengying Lin SID: 3038737132.

1. Meta-learning for Learning 1D functions

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

(a) In the original MAML algorithm, the inner loop performs gradient descent to optimize loss with respect to a task distribution. However, here we're going to use the closed form min-norm solution for regression instead of gradient descent.

Let's recall the closed form solution to the min-norm problem. Write the solution to

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{\beta}\| \text{ , such that } \mathbf{y} = A\boldsymbol{\beta}$$

in terms of A and $\mathbf{y}$.

$\beta = A^T(AA^T)^{-1}y.$

in terms of A and $\mathbf{y}$.

(b) For simplicity, suppose that we have exactly one training point (x, y), and one true feature $\phi_t^u(x) = \phi_1^u(x)$. We have a second (alias) feature that is identical to the first true feature, $\phi_a^u(x) = \phi_2^u(x) = \phi_1^u(x)$. This is a caricature of what always happens when we have fewer training points than model parameters.

The function we wish to learn is $y = \phi_t^u(x)$. We learn coefficients $\hat{\boldsymbol{\beta}}$ using the training data. Note, both the coefficients and the feature weights are 2-d vectors.

Show that in this case, the solution to the min-norm problem 1 is $\hat{\boldsymbol{\beta}} = \frac{1}{c_0^2 + c_1^2} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$

let $A = \begin{pmatrix} c_0 \phi_a^u(x) \\ c_1 \phi_a^u(x) \end{pmatrix}^T$

$$\beta_{min} = A^T(AA^T)^{-1}y$$

$$= \begin{pmatrix} c_0\phi_a^u(x) \\ c_1\phi_a^u(x) \end{pmatrix} \left(c_0^2\phi_a^{u2}(x) + c_1^2\phi_a^{u2}(x)\right)^{-1} \begin{pmatrix} \phi_a^u(x) \\ \phi_a^u(x) \end{pmatrix}$$

$$= \frac{1}{c_0^2 + c_1^2} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

(c) Assume for simplicity that we have access to infinite data from the test distribution for the purpose of updating the feature weights $\mathbf{c}$. **Calculate the gradient of the expected test error with respect to the feature weights c_0 and c_1, respectively:**

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{c}} \left(\mathbb{E}_{x_{test}, y_{test}} \left[\frac{1}{2} \left\| y - \hat{\beta}_0 c_0 \phi_t^u(x) - \hat{\beta}_1 c_1 \phi_a^u(x) \right\|_2^2 \right] \right).$$

Use the values for β from the previous part. *(Hint: the features $\phi_i^u(x)$ are orthonormal under the test distribution. They are not identical here.)*

$$E_{x_{test}, y_{test}} \left[\frac{1}{2} \| y - \hat{\beta}_0 c_0 \phi_t^u(x) - \hat{\beta}_1 c_1 \phi_a^u(x) \|_2^2 \right]$$

$$= \frac{1}{2} E_{x_{test}, y_{test}} \left[\| \phi_t^u(x) - \frac{c_0^2}{c_0^2 + c_1^2} \phi_t^u(x) \|_2^2 + \| \frac{c_1^2}{c_0^2 + c_1^2} \phi_a^u(x) \|_2^2 \right]$$

$$= \frac{1}{2} \left[\left(1 - \frac{c_0^2}{c_0^2 + c_1^2}\right)^2 + \left(\frac{c_1^2}{c_0^2 + c_1^2}\right)^2 \right]$$

$$= \frac{c_1^4}{(c_0^2 + c_1^2)^2}$$

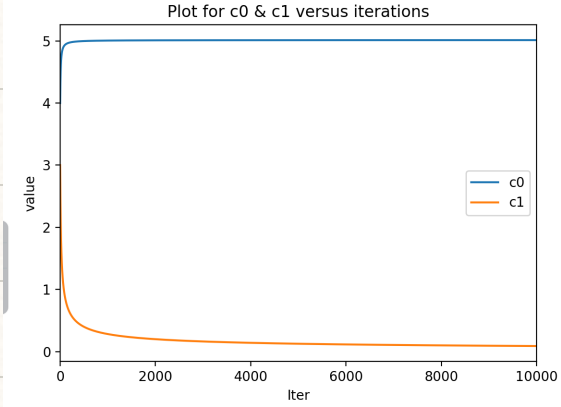
$$\frac{dE}{dc_0} = -\frac{4c_1^4 c_0}{(c_0^2 + c_1^2)^3}$$

$$\frac{dE}{dc_1} = \frac{4c_1^3(c_0^2 + c_1^2) - 4c_1^5}{(c_0^2 + c_1^2)^3}$$

$$= \frac{4c_1^3 c_0^2}{(c_0^2 + c_1^2)^3}.$$

(d) **Generate a plot** showing that, for some initialization $\mathbf{c}^{(0)}$, as the number of iterations $i \to \infty$ the weights empirically converge to $c_0 = \|\mathbf{c}^{(0)}\|$, $c_1 = 0$ using gradient descent with a sufficiently small step size. Include the initialization and its norm and the final weights. What will β go to?

Run the code in the Jupyter Notebook and then answer these questions:



Init: $c_0 = 4$, $c_1 = 3$. $\|c^{(0)}\| = 5$.

Final: $c_0 = 5.0126$ $c_1 = 0.0889$

(e) (In MAML for regression using closed-form solutions) Considering the plot of regression test loss versus `n_train_post`, **how does the performance of the meta-learned feature weights compare to the case where all feature weights are set to 1?** Additionally, **how does their performance compare to the oracle**, which performs regression using only the features present in the data? Can you **explain the reason for the downward spike observed at `n_train_post = 32`**?

The former performs way more better.

Oracle performs better than the two cuz it has the knowledge of the true features.

(f) By examining the changes in feature weights over time during meta-learning, **can you justify the observed improvement in performance?** Specifically, can you **explain why certain feature weights are driven towards zero**?

At first all of the weights are initialized to 0. Afterwards most of them are driven to 0 while the others are gradually driven to large values.

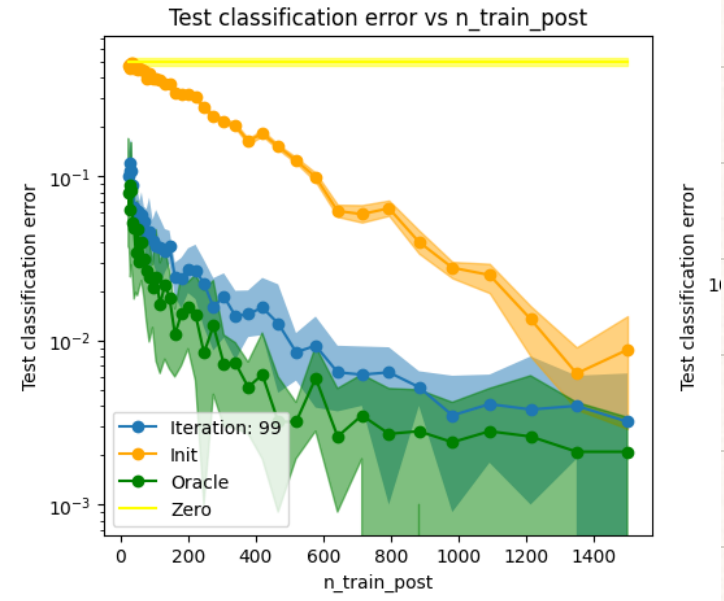
For those driven to 0, it could be because they are less significant.

(g) (In MAML for regression using gradient descent) **With `num_gd_steps = 5`, does meta-learning contribute to improved performance during test time? Furthermore, if we change `num_gd_steps` to 1, does meta-learning continue to function effectively?**

Yes. No, its performance degraded because it couldn't be finely adapted to the data within just 1 step.

(h) (In MAML for classification) Based on the plot of classification error versus `n_train_post`, **how does the performance of the meta-learned feature weights compare to the case where all feature weights are 1? How does the performance of the meta-learned feature weights compare to the oracle** (which performs logistic regression using only the features present in the data)?

Iteration: 100



The same answer in the regression part.

(i) By observing the evolution of the feature weights over time as we perform meta-learning, **can you justify the improvement in performance? Specifically, can you explain why some feature weights are being driven towards zero?**

The same answer in the regression part.

2. Vision Transformer

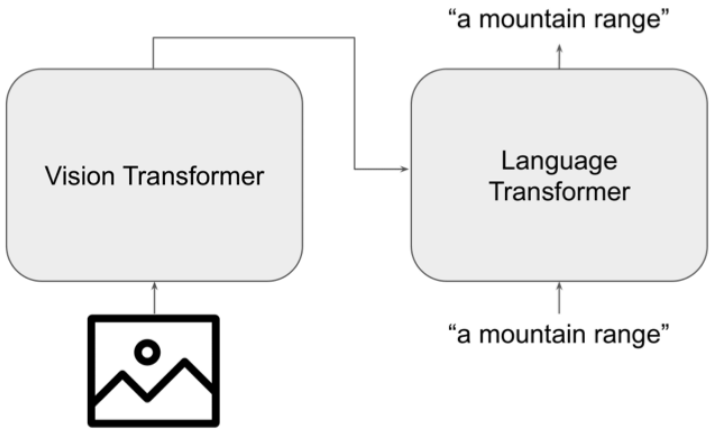


Figure 3: Image captioning model

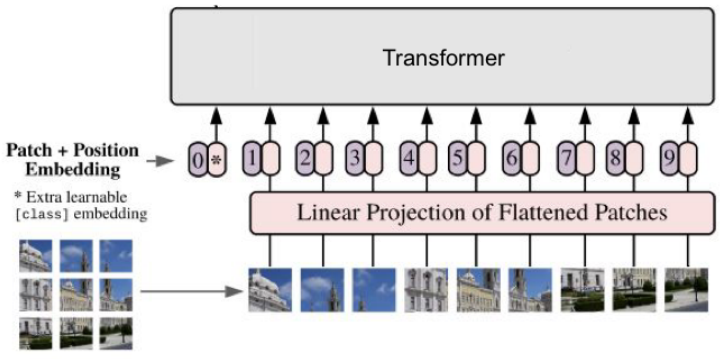


Figure 4: Vision Transformer

(a) For each transformer, state whether it is more appropriate to use a transformer encoder (a transformer with no masking except to handle padding) or decoder (a transformer with autoregressive self-attention masking) and why.

Vision transformer?

- ○ Encoder-style transformer
- ⊘ Decoder-style transformer

Reason: It feeds in already encoded vectors.

Language transformer?

- ⊘ Encoder-style transformer
- ○ Decoder-style transformer

Reason: Need to encode the texts into vectors.

(b) A standard language transformer for captioning problems alternates between layers with cross-attention between visual and language features and layers with self-attention among language features. Let's say we modify the language transformer to have a single layer which performs both attention operations at once. The grid below shows the attention mask for this operation. (For now, assume the vision transformer only outputs 3 image tokens called <ENC1>, <ENC2>, and <ENC3>. <SOS> is the start token, and <PAD> is a padding token.)

(i) One axis on this grid represents sequence embeddings used to make the queries, and the other axis represents sequence embeddings used to make the keys. **Which is which?**

- ◯ Each column creates a query, each row creates a key and a value
- ◯ Each column creates a key and a value, each row creates a query
- ◯ Each column creates a query and a value, each row creates a key
- ◯ Each column creates a key, each row creates a query and a value

(ii) **Mark X in some of the blank cells in the grid to illustrate the attention masks.** (A X marked cell is masked out, a blank cell is not.)

	<SOS>	a	mountain	range	<PAD>
<SOS>		X	X	X	X
a			X	X	X
mountain				X	X
range					X
<PAD>					
<ENC1>					
<ENC2>					
<ENC3>					

(c) In discussion, we showed that the runtime complexity of vision transformer attention is $O(D(H^4/P^4))$, where H is the image height and width, P is the patch size, and D is the feature dimension of the queries, keys, and values. Some recent papers have reduced the complexity of vision transformer attention by segmenting an image into windows, as shown in Figure 5.

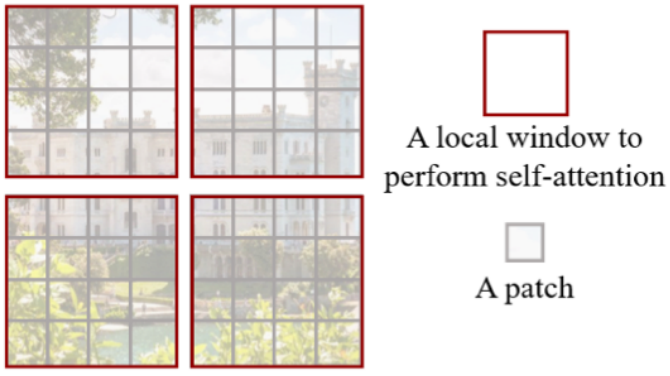


Figure 5: Vision transformer attention with windows

Patches only attend to other patches within the same window. **What is the Big-O runtime complexity of the attention operation after this modification?** Assume each window consists of K by K patches.

