In [1]:

```python
import os, math, time, copy
from collections import OrderedDict

import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms

from models import VGG16_quant
from models.quant_layer import QuantConv2d, act_quantization

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.backends.cudnn.benchmark = True
print("=> Using device:", device)

# ----------------- CIFAR-10 ----------------
BATCH_SIZE = 128

normalize = transforms.Normalize(
    mean=[0.491, 0.482, 0.447],
    std =[0.247, 0.243, 0.262]
)

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True,
    transform=transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
    ])
)
testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ])
```

```
)

trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2, pin_memory=True
)
testloader  = torch.utils.data.DataLoader(
    testset,  batch_size=BATCH_SIZE, shuffle=False, num_workers=2, pin_memory=True
)

dataiter = iter(testloader)
images, labels = next(dataiter)
images, labels = images.to(device), labels.to(device)
print("=> One batch from testloader:", images.shape, labels.shape)

@torch.no_grad()
def evaluate(model, loader, criterion):
    model.eval()
    total, correct = 0, 0
    loss_sum = 0.0
    for x, y in loader:
        x, y = x.to(device), y.to(device)
        out = model(x)
        loss = criterion(out, y)
        loss_sum += loss.item() * x.size(0)
        _, pred = out.max(1)
        correct += pred.eq(y).sum().item()
        total += y.size(0)

    acc = 100.0 * correct / total
    return acc, loss_sum / total
```

```
=> Using device: cuda
=> One batch from testloader: torch.Size([128, 3, 32, 32]) torch.Size([128])
```

In [2]:

```
model = VGG16_quant().to(device)
model.eval()

ckpt_path = "./results/VGG16_quant/model_best.pth.tar"
print("=> Loading checkpoint:", ckpt_path)

ckpt = torch.load(ckpt_path, map_location=device)
```

```
state_dict = ckpt["state_dict"] if "state_dict" in ckpt else ckpt

new_state_dict = OrderedDict()
for k, v in state_dict.items():
    if k.startswith("module."):
        new_state_dict[k[7:]] = v
    else:
        new_state_dict[k] = v

model.load_state_dict(new_state_dict, strict=True)

criterion = nn.CrossEntropyLoss().to(device)

acc_full, test_loss = evaluate(model, testloader, criterion)
print(f"\n[Checkpoint model] Test Accuracy: {acc_full:.2f}% | Test Loss: {test_loss:.4f}\n")
```

```
=> Loading checkpoint: ./results/VGG16_quant/model_best.pth.tar

[Checkpoint model] Test Accuracy: 90.68% | Test Loss: 0.2997
```

In [3]:

```
qconvs = [m for m in model.modules() if isinstance(m, QuantConv2d)]
print("=> Number of QuantConv2d layers:", len(qconvs))

for idx, m in enumerate(qconvs):
    print(f"  Layer {idx}: in={m.in_channels}, out={m.out_channels}, k={m.kernel_size}")

for m in qconvs:
    m.act_bit = 4
```

```
=> Number of QuantConv2d layers: 13
  Layer 0: in=3, out=64, k=(3, 3)
  Layer 1: in=64, out=64, k=(3, 3)
  Layer 2: in=64, out=128, k=(3, 3)
  Layer 3: in=128, out=128, k=(3, 3)
  Layer 4: in=128, out=256, k=(3, 3)
  Layer 5: in=256, out=256, k=(3, 3)
  Layer 6: in=256, out=256, k=(3, 3)
  Layer 7: in=256, out=512, k=(3, 3)
  Layer 8: in=512, out=512, k=(3, 3)
  Layer 9: in=512, out=512, k=(3, 3)
  Layer 10: in=512, out=512, k=(3, 3)
  Layer 11: in=512, out=512, k=(3, 3)
  Layer 12: in=512, out=512, k=(3, 3)
```

In [4]:

```python
def set_all_act_bits(model, bit: int):
    """
    Change the activation bit of all QuantConv2d quantizations in the model to the same value (2 or 4).
    This only affects activation quantization, not weight quantization.
    """
    for m in model.modules():
        if isinstance(m, QuantConv2d):
            m.act_bit = bit
            m.act_alq = act_quantization(bit)  # act_quantization(b) 返回一个新的量化函数


def set_layer_act_bit(model, layer_idx: int, bit: int):
    """
    Only modify the activation bit of the layer_idx QuantConv2d.
    """
    qconvs_local = [m for m in model.modules() if isinstance(m, QuantConv2d)]
    m = qconvs_local[layer_idx]
    m.act_bit = bit
    m.act_alq = act_quantization(bit)


def eval_acc(model):
    acc, _ = evaluate(model, testloader, criterion)
    return acc
```

In [5]:
```
set_all_act_bits(model, 4)

acc_all4 = eval_acc(model)
print(f"[All-4bit activation] Test Accuracy: {acc_all4:.2f}%")
```

```
[All-4bit activation] Test Accuracy: 90.68%
```

In [6]:
```
num_layers = len(qconvs)
sensitivity_results = []

print("=> Running per-layer sensitivity analysis ...")

for idx in range(num_layers):
    set_all_act_bits(model, 4)

    set_layer_act_bit(model, idx, 2)

    acc = eval_acc(model)
    acc_drop = (acc_all4 - acc) / 100.0

    sensitivity_results.append({
        "layer": idx,
        "acc": acc,
        "acc_drop": acc_all4 - acc,
    })
    print(f"  [Layer {idx:2d} -> 2bit] acc={acc:.2f}%, drop={acc_all4 - acc:.2f}%")

sensitivity_sorted = sorted(sensitivity_results, key=lambda x: x["acc_drop"])

print("\n=> Per-layer sensitivity sorted by accuracy drop:")
for r in sensitivity_sorted:
    print(f"  Layer {r['layer']:2d}: acc={r['acc']:.2f}%, drop={r['acc_drop']:.2f}%")
```

```
=> Running per-layer sensitivity analysis ...
  [Layer  0 -> 2bit] acc=81.18%, drop=9.50%
  [Layer  1 -> 2bit] acc=89.53%, drop=1.15%
  [Layer  2 -> 2bit] acc=89.82%, drop=0.86%
  [Layer  3 -> 2bit] acc=89.97%, drop=0.71%
  [Layer  4 -> 2bit] acc=89.68%, drop=1.00%
  [Layer  5 -> 2bit] acc=89.31%, drop=1.37%
  [Layer  6 -> 2bit] acc=90.14%, drop=0.54%
  [Layer  7 -> 2bit] acc=90.04%, drop=0.64%
  [Layer  8 -> 2bit] acc=90.58%, drop=0.10%
  [Layer  9 -> 2bit] acc=90.74%, drop=-0.06%
  [Layer 10 -> 2bit] acc=90.52%, drop=0.16%
  [Layer 11 -> 2bit] acc=90.56%, drop=0.12%
  [Layer 12 -> 2bit] acc=90.56%, drop=0.12%

=> Per-layer sensitivity sorted by accuracy drop:
  Layer  9: acc=90.