There are K^2 patches in a window.

$(H/kp)^2$ windows in total.

Computing attention between each pair of patches: $O(p^2)$

The time complexity is $O(k^4 \cdot H^2/k^2p^2 \cdot p^2) = O(H^2k^2)$.

3. Pretraining and Finetuning

When we use a pretrained model without fine-tuning, we typically just train a new task-specific head. With standard fine-tuning, we also allow the model weights to be adapted.

However, it has recently been found that we can selectively fine-tune a subset of layers to get better performance especially under certain kinds of distribution shifts on the inputs. Suppose that we have a ResNet-26 model pretrained with CIFAR-10. Our target task is CIFAR-10-C, which adds pixel-level corruptions (like adding noise, different kinds of blurring, pixelation, changing brightness and contrast, etc) to CIFAR-10. If we could only afford to fine-tune one layer, **which layer (i.e. 1,2,3,4,5) in Figure 6 should we choose to finetune to get the best performance on CIFAR-10-C? Give brief intuition as to why.**

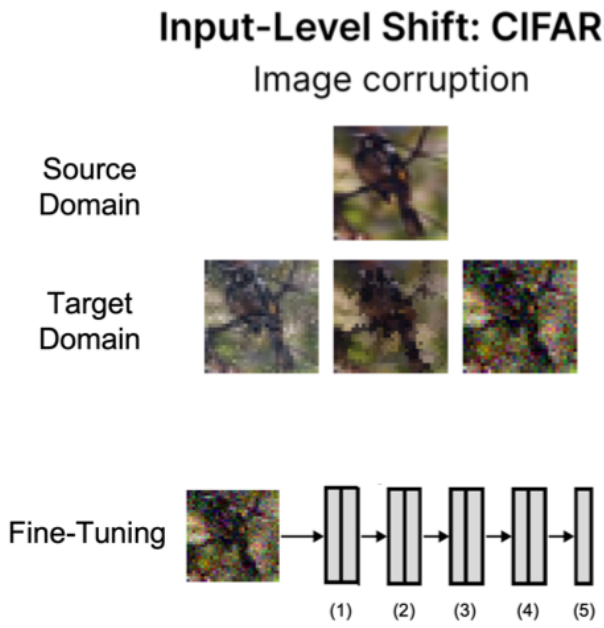


Figure 6: Fine-tuning the model pretrained with CIFAR-10 on CIFAR-10-C dataset

The 5th layer, because it is considered that the lower layers usually extract common features while the deep layers will be responsible for extracting more complex and high-level features which are more targeted at task.

4. Prompting Language Models

(a) **Exploring Pretrained LMs**

Play around with the web interface at https://dashboard.cohere.ai/playground/generate. This plaground provides you an interface to interact with a large language model from Cohere and tweak various parameters. You will need to sign up for a free account.

Once you're logged in, you can choose a model in the parameters pane on the right. "command-xlarge-nightly" is a generative model that responds well with instruction-like prompts. "xlarge" and "medium" are generative models focusing on sentence completion. Spend a while exploring prompting these models for different tasks. Here are some suggestions:

- Look through the 'Examples ...' button at the top of the page for example prompts.
- Ask the model to answer factual questions.
- Prompt the model to generate a list of 100 numbers sampled uniformly between 0 and 9. Are the numbers actually randomly distributed?
- Insert a poorly written sentence, and have the model correct the errors.
- Have the model brainstorm creative ideas (names for a storybook character, recipes, solutions to solve a problem, etc.)
- Chat with the model like a chatbot.

Answer the questions below:

i. **Describe one new thing you learned by playing with these models.**

ii. **How does the temperature parameter affect the outputs?** Justify your answer with a few examples.

iii. **Describe a task where the larger models (e.g., "xlarge" or "command-xlarge-nightly") significantly outperform the smaller ones (e.g., "medium").** Paste in examples from the biggest and smallest model to show this.

iv. **Describe a task where even the largest model performs badly**. Paste in an example to show this.

v. **Describe a task where the model's outputs improve significantly with few-shot prompting compared to zero-shot prompting.**

i. Not every GPT can be called chatGPT !!.

ii. The higher the temperature, the more random the result.

e.g. when input "How to fit in a country with a foreign language."

when temperature set to 0.1, the output is:

How to fit in a new environment with a foreign language

How to fit in a new environment with a foreign language

How to fit in a new environment with a foreign language

When temperature set to 0.4:

How to learn a new language
How to learn a new language in a short time
How to learn a new language in a short time
How to learn a new language in a short time

When temperature set to 1.2:

This week's question was sent in by a member of the Art of Likability community.
Question: How to fit in a new environment with a foreign language
This question came in from Michael, a young guy who is originally from New York City.
In college, he studied in Germany for a semester.
He made some friends, had a great time, and fell in love with a local German girl.
After graduation, he decided to stay in Germany and get a job.
But now he's realizing how much of a struggle it's going to be to make new friends.
Here's what Michael wrote to me:
…

iii. e.g. Prove 0=1.

Command - medium- nightly:

To prove 1=0, you need to find a counterexample. A counterexample is a number that disproves a statement. In this case, you are trying to prove that 1=0, so you need to find a number that is equal to 1 but does not equal 0. One such number is 5.

If you input 5 into the equation 1=0, you get the equation 5=0, which is false. This counterexample disproves the statement 1=0.

Command - xlarge - nightly:

No, it is not possible to prove 1 = 0, because 1 is a positive number and 0 is a negative number, and they are not equal.

iv) Q: Introduce the basic idea of your training process:
A: This is an example description. To edit, click on the pencil icon.
This is an example of a description. To edit, click on the pencil icon.

(v.)

(b) **Using LMs for classification**

Run `lm_prompting.ipynb`, then answer the following questions. If you did not do part (a), you will still need to get a Cohere account to complete this part.

i. Analyze the command-xlarge-nightly model's failures. **What kinds of failures do you see with different prompting strategies?**

ii. **Does providing correct labels in few-shot prompting have a significant impact on accuracy?**

iii. **Observe the model's log probabilities. Does it seem more confident when it is correct than when it is incorrect?**

iv. **Why do you think the GPT2 model performed so much worse than the command-xlarge-nightly model on the question answering task?**

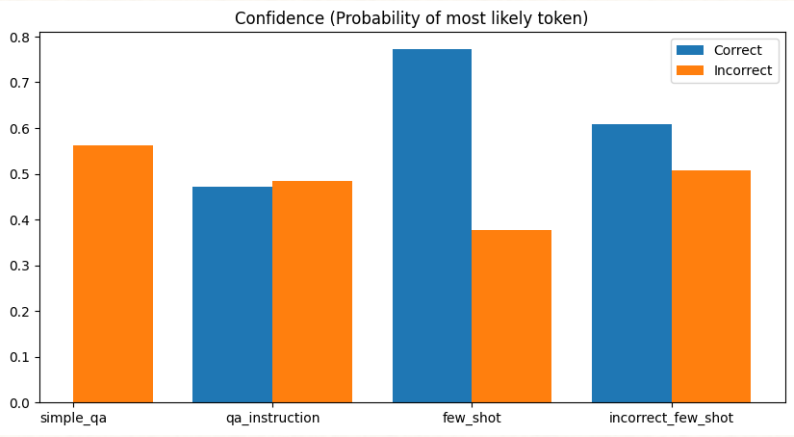
v. **How did soft prompting compare to hard prompting on the pluralize task?**

vi. You should see that when the model fails (especially early in training of a soft prompt or with a bad hard prompt) it often outputs common but uninformative tokens such as the, ", or \n. **Why does this occur?**

i. when strategies is simple/simple-qa/qa-instruction, the failures include invalid answers and WA.

ii. Yes.

iii.



when using few_shot/incorrect_few_shot, the model is more confident when it is correct. Otherwise it's the opposite.

iv. GPT2 is trained to generate texts that are related and coherent to its input.

v. Soft-embedding performs way more better than hard one in pluralizing a word because the former is more effective at capturing semantic relationships of words while the latter only considers word frequencies.

5. Soft-Prompting Language Models

You are using a pretrained language model with prompting to answer math word problems. You are using chain-of-thought reasoning, a technique that induces the model to "show its work" before outputting a final answer.

Here is an example of how this works:

```
[prompt] Question: If you split a dozen apples evenly among yourself
and three friends, how many apples do you get? Answer: There are 12
apples, and the number of people is 3 + 1 = 4. Therefore, 12 / 4 = 3.
Final answer: 3\n
```

If we were doing hard prompting with a frozen language model, we would use a hand-designed [prompt] that is a set of tokens prepended to each question (for instance, the prompt might contain instructions for the task). At test time, you would pass the model the sequence and end after "Answer:" The language model will continue the sequence. You extract answers from the output sequence by parsing any tokens between the phrase "Final answer: " and the newline character "\n".

(a) Let's say you want to improve a frozen GPT model's performance on this task through soft prompting and training the soft prompt using a gradient-based method. This soft prompt consists of 5 vectors prepended to the sequence at the input — these bypass the standard layer of embedding tokens into vectors. (Note: we do not apply a soft prompt at other layers.) Imagine an input training sequence which looks like this:

```
["Tokens" 1-5: soft prompt] [Tokens 6-50: question]
[Tokens 51-70: chain of thought reasoning]
[Token 71: answer] [Token 72: newline]
[Tokens 73-100: padding].
```

We compute the loss by passing this sequence through a transformer model and computing the cross-entropy loss on the output predictions. If we want to train the soft-prompt to output correct reasoning and produce the correct answer, **which output tokens will be used to compute the loss?** (Remember that the target sequence is shifted over by 1 compared to the input sequence. So, for example, the answer token is position 71 in the input and position 70 in the target).

Output tokens 1-5.

(b) Continuing the setup above, **how many parameters are being trained in this model?** You may write this in terms of the max sequence length S, the token embedding dimension E, the vocab size V, the hidden state size H, the number of layers L, and the attention query/key feature dimension D.

Embedding layer: VE.

For decoder:

- Self attention: $3H^2$ (The matrices used for generating keys, values and queries are all $H \times H$).
- Cross attention: $3H^2$.

Final linear layer : HV

The overall parameters are $VE + 6H^2L + HV$.

(c) **Mark each of the following statements as True or False and give a brief explanation.**

(i) If you are using an autoregressive GPT model as described in part (a), it's possible to precompute the representations at each layer for the indices corresponding to prompt tokens (i.e. compute them once for use in all different training points within a batch).

(ii) If you compare the validation-set performance of the *best possible* K-token hard prompt to the *best possible* K-vector soft prompt, the soft-prompt performance will always be equal or better.

(iii) If you are not constrained by computational cost, then fully finetuning the language model is always guaranteed to be a better choice than soft prompt tuning.

(iv) If you use a dataset of samples from Task A to do prompt tuning to generate a soft prompt which is only prepended to inputs of Task A, then performance on some other Task B with its own soft prompt might decrease due to catastrophic forgetting.

i) False. The auto-regressive process is strictly causal.

ii). False. It depends on what sort of task we are performing.

iii). False. If we don't have sufficient data, fine-tune the whole network might lead to overfitting.

iv). True

(d) Suppose that you had a family of related tasks for which you want use a frozen GPT-style language model together with learned soft-prompting to give solutions for the task. Suppose that you have substantial training data for many examples of tasks from this family. **Describe how you would adapt a meta-learning approach like MAML for this situation?**

(HINT: This is a relatively open-ended question, but you need to think about what it is that you want to learn during meta-learning, how you will learn it, and how you will use what you have learned when faced with a previously unseen task from this family.)

- In each iteration, sample data pts from different tasks and then use the model to generate soft-prompting.
- Feed the embeddings to frozen pretrained GPT and get the task-specific loss.
- Do backprop and update the model.

6. TinyML - Quantization and Pruning.

(This question has been adapted with permission from MIT 6.S965 Fall 2022)

TinyML aims at addressing the need for efficient, low-latency, and localized machine learning solutions in the age of IoT and edge computing. It enables real-time decision-making and analytics on the device itself, ensuring faster response times, lower energy consumption, and improved data privacy.

To achieve these efficiency gains, techniques like quantization and pruning become critical. Quantization reduces the size of the model and the memory footprint by representing weights and activations with fewer bits, while pruning eliminates unimportant weights or neurons, further compressing the model.

(a) Please complete `pruning.ipynb`, then answer the following questions.

i. In part 1 the histogram of weights is plotted. **What are the common characteristics of the weight distribution in the different layers?**

ii. **How do these characteristics help pruning?**

iii. After viewing the sensitivity curves, please answer the following questions. **What's the relationship between pruning sparsity and model accuracy? (i.e., does accuracy increase or decrease when sparsity becomes higher?)**

iv. **Do all the layers have the same sensitivity?**

v. **Which layer is the most sensitive to the pruning sparsity?**

vi. (Optional) After completing part 7 in the notebook, please answer the following questions. **Explain why removing 30 percent of channels roughly leads to 50 percent computation reduction.**

vii. (Optional) **Explain why the latency reduction ratio is slightly smaller than computation reduction.**

viii. (Optional) **What are the advantages and disadvantages of fine-grained pruning and channel pruning? You can discuss from the perspective of compression ratio, accuracy, latency, hardware support (*i.e.*, requiring specialized hardware accelerator), etc.**

ix. (Optional) **If you want to make your model run faster on a smartphone, which pruning method will you use? Why?**

See jupyter notebook.

method will you use? Why?

(b) Please complete `quantization.ipynb`, then answer the following questions.

i. After completing K-means Quantization, please answer the following questions. **If 4-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

ii. **If n-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

iii. After quantization aware training we see that even models that use 4 bit, or even 2 bit precision can still perform well. **Why do you think low precision quantization works at all?**

iv. (Optional) Please read through and complete up to question 4 in the notebook, then answer this question.

Recall that linear quantization can be represented as $r = S(q - Z)$. Linear quantization projects the floating point range $[fp_{min}, fp_{max}]$ to the quantized range $[quantized_{min}, quantized_{max}]$.

That is to say,

$$r_{\max} = S(q_{\max} - Z)$$
$$r_{\min} = S(q_{\min} - Z)$$

Substracting these two equations, we have,

$$S = r_{\max}/q_{\max}$$
$$S = (r_{\max} + r_{\min})/(q_{\max} + q_{\min})$$
$$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$$
$$S = r_{\max}/q_{\max} - r_{\min}/q_{\min}$$

Which of these is the correct result of subtracting the two equations?

v. (Optional) Once we determine the scaling factor S, we can directly use the relationship between $r_{\min}$ and $q_{\min}$ to calculate the zero point Z.

$$Z = \text{int}(\text{round}(r_{\min}/S - q_{\min})$$
$$Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S))$$
$$Z = q_{\min} - r_{\min}/S$$
$$Z = r_{\min}/S - q_{\min}$$

Which of these are the correct zero point?

vi. (Optional) After finishing question 9 on the notebook, **please explain why there is no ReLU layer in the linear quantized model.**

vii. (Optional) After completing the notebook, **please compare the advantages and disadvantages of k-means-based quantization and linear quantization.** You can discuss from the perspective of accuracy, latency, hardware support, etc.

hw10_quantization

April 21, 2023

1 CS 182 Homework 10: Quantization

This notebook has been adapted with permission from MIT 6.S965 Fall 2022. Original authors: Yujun Lin, Ji Lin, Zhijian Liu and Song Han

1.1 Goals

In this assignment, you will practice quantizing a classical neural network model to reduce both model size and latency. The goals of this assignment are as follows:

- Understand the basic concept of **quantization**
- Implement and apply **k-means quantization**
- Implement and apply **quantization-aware training** for k-means quantization
- Implement and apply **linear quantization**
- Implement and apply **integer-only inference** for linear quantization
- Get a basic understanding of performance improvement (such as speedup) from quantization
- Understand the differences and tradeoffs between these quantization approaches

1.2 Contents

There are 2 main sections: ***K-Means Quantization*** and ***Linear Quantization***. The second section (Questions 4-10) are OPTIONAL.

There are ***10*** questions in total: - For *K-Means Quantization*, there are ***3*** questions (Question 1-3). - For *Linear Quantization*, there are ***6*** questions (Question 4-9). - Question 10 compares k-means quantization and linear quantization.

2 Setup

First, install the required packages and download the datasets and pretrained model. Here we use CIFAR10 dataset and VGG network which is the same as what we used in the Lab 0 tutorial.

[8]:
```
print('Installing torchprofile...')
!pip install torchprofile 1>/dev/null
print('Installing fast-pytorch-kmeans...')
! pip install fast-pytorch-kmeans 1>/dev/null
print('All required packages have been successfully installed!')
```

```
Installing torchprofile…
Installing fast-pytorch-kmeans…
All required packages have been successfully installed!
```

[9]:
```
import copy
import math
import random
from collections import OrderedDict, defaultdict

from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
import numpy as np
from tqdm.auto import tqdm

import torch
from torch import nn
from torch.optim import *
from torch.optim.lr_scheduler import *
from torch.utils.data import DataLoader
from torchprofile import profile_macs
from torchvision.datasets import *
from torchvision.transforms import *

from torchprofile import profile_macs

assert torch.cuda.is_available(), \
"The current runtime does not have CUDA support." \
"Please go to menu bar (Runtime - Change runtime type) and select GPU"
```

[10]:
```
random.seed(0)
np.random.seed(0)
torch.manual_seed(0)
```

[10]:
```
<torch._C.Generator at 0x7f206c112790>
```

[11]:
```
def download_url(url, model_dir='.', overwrite=False):
    import os, sys
    from urllib.request import urlretrieve
    target_dir = url.split('/')[-1]
    model_dir = os.path.expanduser(model_dir)
    try:
        if not os.path.exists(model_dir):
            os.makedirs(model_dir)
        model_dir = os.path.join(model_dir, target_dir)
        cached_file = model_dir
        if not os.path.exists(cached_file) or overwrite:
            sys.stderr.write('Downloading: "{}" to {}\n'.format(url,␣
 ↪cached_file))
```

```
            urlretrieve(url, cached_file)
        return cached_file
    except Exception as e:
        # remove lock file so download can be executed next time.
        os.remove(os.path.join(model_dir, 'download.lock'))
        sys.stderr.write('Failed to download from url %s' % url + '\n' + str(e)␣
↪+ '\n')
        return None
```

[12]:
```
class VGG(nn.Module):
  ARCH = [64, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M']

  def __init__(self) -> None:
    super().__init__()

    layers = []
    counts = defaultdict(int)

    def add(name: str, layer: nn.Module) -> None:
      layers.append((f"{name}{counts[name]}", layer))
      counts[name] += 1

    in_channels = 3
    for x in self.ARCH:
      if x != 'M':
        # conv-bn-relu
        add("conv", nn.Conv2d(in_channels, x, 3, padding=1, bias=False))
        add("bn", nn.BatchNorm2d(x))
        add("relu", nn.ReLU(True))
        in_channels = x
      else:
        # maxpool
        add("pool", nn.MaxPool2d(2))
    add("avgpool", nn.AvgPool2d(2))
    self.backbone = nn.Sequential(OrderedDict(layers))
    self.classifier = nn.Linear(512, 10)

  def forward(self, x: torch.Tensor) -> torch.Tensor:
    # backbone: [N, 3, 32, 32] => [N, 512, 2, 2]
    x = self.backbone(x)

    # avgpool: [N, 512, 2, 2] => [N, 512]
    # x = x.mean([2, 3])
    x = x.view(x.shape[0], -1)

    # classifier: [N, 512] => [N, 10]
    x = self.classifier(x)
```

```
    return x
```

[13]:
```
def train(
  model: nn.Module,
  dataloader: DataLoader,
  criterion: nn.Module,
  optimizer: Optimizer,
  scheduler: LambdaLR,
  callbacks = None
) -> None:
  model.train()

  for inputs, targets in tqdm(dataloader, desc='train', leave=False):
    # Move the data from CPU to GPU
    inputs = inputs.cuda()
    targets = targets.cuda()

    # Reset the gradients (from the last iteration)
    optimizer.zero_grad()

    # Forward inference
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # Backward propagation
    loss.backward()

    # Update optimizer and LR scheduler
    optimizer.step()
    scheduler.step()

    if callbacks is not None:
        for callback in callbacks:
            callback()
```

[14]:
```
@torch.inference_mode()
def evaluate(
  model: nn.Module,
  dataloader: DataLoader,
  extra_preprocess = None
) -> float:
  model.eval()

  num_samples = 0
  num_correct = 0

  for inputs, targets in tqdm(dataloader, desc="eval", leave=False):
```

```
    # Move the data from CPU to GPU
    inputs = inputs.cuda()
    if extra_preprocess is not None:
        for preprocess in extra_preprocess:
            inputs = preprocess(inputs)

    targets = targets.cuda()

    # Inference
    outputs = model(inputs)

    # Convert logits to class indices
    outputs = outputs.argmax(dim=1)

    # Update metrics
    num_samples += targets.size(0)
    num_correct += (outputs == targets).sum()

  return (num_correct / num_samples * 100).item()
```

Helpler Functions (Flops, Model Size calculation, etc.)

[15]:
```
def get_model_flops(model, inputs):
    num_macs = profile_macs(model, inputs)
    return num_macs
```

[16]:
```
def get_model_size(model: nn.Module, data_width=32):
    """
    calculate the model size in bits
    :param data_width: #bits per element
    """
    num_elements = 0
    for param in model.parameters():
        num_elements += param.numel()
    return num_elements * data_width

Byte = 8
KiB = 1024 * Byte
MiB = 1024 * KiB
GiB = 1024 * MiB
```

Define misc funcions for verification.

[17]:
```
def test_k_means_quantize(
    test_tensor=torch.tensor([
        [-0.3747,  0.0874,  0.3200, -0.4868,  0.4404],
        [-0.0402,  0.2322, -0.2024, -0.4986,  0.1814],
        [ 0.3102, -0.3942, -0.2030,  0.0883, -0.4741],
```

```
        [-0.1592, -0.0777, -0.3946, -0.2128,  0.2675],
        [ 0.0611, -0.1933, -0.4350,  0.2928, -0.1087]]),
    bitwidth=2):
    def plot_matrix(tensor, ax, title, cmap=ListedColormap(['white'])):
        ax.imshow(tensor.cpu().numpy(), vmin=-0.5, vmax=0.5, cmap=cmap)
        ax.set_title(title)
        ax.set_yticklabels([])
        ax.set_xticklabels([])
        for i in range(tensor.shape[1]):
            for j in range(tensor.shape[0]):
                text = ax.text(j, i, f'{tensor[i, j].item():.2f}',
                               ha="center", va="center", color="k")

    fig, axes = plt.subplots(1,2, figsize=(8, 12))
    ax_left, ax_right = axes.ravel()

    plot_matrix(test_tensor, ax_left, 'original tensor')

    num_unique_values_before_quantization = test_tensor.unique().numel()
    k_means_quantize(test_tensor, bitwidth=bitwidth)
    num_unique_values_after_quantization = test_tensor.unique().numel()
    print('* Test k_means_quantize()')
    print(f'    target bitwidth: {bitwidth} bits')
    print(f'        num unique values before k-means quantization:␣
↪{num_unique_values_before_quantization}')
    print(f'        num unique values after  k-means quantization:␣
↪{num_unique_values_after_quantization}')
    assert num_unique_values_after_quantization == min((1 << bitwidth),␣
↪num_unique_values_before_quantization)
    print('* Test passed.')

    plot_matrix(test_tensor, ax_right, f'{bitwidth}-bit k-means quantized␣
↪tensor', cmap='tab20c')
    fig.tight_layout()
    plt.show()
```

[18]:
```
def test_linear_quantize(
    test_tensor=torch.tensor([
        [ 0.0523,  0.6364, -0.0968, -0.0020,  0.1940],
        [ 0.7500,  0.5507,  0.6188, -0.1734,  0.4677],
        [-0.0669,  0.3836,  0.4297,  0.6267, -0.0695],
        [ 0.1536, -0.0038,  0.6075,  0.6817,  0.0601],
        [ 0.6446, -0.2500,  0.5376, -0.2226,  0.2333]]),
    quantized_test_tensor=torch.tensor([
        [-1,  1, -1, -1,  0],
        [ 1,  1,  1, -2,  0],
        [-1,  0,  0,  1, -1],
```

```
        [-1, -1,  1,  1, -1],
        [ 1, -2,  1, -2,  0]], dtype=torch.int8),
    real_min=-0.25, real_max=0.75, bitwidth=2, scale=1/3, zero_point=-1):
    def plot_matrix(tensor, ax, title, vmin=0, vmax=1,␣
↪cmap=ListedColormap(['white'])):
        ax.imshow(tensor.cpu().numpy(), vmin=vmin, vmax=vmax, cmap=cmap)
        ax.set_title(title)
        ax.set_yticklabels([])
        ax.set_xticklabels([])
        for i in range(tensor.shape[0]):
            for j in range(tensor.shape[1]):
                datum = tensor[i, j].item()
                if isinstance(datum, float):
                    text = ax.text(j, i, f'{datum:.2f}',
                                   ha="center", va="center", color="k")
                else:
                    text = ax.text(j, i, f'{datum}',
                                   ha="center", va="center", color="k")
    quantized_min, quantized_max = get_quantized_range(bitwidth)
    fig, axes = plt.subplots(1,3, figsize=(10, 32))
    plot_matrix(test_tensor, axes[0], 'original tensor', vmin=real_min,␣
↪vmax=real_max)
    _quantized_test_tensor = linear_quantize(
        test_tensor, bitwidth=bitwidth, scale=scale, zero_point=zero_point)
    _reconstructed_test_tensor = scale * (_quantized_test_tensor.float() -␣
↪zero_point)
    print('* Test linear_quantize()')
    print(f'    target bitwidth: {bitwidth} bits')
    print(f'        scale: {scale}')
    print(f'        zero point: {zero_point}')
    assert _quantized_test_tensor.equal(quantized_test_tensor)
    print('* Test passed.')
    plot_matrix(_quantized_test_tensor, axes[1], f'2-bit linear quantized␣
↪tensor',
                vmin=quantized_min, vmax=quantized_max, cmap='tab20c')
    plot_matrix(_reconstructed_test_tensor, axes[2], f'reconstructed tensor',
                vmin=real_min, vmax=real_max, cmap='tab20c')
    fig.tight_layout()
    plt.show()
```

[19]:
```
def test_quantized_fc(
    input=torch.tensor([
        [0.6118, 0.7288, 0.8511, 0.2849, 0.8427, 0.7435, 0.4014, 0.2794],
        [0.3676, 0.2426, 0.1612, 0.7684, 0.6038, 0.0400, 0.2240, 0.4237],
        [0.6565, 0.6878, 0.4670, 0.3470, 0.2281, 0.8074, 0.0178, 0.3999],
        [0.1863, 0.3567, 0.6104, 0.0497, 0.0577, 0.2990, 0.6687, 0.8626]]),
    weight=torch.tensor([
```

```
        [ 1.2626e-01, -1.4752e-01,  8.1910e-02,  2.4982e-01, -1.0495e-01,
         -1.9227e-01, -1.8550e-01, -1.5700e-01],
        [ 2.7624e-01, -4.3835e-01,  5.1010e-02, -1.2020e-01, -2.0344e-01,
          1.0202e-01, -2.0799e-01,  2.4112e-01],
        [-3.8216e-01, -2.8047e-01,  8.5238e-02, -4.2504e-01, -2.0952e-01,
          3.2018e-01, -3.3619e-01,  2.0219e-01],
        [ 8.9233e-02, -1.0124e-01,  1.1467e-01,  2.0091e-01,  1.1438e-01,
         -4.2427e-01,  1.0178e-01, -3.0941e-04],
        [-1.8837e-02, -2.1256e-01, -4.5285e-01,  2.0949e-01, -3.8684e-01,
         -1.7100e-01, -4.5331e-01, -2.0433e-01],
        [-2.0038e-01, -5.3757e-02,  1.8997e-01, -3.6866e-01,  5.5484e-02,
          1.5643e-01, -2.3538e-01,  2.1103e-01],
        [-2.6875e-01,  2.4984e-01, -2.3514e-01,  2.5527e-01,  2.0322e-01,
          3.7675e-01,  6.1563e-02,  1.7201e-01],
        [ 3.3541e-01, -3.3555e-01, -4.3349e-01,  4.3043e-01, -2.0498e-01,
         -1.8366e-01, -9.1553e-02, -4.1168e-01]]),
   bias=torch.tensor([ 0.1954, -0.2756,  0.3113,  0.1149,  0.4274,  0.2429, -0.
↪1721, -0.2502]),
   quantized_bias=torch.tensor([ 3, -2,  3,  1,  3,  2, -2, -2], dtype=torch.
↪int32),
   shifted_quantized_bias=torch.tensor([-1,  0, -3, -1, -3,  0,  2, -4],␣
↪dtype=torch.int32),
   calc_quantized_output=torch.tensor([
        [ 0, -1,  0, -1, -1,  0,  1, -2],
        [ 0,  0, -1,  0,  0,  0,  0, -1],
        [ 0,  0,  0, -1,  0,  0,  0, -1],
        [ 0,  0,  0,  0,  0,  1, -1, -2]], dtype=torch.int8),
   bitwidth=2, batch_size=4, in_channels=8, out_channels=8):
   def plot_matrix(tensor, ax, title, vmin=0, vmax=1,␣
↪cmap=ListedColormap(['white'])):
        ax.imshow(tensor.cpu().numpy(), vmin=vmin, vmax=vmax, cmap=cmap)
        ax.set_title(title)
        ax.set_yticklabels([])
        ax.set_xticklabels([])
        for i in range(tensor.shape[0]):
            for j in range(tensor.shape[1]):
                datum = tensor[i, j].item()
                if isinstance(datum, float):
                    text = ax.text(j, i, f'{datum:.2f}',
                                   ha="center", va="center", color="k")
                else:
                    text = ax.text(j, i, f'{datum}',
                                   ha="center", va="center", color="k")

   output = torch.nn.functional.linear(input, weight, bias)

   quantized_weight, weight_scale, weight_zero_point = \
```

```
        linear_quantize_weight_per_channel(weight, bitwidth)
    quantized_input, input_scale, input_zero_point = \
        linear_quantize_feature(input, bitwidth)
    _quantized_bias, bias_scale, bias_zero_point = \
        linear_quantize_bias_per_output_channel(bias, weight_scale, input_scale)
    assert _quantized_bias.equal(_quantized_bias)
    _shifted_quantized_bias = \
        shift_quantized_linear_bias(quantized_bias, quantized_weight,␣
↪input_zero_point)
    assert _shifted_quantized_bias.equal(shifted_quantized_bias)
    quantized_output, output_scale, output_zero_point = \
        linear_quantize_feature(output, bitwidth)

    _calc_quantized_output = quantized_linear(
        quantized_input, quantized_weight, shifted_quantized_bias,
        bitwidth, bitwidth,
        input_zero_point, output_zero_point,
        input_scale, weight_scale, output_scale)
    assert _calc_quantized_output.equal(calc_quantized_output)

    reconstructed_weight = weight_scale * (quantized_weight.float() -␣
↪weight_zero_point)
    reconstructed_input = input_scale * (quantized_input.float() -␣
↪input_zero_point)
    reconstructed_bias = bias_scale * (quantized_bias.float() - bias_zero_point)
    reconstructed_calc_output = output_scale * (calc_quantized_output.float() -␣
↪output_zero_point)

    fig, axes = plt.subplots(3,3, figsize=(15, 12))
    quantized_min, quantized_max = get_quantized_range(bitwidth)
    plot_matrix(weight, axes[0, 0], 'original weight', vmin=-0.5, vmax=0.5)
    plot_matrix(input.t(), axes[1, 0], 'original input', vmin=0, vmax=1)
    plot_matrix(output.t(), axes[2, 0], 'original output', vmin=-1.5, vmax=1.5)
    plot_matrix(quantized_weight, axes[0, 1], f'{bitwidth}-bit linear quantized␣
↪weight',
                vmin=quantized_min, vmax=quantized_max, cmap='tab20c')
    plot_matrix(quantized_input.t(), axes[1, 1], f'{bitwidth}-bit linear␣
↪quantized input',
                vmin=quantized_min, vmax=quantized_max, cmap='tab20c')
    plot_matrix(calc_quantized_output.t(), axes[2, 1], f'quantized output from␣
↪quantized_linear()',
                vmin=quantized_min, vmax=quantized_max, cmap='tab20c')
    plot_matrix(reconstructed_weight, axes[0, 2], f'reconstructed weight',
                vmin=-0.5, vmax=0.5, cmap='tab20c')
    plot_matrix(reconstructed_input.t(), axes[1, 2], f'reconstructed input',
                vmin=0, vmax=1, cmap='tab20c')
```

```
    plot_matrix(reconstructed_calc_output.t(), axes[2, 2], f'reconstructed␣
 ↪output',
                vmin=-1.5, vmax=1.5, cmap='tab20c')

    print('* Test quantized_fc()')
    print(f'    target bitwidth: {bitwidth} bits')
    print(f'      batch size: {batch_size}')
    print(f'      input channels: {in_channels}')
    print(f'      output channels: {out_channels}')
    print('* Test passed.')
    fig.tight_layout()
    plt.show()
```

Load Pretrained Model

[21]:
```
checkpoint_url = "https://inst.eecs.berkeley.edu/~cs182/sp23/assets/data/vgg.
 ↪cifar.pretrained.pth"
# checkpoint_url = "https://hanlab.mit.edu/files/course/labs/vgg.cifar.
 ↪pretrained.pth"
checkpoint = torch.load(download_url(checkpoint_url), map_location="cpu")
model = VGG().cuda()
print(f"=> loading checkpoint '{checkpoint_url}'")
model.load_state_dict(checkpoint['state_dict'])
recover_model = lambda : model.load_state_dict(checkpoint['state_dict'])
```

```
Downloading: "https://inst.eecs.berkeley.edu/~cs182/sp23/assets/data/vgg.cifar.p
retrained.pth" to ./vgg.cifar.pretrained.pth

=> loading checkpoint 'https://inst.eecs.berkeley.edu/~cs182/sp23/assets/data/vg
g.cifar.pretrained.pth'
```

[22]:
```
image_size = 32
transforms = {
    "train": Compose([
        RandomCrop(image_size, padding=4),
        RandomHorizontalFlip(),
        ToTensor(),
    ]),
    "test": ToTensor(),
}
dataset = {}
for split in ["train", "test"]:
  dataset[split] = CIFAR10(
    root="data/cifar10",
    train=(split == "train"),
    download=True,
    transform=transforms[split],
  )
```

```
dataloader = {}
for split in ['train', 'test']:
  dataloader[split] = DataLoader(
    dataset[split],
    batch_size=512,
    shuffle=(split == 'train'),
    num_workers=0,
    pin_memory=True,
  )
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
data/cifar10/cifar-10-python.tar.gz
```

```
100%|      | 170498071/170498071 [00:05<00:00, 29375728.67it/s]
```

```
Extracting data/cifar10/cifar-10-python.tar.gz to data/cifar10
Files already downloaded and verified
```

3 Let's First Evaluate the Accuracy and Model Size of the FP32 Model

[42]:
```
fp32_model_accuracy = evaluate(model, dataloader['test'])
fp32_model_size = get_model_size(model)
print(f"fp32 model has accuracy={fp32_model_accuracy:.2f}%")
print(f"fp32 model has size={fp32_model_size/MiB:.2f} MiB")
```

```
eval:   0%|          | 0/20 [00:00<?, ?it/s]
```

```
fp32 model has accuracy=92.75%
fp32 model has size=35.20 MiB
```

4 K-Means Quantization

Network quantization compresses the network by reducing the bits per weight required to represent the deep network. The quantized network can have a faster inference speed with hardware support.

In this section, we will explore the K-means quantization for neural networks as in Deep Compression: Compressing Deep Neural Networks With Pruning, Trained Quantization And Huffman Coding.

A n-bit k-means quantization will divide synapses into 2^n clusters, and synapses in the same cluster will share the same weight value.

Therefore, k-means quantization will create a codebook, inlcuding * `centroids`: 2^n fp32 cluster centers. * `labels`: a n-bit integer tensor with the same #elements of the original fp32 weights tensor. Each integer indicates which cluster it belongs to.

During the inference, a fp32 tensor is generated based on the codebook for inference:

$$quantized\_weight = codebook.centroids[codebook.labels].view\_as(weight)$$

[23]:
```
from collections import namedtuple

Codebook = namedtuple('Codebook', ['centroids', 'labels'])
```

4.1 Question 1

Please complete the following K-Means quantization function.

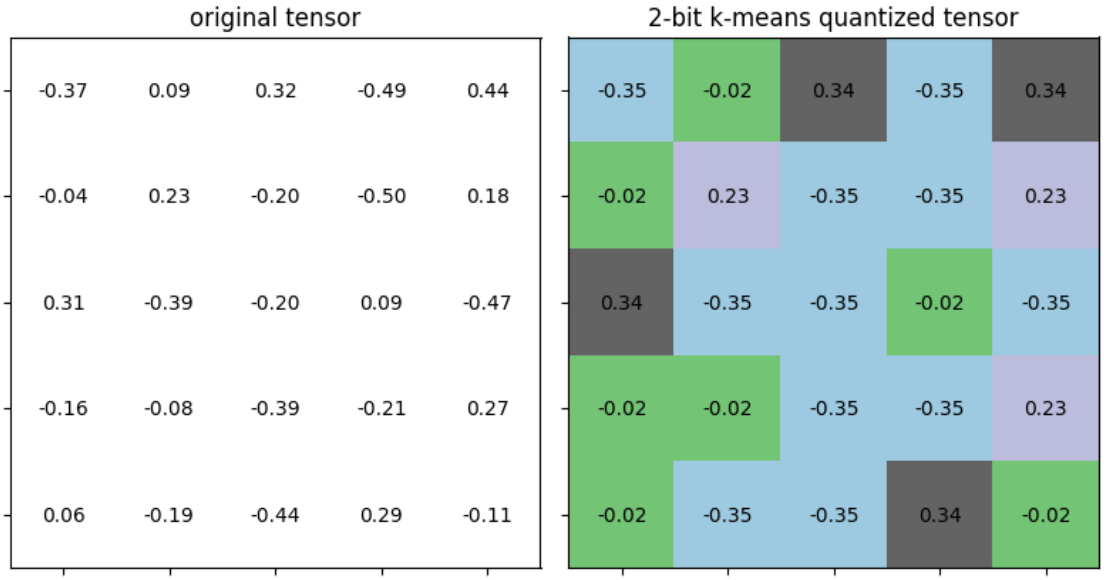
[34]:
```
from fast_pytorch_kmeans import KMeans

def k_means_quantize(fp32_tensor: torch.Tensor, bitwidth=4, codebook=None):
    """
    quantize tensor using k-means clustering
    :param fp32_tensor:
    :param bitwidth: [int] quantization bit width, default=4
    :param codebook: [Codebook] (the cluster centroids, the cluster label␣
 ↪tensor)
    :return:
        [Codebook = (centroids, labels)]
            centroids: [torch.(cuda.)FloatTensor] the cluster centroids
            labels: [torch.(cuda.)LongTensor] cluster label tensor
    """
    if codebook is None:
        ################ YOUR CODE STARTS HERE ################
        # get number of clusters based on the quantization precision
        # hint: one line of code
        n_clusters = 2 ** bitwidth
        ################ YOUR CODE ENDS HERE ##################
        # use k-means to get the quantization centroids
        kmeans = KMeans(n_clusters=n_clusters, mode='euclidean', verbose=0)
        print(fp32_tensor.view(-1, 1).shape)
        labels = kmeans.fit_predict(fp32_tensor.view(-1, 1)).to(torch.long)
        centroids = kmeans.centroids.to(torch.float).view(-1)
        codebook = Codebook(centroids, labels)
    ################ YOUR CODE STARTS HERE ################
    # decode the codebook into k-means quantized tensor for inference
    # hint: one line of code
    quantized_tensor = codebook.centroids[codebook.labels].view_as(fp32_tensor)
    ################ YOUR CODE ENDS HERE ##################
    fp32_tensor.set_(quantized_tensor.view_as(fp32_tensor))
    return codebook
```

Let's verify the functionality of defined k-means quantization by applying the function above on a dummy tensor.

[35]:
```
test_k_means_quantize()
```

```
torch.Size([25, 1])
* Test k_means_quantize()
    target bitwidth: 2 bits
        num unique values before k-means quantization: 25
        num unique values after  k-means quantization: 4
* Test passed.
```



4.2 Question 2

The last code cell performs 2-bit k-means quantization and plots the tensor before and after the quantization. Each cluster is rendered with a unique color. There are 4 unique colors rendered in the quantized tensor.

Given this observation, please answer the following questions.

4.2.1 Question 2.1

If 4-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?

Your Answer: $2^4 = 16$

4.2.2 Question 2.2

If n-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?

Your Answer: 2^n

4.3 K-Means Quantization on Whole Model

Similar to what we did in the pruning section, we now wrap the k-means quantization function into a class for quantizing the whole model. In class `KMeansQuantizer`, we have to keep a record of the codebooks (i.e., `centroids` and `labels`) so that we could apply or update the codebooks whenever the model weights change.

[36]:
```
from torch.nn import parameter
class KMeansQuantizer:
    def __init__(self, model : nn.Module, bitwidth=4):
        self.codebook = KMeansQuantizer.quantize(model, bitwidth)

    @torch.no_grad()
    def apply(self, model, update_centroids):
        for name, param in model.named_parameters():
            if name in self.codebook:
                if update_centroids:
                    update_codebook(param, codebook=self.codebook[name])
                self.codebook[name] = k_means_quantize(
                    param, codebook=self.codebook[name])

    @staticmethod
    @torch.no_grad()
    def quantize(model: nn.Module, bitwidth=4):
        codebook = dict()
        if isinstance(bitwidth, dict):
            for name, param in model.named_parameters():
                if name in bitwidth:
                    codebook[name] = k_means_quantize(param,␣
 ↪bitwidth=bitwidth[name])
        else:
            for name, param in model.named_parameters():
                if param.dim() > 1:
                    codebook[name] = k_means_quantize(param, bitwidth=bitwidth)
        return codebook
```

Now let's quantize model into 8 bits, 4 bits and 2 bits using K-Means Quantization. *Note that we ignore the storage for codebooks when calculating the model size.*

[37]:
```
print('Note that the storage for codebooks is ignored when calculating the␣
 ↪model size.')
quantizers = dict()
for bitwidth in [8, 4, 2]:
    recover_model()
    print(f'k-means quantizing model into {bitwidth} bits')
    quantizer = KMeansQuantizer(model, bitwidth)
    quantized_model_size = get_model_size(model, bitwidth)
```

```
    print(f"    {bitwidth}-bit k-means quantized model has␣
↪size={quantized_model_size/MiB:.2f} MiB")
    quantized_model_accuracy = evaluate(model, dataloader['test'])
    print(f"    {bitwidth}-bit k-means quantized model has␣
↪accuracy={quantized_model_accuracy:.2f}%")
    quantizers[bitwidth] = quantizer
```

```
Note that the storage for codebooks is ignored when calculating the model size.
k-means quantizing model into 8 bits
torch.Size([1728, 1])
torch.Size([73728, 1])
torch.Size([294912, 1])
torch.Size([589824, 1])
torch.Size([1179648, 1])
torch.Size([2359296, 1])
torch.Size([2359296, 1])
torch.Size([2359296, 1])
torch.Size([5120, 1])
    8-bit k-means quantized model has size=8.80 MiB
eval:   0%|          | 0/20 [00:00<?, ?it/s]
    8-bit k-means quantized model has accuracy=92.75%
k-means quantizing model into 4 bits
torch.Size([1728, 1])
torch.Size([73728, 1])
torch.Size([294912, 1])
torch.Size([589824, 1])
torch.Size([1179648, 1])
torch.Size([2359296, 1])
torch.Size([2359296, 1])
torch.Size([2359296, 1])
torch.Size([5120, 1])
    4-bit k-means quantized model has size=4.40 MiB
eval:   0%|          | 0/20 [00:00<?, ?it/s]
    4-bit k-means quantized model has accuracy=84.01%
k-means quantizing model into 2 bits
torch.Size([1728, 1])
torch.Size([73728, 1])
torch.Size([294912, 1])
torch.Size([589824, 1])
torch.Size([1179648, 1])
torch.Size([2359296, 1])
torch.Size([2359296, 1])
torch.Size([2359296, 1])
torch.Size([5120, 1])
    2-bit k-means quantized model has size=2.20 MiB
```

```
eval:   0%|          | 0/20 [00:00<?, ?it/s]
```

```
    2-bit k-means quantized model has accuracy=12.26%
```

4.4 Trained K-Means Quantization

As we can see from the results of last cell, the accuracy significantly drops when quantizing the model into lower bits. Therefore, we have to perform quantization-aware training to recover the accuracy.

During the k-means quantization-aware training, the centroids are also updated, which is proposed in Deep Compression: Compressing Deep Neural Networks With Pruning, Trained Quantization And Huffman Coding.

The gradient for the centroids is calculated as,

$$\frac{\partial \mathcal{L}}{\partial C_k} = \sum_j \frac{\partial \mathcal{L}}{\partial W_j} \frac{\partial W_j}{\partial C_k} = \sum_j \frac{\partial \mathcal{L}}{\partial W_j} \mathbf{1}(I_j = k)$$

where $\mathcal{L}$ is the loss, C_k is k-th centroid, I_j is the label for weight W_j. $\mathbf{1}()$ is the indicator function, and $\mathbf{1}(I_j = k)$ means 1 if $I_j = k$ else 0, *i.e.*, $I_j == k$.

Here in the lab, **for simplicity**, we directly update the centroids according to the latest weights:

$$C_k = \frac{\sum_j W_j \mathbf{1}(I_j=k)}{\sum_j \mathbf{1}(I_j=k)}$$

4.4.1 Question 3

Please complete the following codebook update function.

Hint:

The above equation for updating centroids is indeed using the `mean` of weights in the same cluster to be the updated centroid value.

[40]:
```
def update_codebook(fp32_tensor: torch.Tensor, codebook: Codebook):
    """
    update the centroids in the codebook using updated fp32_tensor
    :param fp32_tensor: [torch.(cuda.)Tensor]
    :param codebook: [Codebook] (the cluster centroids, the cluster label␣
 ↪tensor)
    """
    n_clusters = codebook.centroids.numel()
    fp32_tensor = fp32_tensor.view(-1)
    for k in range(n_clusters):
    ################ YOUR CODE STARTS HERE ###############
        # hint: one line of code
        codebook.centroids[k] = torch.sum((codebook.labels==k) * fp32_tensor)/␣
 ↪torch.sum(codebook.labels==k)
    ################ YOUR CODE ENDS HERE #################
```

Now let's run the following code cell to finetune the k-means quantized model to recover the accuracy. We will stop finetuning if accuracy drop is less than 0.5.

[43]:
```
accuracy_drop_threshold = 0.5
quantizers_before_finetune = copy.deepcopy(quantizers)
quantizers_after_finetune = quantizers

for bitwidth in [8, 4, 2]:
    recover_model()
    quantizer = quantizers[bitwidth]
    print(f'k-means quantizing model into {bitwidth} bits')
    quantizer.apply(model, update_centroids=False)
    quantized_model_size = get_model_size(model, bitwidth)
    print(f"    {bitwidth}-bit k-means quantized model has␣
 ↪size={quantized_model_size/MiB:.2f} MiB")
    quantized_model_accuracy = evaluate(model, dataloader['test'])
    print(f"    {bitwidth}-bit k-means quantized model has␣
 ↪accuracy={quantized_model_accuracy:.2f}% before quantization-aware training␣
 ↪")
    accuracy_drop = fp32_model_accuracy - quantized_model_accuracy
    if accuracy_drop > accuracy_drop_threshold:
        print(f"        Quantization-aware training due to accuracy␣
 ↪drop={accuracy_drop:.2f}% is larger than threshold={accuracy_drop_threshold:.
 ↪2f}%")
        num_finetune_epochs = 5
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,␣
 ↪num_finetune_epochs)
        criterion = nn.CrossEntropyLoss()
        best_accuracy = 0
        epoch = num_finetune_epochs
        while accuracy_drop > accuracy_drop_threshold and epoch > 0:
            train(model, dataloader['train'], criterion, optimizer, scheduler,
                  callbacks=[lambda: quantizer.apply(model,␣
 ↪update_centroids=True)])
            model_accuracy = evaluate(model, dataloader['test'])
            is_best = model_accuracy > best_accuracy
            best_accuracy = max(model_accuracy, best_accuracy)
            print(f'        Epoch {num_finetune_epochs-epoch} Accuracy␣
 ↪{model_accuracy:.2f}% / Best Accuracy: {best_accuracy:.2f}%')
            accuracy_drop = fp32_model_accuracy - best_accuracy
            epoch -= 1
    else:
        print(f"        No need for quantization-aware training since accuracy␣
 ↪drop={accuracy_drop:.2f}% is smaller than threshold={accuracy_drop_threshold:
 ↪.2f}%")
```

```
k-means quantizing model into 8 bits
    8-bit k-means quantized model has size=8.80 MiB
```

```
eval:   0%|          | 0/20 [00:00<?, ?it/s]
```

```
    8-bit k-means quantized model has accuracy=92.75% before quantization-aware
training
        No need for quantization-aware training since accuracy drop=0.00% is
smaller than threshold=0.50%
k-means quantizing model into 4 bits
    4-bit k-means quantized model has size=4.40 MiB
eval:   0%|          | 0/20 [00:00<?, ?it/s]
    4-bit k-means quantized model has accuracy=84.01% before quantization-aware
training
        Quantization-aware training due to accuracy drop=8.74% is larger than
threshold=0.50%
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:   0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 0 Accuracy 92.23% / Best Accuracy: 92.23%
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:   0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 1 Accuracy 92.31% / Best Accuracy: 92.31%
k-means quantizing model into 2 bits
    2-bit k-means quantized model has size=2.20 MiB
eval:   0%|          | 0/20 [00:00<?, ?it/s]
    2-bit k-means quantized model has accuracy=12.26% before quantization-aware
training
        Quantization-aware training due to accuracy drop=80.49% is larger than
threshold=0.50%
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:   0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 0 Accuracy 90.29% / Best Accuracy: 90.29%
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:   0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 1 Accuracy 90.89% / Best Accuracy: 90.89%
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:   0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 2 Accuracy 91.32% / Best Accuracy: 91.32%
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:   0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 3 Accuracy 91.38% / Best Accuracy: 91.38%
```

```
train:   0%|          | 0/98 [00:00<?, ?it/s]
eval:    0%|          | 0/20 [00:00<?, ?it/s]
        Epoch 4 Accuracy 91.46% / Best Accuracy: 91.46%
```

4.4.2 Question 3.1

After quantization aware training we see that even models that use 4 bit, or even 2 bit precision can still perform well. Why do you think low precision quantization works at all?

Your Answer: Sometimes the higher bits are not that significant.

5 Linear Quantization (OPTIONAL)

In this section, we will implement and perform linear quantization.

Linear quantization directly rounds the floating-point value into the nearest quantized integer after range truncation and scaling.

Linear quantization can be represented as

$r = S(q - Z)$

where r is a floating point real number, q is a n-bit integer, Z is a n-bit integer, and S is a floating point real number. Z is quantization zero point and S is quantization scaling factor. Both constant Z and S are quantization parameters.

5.1 n-bit Integer (OPTIONAL)

A n-bit signed integer is usually represented in two's complement notation.

A n-bit signed integer can enode integers in the range $[-2^{n-1}, 2^{n-1} - 1]$. For example, a 8-bit integer falls in the range [-128, 127].

[ ]:
```python
def get_quantized_range(bitwidth):
    quantized_max = (1 << (bitwidth - 1)) - 1
    quantized_min = -(1 << (bitwidth - 1))
    return quantized_min, quantized_max
```

5.2 Question 4 (OPTIONAL)

Please complete the following linear quantization function.

Hint: * From $r = S(q - Z)$, we have $q = r/S + Z$. * Both r and S are floating numbers, and thus we cannot directly add integer Z to r/S. Therefore $q = \text{int}(\text{round}(r/S)) + Z$. * To convert `torch.FloatTensor` to `torch.IntTensor`, we could use `torch.round()`, `torch.Tensor.round()`, `torch.Tensor.round_()` to first convert all values to floating integer, and then use `torch.Tensor.to(torch.int8)` to convert the data type from `torch.float` to `torch.int8`.

[ ]:
```python
def linear_quantize(fp_tensor, bitwidth, scale, zero_point, dtype=torch.int8)␣
 ↪-> torch.Tensor:
```

```
    """
    linear quantization for single fp_tensor
      from
        fp_tensor = (quantized_tensor - zero_point) * scale
      we have,
        quantized_tensor = int(round(fp_tensor / scale)) + zero_point
    :param tensor: [torch.(cuda.)FloatTensor] floating tensor to be quantized
    :param bitwidth: [int] quantization bit width
    :param scale: [torch.(cuda.)FloatTensor] scaling factor
    :param zero_point: [torch.(cuda.)IntTensor] the desired centroid of tensor␣
↪values
    :return:
        [torch.(cuda.)FloatTensor] quantized tensor whose values are integers
    """
    assert(fp_tensor.dtype == torch.float)
    assert(isinstance(scale, float) or
           (scale.dtype == torch.float and scale.dim() == fp_tensor.dim()))
    assert(isinstance(zero_point, int) or
           (zero_point.dtype == dtype and zero_point.dim() == fp_tensor.dim()))

    ################ YOUR CODE STARTS HERE ################
    # Step 1: scale the fp_tensor
    scaled_tensor = 0
    # Step 2: round the floating value to integer value
    rounded_tensor = 0
    ################ YOUR CODE ENDS HERE ##################

    rounded_tensor = rounded_tensor.to(dtype)

    ################ YOUR CODE STARTS HERE ################
    # Step 3: shift the rounded_tensor to make zero_point 0
    shifted_tensor = 0
    ################ YOUR CODE ENDS HERE ##################

    # Step 4: clamp the shifted_tensor to lie in bitwidth-bit range
    quantized_min, quantized_max = get_quantized_range(bitwidth)
    quantized_tensor = shifted_tensor.clamp_(quantized_min, quantized_max)
    return quantized_tensor
```

Let's verify the functionality of defined linear quantization by applying the function above on a dummy tensor.

[ ]:
```
test_linear_quantize()
```

5.3 Question 5 (OPTIONAL)

Now we have to determine the scaling factor S and zero point Z for linear quantization.

Recall that linear quantization can be represented as

$r = S(q - Z)$

5.3.1 Scale

Linear quantization projects the floating point range [*fp_min*, *fp_max*] to the quantized range [*quantized_min*, *quantized_max*]. That is to say,

$$r_{\max} = S(q_{\max} - Z)$$

$$r_{\min} = S(q_{\min} - Z)$$

Substracting these two equations, we have,

Question 5.1 (1 pts) Please select the correct answer and delete the wrong answers in the next text cell.

$S = r_{\max}/q_{\max}$

$S = (r_{\max} + r_{\min})/(q_{\max} + q_{\min})$

$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$

$S = r_{\max}/q_{\max} - r_{\min}/q_{\min}$

$S = (r_{\max} + r_{\min})/(q_{\max} + q_{\min})$

$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$

$S = r_{\max}/q_{\max} - r_{\min}/q_{\min}$

There are different approaches to determine the $r_{\min}$ and $r_{\max}$ of a floating point tensor `fp_tensor`.

- The most common method is directly using the minimum and maximum value of `fp_tensor`.
- Another widely used method is minimizing Kullback-Leibler-J divergence to determine the *fp_max*.

5.3.2 zero point

Once we determine the scaling factor S, we can directly use the relationship between $r_{\min}$ and $q_{\min}$ to calculate the zero point Z.

Question 5.2 (OPTIONAL) Please select the correct answer and delete the wrong answers in the next text cell.

2

5.3.3 Question 5.3 (OPTIONAL)

Please complete the following function for calculating the scale S and zero point Z from floating point tensor r.

[ ]:
```
def get_quantization_scale_and_zero_point(fp_tensor, bitwidth):
    """
    get quantization scale for single tensor
```

```
    :param fp_tensor: [torch.(cuda.)Tensor] floating tensor to be quantized
    :param bitwidth: [int] quantization bit width
    :return:
        [float] scale
        [int] zero_point
    """
    quantized_min, quantized_max = get_quantized_range(bitwidth)
    fp_max = fp_tensor.max().item()
    fp_min = fp_tensor.min().item()

    ################ YOUR CODE STARTS HERE ################
    # hint: one line of code for calculating scale
    scale = 0
    # hint: one line of code for calculating zero_point
    zero_point = 0
    ################ YOUR CODE ENDS HERE ##################

    # clip the zero_point to fall in [quantized_min, quantized_max]
    if zero_point < quantized_min:
        zero_point = quantized_min
    elif zero_point > quantized_max:
        zero_point = quantized_max
    else: # convert from float to int using round()
        zero_point = round(zero_point)
    return scale, int(zero_point)
```

We now wrap `linear_quantize()` in Question 4 and `get_quantization_scale_and_zero_point()` in Question 5 into one function.

[ ]:
```
def linear_quantize_feature(fp_tensor, bitwidth):
    """
    linear quantization for feature tensor
    :param fp_tensor: [torch.(cuda.)Tensor] floating feature to be quantized
    :param bitwidth: [int] quantization bit width
    :return:
        [torch.(cuda.)Tensor] quantized tensor
        [float] scale tensor
        [int] zero point
    """
    scale, zero_point = get_quantization_scale_and_zero_point(fp_tensor,␣
 ↪bitwidth)
    quantized_tensor = linear_quantize(fp_tensor, bitwidth, scale, zero_point)
    return quantized_tensor, scale, zero_point
```

5.4 Special case: linear quantization on weight tensor (OPTIONAL)

Let's first see the distribution of weight values.

[ ]:
```
def plot_weight_distribution(model, bitwidth=32):
    # bins = (1 << bitwidth) if bitwidth <= 8 else 256
    if bitwidth <= 8:
        qmin, qmax = get_quantized_range(bitwidth)
        bins = np.arange(qmin, qmax + 2)
        align = 'left'
    else:
        bins = 256
        align = 'mid'
    fig, axes = plt.subplots(3,3, figsize=(10, 6))
    axes = axes.ravel()
    plot_index = 0
    for name, param in model.named_parameters():
        if param.dim() > 1:
            ax = axes[plot_index]
            ax.hist(param.detach().view(-1).cpu(), bins=bins, density=True,
                    align=align, color = 'blue', alpha = 0.5,
                    edgecolor='black' if bitwidth <= 4 else None)
            if bitwidth <= 4:
                quantized_min, quantized_max = get_quantized_range(bitwidth)
                ax.set_xticks(np.arange(start=quantized_min,␣
 ↪stop=quantized_max+1))
            ax.set_xlabel(name)
            ax.set_ylabel('density')
            plot_index += 1
    fig.suptitle(f'Histogram of Weights (bitwidth={bitwidth} bits)')
    fig.tight_layout()
    fig.subplots_adjust(top=0.925)
    plt.show()

recover_model()
plot_weight_distribution(model)
```

As we can see from the histograms above, the distribution of weight values are nearly symmetric about 0 (except for the classifier in this case). Therefore, we usually make zero point $Z = 0$ when quantizating the weights.

From $r = S(q - Z)$, we have

$$r_{\max} = S \cdot q_{\max}$$

and then

$$S = r_{\max}/q_{\max}$$

We directly use the maximum magnitude of weight values as $r_{\max}$.

[ ]:
```
def get_quantization_scale_for_weight(weight, bitwidth):
    """
    get quantization scale for single tensor of weight
```

```
    :param weight: [torch.(cuda.)Tensor] floating weight to be quantized
    :param bitwidth: [integer] quantization bit width
    :return:
        [floating scalar] scale
    """
    # we just assume values in weight are symmetric
    # we also always make zero_point 0 for weight
    fp_max = max(weight.abs().max().item(), 5e-7)
    _, quantized_max = get_quantized_range(bitwidth)
    return fp_max / quantized_max
```

5.4.1 Per-channel Linear Quantization (OPTIONAL)

Recall that for 2D convolution, the weight tensor is a 4-D tensor in the shape of (num_output_channels, num_input_channels, kernel_height, kernel_width).

Intensive experiments show that using the different scaling factors S and zero points Z for different output channels will perform better. Therefore, we have to determine scaling factor S and zero point Z for the subtensor of each output channel independently.

[ ]:
```
def linear_quantize_weight_per_channel(tensor, bitwidth):
    """
    linear quantization for weight tensor
        using different scales and zero_points for different output channels
    :param tensor: [torch.(cuda.)Tensor] floating weight to be quantized
    :param bitwidth: [int] quantization bit width
    :return:
        [torch.(cuda.)Tensor] quantized tensor
        [torch.(cuda.)Tensor] scale tensor
        [int] zero point (which is always 0)
    """
    dim_output_channels = 0
    num_output_channels = tensor.shape[dim_output_channels]
    scale = torch.zeros(num_output_channels, device=tensor.device)
    for oc in range(num_output_channels):
        _subtensor = tensor.select(dim_output_channels, oc)
        _scale = get_quantization_scale_for_weight(_subtensor, bitwidth)
        scale[oc] = _scale
    scale_shape = [1] * tensor.dim()
    scale_shape[dim_output_channels] = -1
    scale = scale.view(scale_shape)
    quantized_tensor = linear_quantize(tensor, bitwidth, scale, zero_point=0)
    return quantized_tensor, scale, 0
```

5.4.2 A Quick Peek at Linear Quantization on Weights (OPTIONAL)

Now let's have a peek on the weight distribution and model size when applying linear quantization on weights with different bitwidths.

[ ]:
```
@torch.no_grad()
def peek_linear_quantization():
    for bitwidth in [4, 2]:
        for name, param in model.named_parameters():
            if param.dim() > 1:
                quantized_param, scale, zero_point = \
                    linear_quantize_weight_per_channel(param, bitwidth)
                param.copy_(quantized_param)
        plot_weight_distribution(model, bitwidth)
        recover_model()

peek_linear_quantization()
```

5.5 Quantized Inference (OPTIONAL)

After quantization, the inference of convolution and fully-connected layers also change.

Recall that $r = S(q - Z)$, and we have

$$r_{\text{input}} = S_{\text{input}}(q_{\text{input}} - Z_{\text{input}})$$

$$r_{\text{weight}} = S_{\text{weight}}(q_{\text{weight}} - Z_{\text{weight}})$$

$$r_{\text{bias}} = S_{\text{bias}}(q_{\text{bias}} - Z_{\text{bias}})$$

Since $Z_{\text{weight}} = 0$, $r_{\text{weight}} = S_{\text{weight}} q_{\text{weight}}$.

The floating point convolution can be written as,

$$\begin{aligned} r_{\text{output}} &= \text{CONV}[r_{\text{input}}, r_{\text{weight}}] + r_{\text{bias}} \\ &= \text{CONV}[S_{\text{input}}(q_{\text{input}} - Z_{\text{input}}), S_{\text{weight}} q_{\text{weight}}] + S_{\text{bias}}(q_{\text{bias}} - Z_{\text{bias}}) \\ &= \text{CONV}[q_{\text{input}} - Z_{\text{input}}, q_{\text{weight}}] \cdot (S_{\text{input}} \cdot S_{\text{weight}}) + S_{\text{bias}}(q_{\text{bias}} - Z_{\text{bias}}) \end{aligned}$$

To further simplify the computation, we could let

$$Z_{\text{bias}} = 0$$

$$S_{\text{bias}} = S_{\text{input}} \cdot S_{\text{weight}}$$

so that

$$\begin{aligned} r_{\text{output}} &= (\text{CONV}[q_{\text{input}} - Z_{\text{input}}, q_{\text{weight}}] + q_{\text{bias}}) \cdot (S_{\text{input}} \cdot S_{\text{weight}}) = \\ &(\text{CONV}[q_{\text{input}}, q_{\text{weight}}] - \text{CONV}[Z_{\text{input}}, q_{\text{weight}}] + q_{\text{bias}}) \cdot (S_{\text{input}} S_{\text{weight}}) \end{aligned}$$

Since $> r_{\text{output}} = S_{\text{output}}(q_{\text{output}} - Z_{\text{output}})$

we have $> S_{\text{output}}(q_{\text{output}} - Z_{\text{output}}) = (\text{CONV}[q_{\text{input}}, q_{\text{weight}}] - \text{CONV}[Z_{\text{input}}, q_{\text{weight}}] + q_{\text{bias}}) \cdot (S_{\text{input}} S_{\text{weight}})$

and thus $> q_{\text{output}} = (\text{CONV}[q_{\text{input}}, q_{\text{weight}}] - \text{CONV}[Z_{\text{input}}, q_{\text{weight}}] + q_{\text{bias}}) \cdot (S_{\text{input}} S_{\text{weight}} / S_{\text{output}}) + Z_{\text{output}}$

Since Z_{input}, q_{weight}, q_{bias} are determined before inference, let

$$Q_{\text{bias}} = q_{\text{bias}} - \text{CONV}[Z_{\text{input}}, q_{\text{weight}}]$$

we have

$$q_{\text{output}} = (\text{CONV}[q_{\text{input}}, q_{\text{weight}}] + Q_{\text{bias}}) \cdot (S_{\text{input}} S_{\text{weight}} / S_{\text{output}}) + Z_{\text{output}}$$

Similarily, for fully-connected layer, we have

$$q_{\text{output}} = (\text{Linear}[q_{\text{input}}, q_{\text{weight}}] + Q_{\text{bias}}) \cdot (S_{\text{input}} \cdot S_{\text{weight}} / S_{\text{output}}) + Z_{\text{output}}$$

where

$$Q_{\text{bias}} = q_{\text{bias}} - \text{Linear}[Z_{\text{input}}, q_{\text{weight}}]$$

5.5.1 Question 6 (OPTIONAL)

Please complete the following function for linear quantizing the bias.

Hint:

From the above deduction, we know that

$$Z_{\text{bias}} = 0$$

$$S_{\text{bias}} = S_{\text{input}} \cdot S_{\text{weight}}$$

[ ]:
```
def linear_quantize_bias_per_output_channel(bias, weight_scale, input_scale):
    """
    linear quantization for single bias tensor
        quantized_bias = fp_bias / bias_scale
    :param bias: [torch.FloatTensor] bias weight to be quantized
    :param weight_scale: [float or torch.FloatTensor] weight scale tensor
    :param input_scale: [float] input scale
    :return:
        [torch.IntTensor] quantized bias tensor
    """
    assert(bias.dim() == 1)
    assert(bias.dtype == torch.float)
    assert(isinstance(input_scale, float))
    if isinstance(weight_scale, torch.Tensor):
        assert(weight_scale.dtype == torch.float)
        weight_scale = weight_scale.view(-1)
        assert(bias.numel() == weight_scale.numel())

    ################ YOUR CODE STARTS HERE ################
    # hint: one line of code
    bias_scale = 0
    ################ YOUR CODE ENDS HERE #################

    quantized_bias = linear_quantize(bias, 32, bias_scale,
                                     zero_point=0, dtype=torch.int32)
    return quantized_bias, bias_scale, 0
```

5.5.2 Quantized Fully-Connected Layer (OPTIONAL)

For quantized fully-connected layer, we first precompute Q_{bias}. Recall that $Q_{\text{bias}} = q_{\text{bias}} - \text{Linear}[Z_{\text{input}}, q_{\text{weight}}]$.

[ ]:
```
def shift_quantized_linear_bias(quantized_bias, quantized_weight,␣
↪input_zero_point):
    """
    shift quantized bias to incorporate input_zero_point for nn.Linear
        shifted_quantized_bias = quantized_bias - Linear(input_zero_point,␣
↪quantized_weight)
    :param quantized_bias: [torch.IntTensor] quantized bias (torch.int32)
    :param quantized_weight: [torch.CharTensor] quantized weight (torch.int8)
    :param input_zero_point: [int] input zero point
    :return:
        [torch.IntTensor] shifted quantized bias tensor
    """
    assert(quantized_bias.dtype == torch.int32)
    assert(isinstance(input_zero_point, int))
    return quantized_bias - quantized_weight.sum(1).to(torch.int32) *␣
↪input_zero_point
```

Question 7 (OPTIONAL) Please complete the following quantized fully-connected layer inference function.

Hint:

$$q_{\text{output}} = (\text{Linear}[q_{\text{input}}, q_{\text{weight}}] + Q_{\text{bias}}) \cdot (S_{\text{input}} S_{\text{weight}} / S_{\text{output}}) + Z_{\text{output}}$$

[ ]:
```
def quantized_linear(input, weight, bias, feature_bitwidth, weight_bitwidth,
                     input_zero_point, output_zero_point,
                     input_scale, weight_scale, output_scale):
    """
    quantized fully-connected layer
    :param input: [torch.CharTensor] quantized input (torch.int8)
    :param weight: [torch.CharTensor] quantized weight (torch.int8)
    :param bias: [torch.IntTensor] shifted quantized bias or None (torch.int32)
    :param feature_bitwidth: [int] quantization bit width of input and output
    :param weight_bitwidth: [int] quantization bit width of weight
    :param input_zero_point: [int] input zero point
    :param output_zero_point: [int] output zero point
    :param input_scale: [float] input feature scale
    :param weight_scale: [torch.FloatTensor] weight per-channel scale
    :param output_scale: [float] output feature scale
    :return:
        [torch.CharIntTensor] quantized output feature (torch.int8)
    """
    assert(input.dtype == torch.int8)
    assert(weight.dtype == input.dtype)
```

```
    assert(bias is None or bias.dtype == torch.int32)
    assert(isinstance(input_zero_point, int))
    assert(isinstance(output_zero_point, int))
    assert(isinstance(input_scale, float))
    assert(isinstance(output_scale, float))
    assert(weight_scale.dtype == torch.float)

    # Step 1: integer-based fully-connected (8-bit multiplication with 32-bit␣
↪accumulation)
    if 'cpu' in input.device.type:
        # use 32-b MAC for simplicity
        output = torch.nn.functional.linear(input.to(torch.int32), weight.
↪to(torch.int32), bias)
    else:
        # current version pytorch does not yet support integer-based linear()␣
↪on GPUs
        output = torch.nn.functional.linear(input.float(), weight.float(), bias.
↪float())

    ################ YOUR CODE STARTS HERE ################
    # Step 2: scale the output
    #         hint: 1. scales are floating numbers, we need to convert output␣
↪to float as well
    #               2. the shape of weight scale is [oc, 1, 1, 1] while the␣
↪shape of output is [batch_size, oc]
    output = 0

    # Step 3: shift output by output_zero_point
    #         hint: one line of code
    output = 0
    ################ YOUR CODE ENDS HERE #################

    # Make sure all value lies in the bitwidth-bit range
    output = output.round().clamp(*get_quantized_range(feature_bitwidth)).
↪to(torch.int8)
    return output
```

Let's verify the functionality of defined quantized fully connected layer.

[ ]:
```
test_quantized_fc()
```

5.5.3 Quantized Convolution (OPTIONAL)

For quantized convolution layer, we first precompute Q_{bias}. Recall that $Q_{\text{bias}} = q_{\text{bias}} - \text{CONV}[Z_{\text{input}}, q_{\text{weight}}]$.

```
[ ]: def shift_quantized_conv2d_bias(quantized_bias, quantized_weight,␣
     ↪input_zero_point):
        """
        shift quantized bias to incorporate input_zero_point for nn.Conv2d
            shifted_quantized_bias = quantized_bias - Conv(input_zero_point,␣
     ↪quantized_weight)
        :param quantized_bias: [torch.IntTensor] quantized bias (torch.int32)
        :param quantized_weight: [torch.CharTensor] quantized weight (torch.int8)
        :param input_zero_point: [int] input zero point
        :return:
            [torch.IntTensor] shifted quantized bias tensor
        """
        assert(quantized_bias.dtype == torch.int32)
        assert(isinstance(input_zero_point, int))
        return quantized_bias - quantized_weight.sum((1,2,3)).to(torch.int32) *␣
     ↪input_zero_point
```

Question 8 (OPTIONAL) Please complete the following quantized convolution function.

Hint: $> q_{\text{output}} = (\text{CONV}[q_{\text{input}}, q_{\text{weight}}] + Q_{\text{bias}}) \cdot (S_{\text{input}} S_{\text{weight}} / S_{\text{output}}) + Z_{\text{output}}$

```
[ ]: def quantized_conv2d(input, weight, bias, feature_bitwidth, weight_bitwidth,
                          input_zero_point, output_zero_point,
                          input_scale, weight_scale, output_scale,
                          stride, padding, dilation, groups):
        """
        quantized 2d convolution
        :param input: [torch.CharTensor] quantized input (torch.int8)
        :param weight: [torch.CharTensor] quantized weight (torch.int8)
        :param bias: [torch.IntTensor] shifted quantized bias or None (torch.int32)
        :param feature_bitwidth: [int] quantization bit width of input and output
        :param weight_bitwidth: [int] quantization bit width of weight
        :param input_zero_point: [int] input zero point
        :param output_zero_point: [int] output zero point
        :param input_scale: [float] input feature scale
        :param weight_scale: [torch.FloatTensor] weight per-channel scale
        :param output_scale: [float] output feature scale
        :return:
            [torch.(cuda.)CharTensor] quantized output feature
        """
        assert(len(padding) == 4)
        assert(input.dtype == torch.int8)
        assert(weight.dtype == input.dtype)
        assert(bias is None or bias.dtype == torch.int32)
        assert(isinstance(input_zero_point, int))
        assert(isinstance(output_zero_point, int))
        assert(isinstance(input_scale, float))
```

```
    assert(isinstance(output_scale, float))
    assert(weight_scale.dtype == torch.float)

    # Step 1: calculate integer-based 2d convolution (8-bit multiplication with␣
↪32-bit accumulation)
    input = torch.nn.functional.pad(input, padding, 'constant',␣
↪input_zero_point)
    if 'cpu' in input.device.type:
        # use 32-b MAC for simplicity
        output = torch.nn.functional.conv2d(input.to(torch.int32), weight.
↪to(torch.int32), None, stride, 0, dilation, groups)
    else:
        # current version pytorch does not yet support integer-based conv2d()␣
↪on GPUs
        output = torch.nn.functional.conv2d(input.float(), weight.float(),␣
↪None, stride, 0, dilation, groups)
        output = output.round().to(torch.int32)
    if bias is not None:
        output = output + bias.view(1, -1, 1, 1)

    ################ YOUR CODE STARTS HERE ################
    # hint: this code block should be the very similar to quantized_linear()

    # Step 2: scale the output
    #         hint: 1. scales are floating numbers, we need to convert output␣
↪to float as well
    #               2. the shape of weight scale is [oc, 1, 1, 1] while the␣
↪shape of output is [batch_size, oc, height, width]
    output = 0

    # Step 3: shift output by output_zero_point
    #         hint: one line of code
    output = 0
    ################ YOUR CODE ENDS HERE #################

    # Make sure all value lies in the bitwidth-bit range
    output = output.round().clamp(*get_quantized_range(feature_bitwidth)).
↪to(torch.int8)
    return output
```

5.6 Question 9 (OPTIONAL)

Finally, we are putting everything together and perform post-training `int8` quantization for the model. We will convert the convolutional and linear layers in the model to a quantized version one-by-one.

1. Firstly, we will fuse a BatchNorm layer into its previous convolutional layer, which is a

standard practice before quantization. Fusing batchnorm reduces the extra multiplication during inference.

We will also verify that the fused model `model_fused` has the same accuracy as the original model (BN fusion is an equivalent transform that does not change network functionality).

[ ]:
```
def fuse_conv_bn(conv, bn):
    # modified from https://mmcv.readthedocs.io/en/latest/_modules/mmcv/cnn/
 ↪utils/fuse_conv_bn.html
    assert conv.bias is None

    factor = bn.weight.data / torch.sqrt(bn.running_var.data + bn.eps)
    conv.weight.data = conv.weight.data * factor.reshape(-1, 1, 1, 1)
    conv.bias = nn.Parameter(- bn.running_mean.data * factor + bn.bias.data)

    return conv

print('Before conv-bn fusion: backbone length', len(model.backbone))
#  fuse the batchnorm into conv layers
recover_model()
model_fused = copy.deepcopy(model)
fused_backbone = []
ptr = 0
while ptr < len(model_fused.backbone):
    if isinstance(model_fused.backbone[ptr], nn.Conv2d) and \
        isinstance(model_fused.backbone[ptr + 1], nn.BatchNorm2d):
        fused_backbone.append(fuse_conv_bn(
            model_fused.backbone[ptr], model_fused.backbone[ptr+ 1]))
        ptr += 2
    else:
        fused_backbone.append(model_fused.backbone[ptr])
        ptr += 1
model_fused.backbone = nn.Sequential(*fused_backbone)

print('After conv-bn fusion: backbone length', len(model_fused.backbone))
# sanity check, no BN anymore
for m in model_fused.modules():
    assert not isinstance(m, nn.BatchNorm2d)

#  the accuracy will remain the same after fusion
fused_acc = evaluate(model_fused, dataloader['test'])
print(f'Accuracy of the fused model={fused_acc:.2f}%')
```

2. We will run the model with some sample data to get the range of each feature map, so that we can get the range of the feature maps and compute their corresponding scaling factors and zero points.

[ ]:
```
# add hook to record the min max value of the activation
input_activation = {}
output_activation = {}

def add_range_recoder_hook(model):
    import functools
    def _record_range(self, x, y, module_name):
        x = x[0]
        input_activation[module_name] = x.detach()
        output_activation[module_name] = y.detach()

    all_hooks = []
    for name, m in model.named_modules():
        if isinstance(m, (nn.Conv2d, nn.Linear, nn.ReLU)):
            all_hooks.append(m.register_forward_hook(
                functools.partial(_record_range, module_name=name)))
    return all_hooks

hooks = add_range_recoder_hook(model_fused)
sample_data = iter(dataloader['train']).__next__()[0]
model_fused(sample_data.cuda())

# remove hooks
for h in hooks:
    h.remove()
```

3. Finally, let's do model quantization. We will convert the model in the following mapping

```
nn.Conv2d: QuantizedConv2d,
nn.Linear: QuantizedLinear,
# the following twos are just wrappers, as current
# torch modules do not support int8 data format;
# we will temporarily convert them to fp32 for computation
nn.MaxPool2d: QuantizedMaxPool2d,
nn.AvgPool2d: QuantizedAvgPool2d,
```

[ ]:
```
class QuantizedConv2d(nn.Module):
    def __init__(self, weight, bias,
                 input_zero_point, output_zero_point,
                 input_scale, weight_scale, output_scale,
                 stride, padding, dilation, groups,
                 feature_bitwidth=8, weight_bitwidth=8):
        super().__init__()
        # current version Pytorch does not support IntTensor as nn.Parameter
        self.register_buffer('weight', weight)
        self.register_buffer('bias', bias)

        self.input_zero_point = input_zero_point
```

```
        self.output_zero_point = output_zero_point

        self.input_scale = input_scale
        self.register_buffer('weight_scale', weight_scale)
        self.output_scale = output_scale

        self.stride = stride
        self.padding = (padding[1], padding[1], padding[0], padding[0])
        self.dilation = dilation
        self.groups = groups

        self.feature_bitwidth = feature_bitwidth
        self.weight_bitwidth = weight_bitwidth


    def forward(self, x):
        return quantized_conv2d(
            x, self.weight, self.bias,
            self.feature_bitwidth, self.weight_bitwidth,
            self.input_zero_point, self.output_zero_point,
            self.input_scale, self.weight_scale, self.output_scale,
            self.stride, self.padding, self.dilation, self.groups
            )

class QuantizedLinear(nn.Module):
    def __init__(self, weight, bias,
                 input_zero_point, output_zero_point,
                 input_scale, weight_scale, output_scale,
                 feature_bitwidth=8, weight_bitwidth=8):
        super().__init__()
        # current version Pytorch does not support IntTensor as nn.Parameter
        self.register_buffer('weight', weight)
        self.register_buffer('bias', bias)

        self.input_zero_point = input_zero_point
        self.output_zero_point = output_zero_point

        self.input_scale = input_scale
        self.register_buffer('weight_scale', weight_scale)
        self.output_scale = output_scale

        self.feature_bitwidth = feature_bitwidth
        self.weight_bitwidth = weight_bitwidth

    def forward(self, x):
        return quantized_linear(
            x, self.weight, self.bias,
```

```
            self.feature_bitwidth, self.weight_bitwidth,
            self.input_zero_point, self.output_zero_point,
            self.input_scale, self.weight_scale, self.output_scale
            )

class QuantizedMaxPool2d(nn.MaxPool2d):
    def forward(self, x):
        # current version PyTorch does not support integer-based MaxPool
        return super().forward(x.float()).to(torch.int8)

class QuantizedAvgPool2d(nn.AvgPool2d):
    def forward(self, x):
        # current version PyTorch does not support integer-based AvgPool
        return super().forward(x.float()).to(torch.int8)

# we use int8 quantization, which is quite popular
feature_bitwidth = weight_bitwidth = 8
quantized_model = copy.deepcopy(model_fused)
quantized_backbone = []
ptr = 0
while ptr < len(quantized_model.backbone):
    if isinstance(quantized_model.backbone[ptr], nn.Conv2d) and \
        isinstance(quantized_model.backbone[ptr + 1], nn.ReLU):
        conv = quantized_model.backbone[ptr]
        conv_name = f'backbone.{ptr}'
        relu = quantized_model.backbone[ptr + 1]
        relu_name = f'backbone.{ptr + 1}'

        input_scale, input_zero_point = \
            get_quantization_scale_and_zero_point(
                input_activation[conv_name], feature_bitwidth)

        output_scale, output_zero_point = \
            get_quantization_scale_and_zero_point(
                output_activation[relu_name], feature_bitwidth)

        quantized_weight, weight_scale, weight_zero_point = \
            linear_quantize_weight_per_channel(conv.weight.data,␣
 ↪weight_bitwidth)
        quantized_bias, bias_scale, bias_zero_point = \
            linear_quantize_bias_per_output_channel(
                conv.bias.data, weight_scale, input_scale)
        shifted_quantized_bias = \
            shift_quantized_conv2d_bias(quantized_bias, quantized_weight,
                                        input_zero_point)

        quantized_conv = QuantizedConv2d(
```

```
            quantized_weight, shifted_quantized_bias,
            input_zero_point, output_zero_point,
            input_scale, weight_scale, output_scale,
            conv.stride, conv.padding, conv.dilation, conv.groups,
            feature_bitwidth=feature_bitwidth, weight_bitwidth=weight_bitwidth
        )

        quantized_backbone.append(quantized_conv)
        ptr += 2
    elif isinstance(quantized_model.backbone[ptr], nn.MaxPool2d):
        quantized_backbone.append(QuantizedMaxPool2d(
            kernel_size=quantized_model.backbone[ptr].kernel_size,
            stride=quantized_model.backbone[ptr].stride
            ))
        ptr += 1
    elif isinstance(quantized_model.backbone[ptr], nn.AvgPool2d):
        quantized_backbone.append(QuantizedAvgPool2d(
            kernel_size=quantized_model.backbone[ptr].kernel_size,
            stride=quantized_model.backbone[ptr].stride
            ))
        ptr += 1
    else:
        raise NotImplementedError(type(quantized_model.backbone[ptr]))  #␣
 ↪should not happen
quantized_model.backbone = nn.Sequential(*quantized_backbone)

# finally, quantized the classifier
fc_name = 'classifier'
fc = model.classifier
input_scale, input_zero_point = \
    get_quantization_scale_and_zero_point(
        input_activation[fc_name], feature_bitwidth)

output_scale, output_zero_point = \
    get_quantization_scale_and_zero_point(
        output_activation[fc_name], feature_bitwidth)

quantized_weight, weight_scale, weight_zero_point = \
    linear_quantize_weight_per_channel(fc.weight.data, weight_bitwidth)
quantized_bias, bias_scale, bias_zero_point = \
    linear_quantize_bias_per_output_channel(
        fc.bias.data, weight_scale, input_scale)
shifted_quantized_bias = \
    shift_quantized_linear_bias(quantized_bias, quantized_weight,
                                input_zero_point)

quantized_model.classifier = QuantizedLinear(
```

```
    quantized_weight, shifted_quantized_bias,
    input_zero_point, output_zero_point,
    input_scale, weight_scale, output_scale,
    feature_bitwidth=feature_bitwidth, weight_bitwidth=weight_bitwidth
)
```

The quantization process is done! Let's print and visualize the model architecture and also verify the accuracy of the quantized model.

5.6.1 Question 9.1 (OPTIONAL)

To run the quantized model, we need an extra preprocessing to map the input data from range (0, 1) into `int8` range of (-128, 127). Fill in the code below to finish the extra preprocessing.

Hint: you should find that the quantized model has roughly the same accuracy as the `fp32` counterpart.

[ ]:
```
print(quantized_model)

def extra_preprocess(x):
    # hint: you need to convert the original fp32 input of range (0, 1)
    #  into int8 format of range (-128, 127)
    ############### YOUR CODE STARTS HERE ###############
    return 0.clamp(-128, 127).to(torch.int8)
    ############### YOUR CODE ENDS HERE #################

int8_model_accuracy = evaluate(quantized_model, dataloader['test'],
                               extra_preprocess=[extra_preprocess])
print(f"int8 model has accuracy={int8_model_accuracy:.2f}%")
```

5.7 Question 9.2 (OPTIONAL)

Explain why there is no ReLU layer in the linear quantized model.

Your Answer:

6 Question 10 (OPTIONAL)

Please compare the advantages and disadvantages of k-means-based quantization and linear quantization. You can discuss from the perspective of accuracy, latency, hardware support, etc.

Your Answer: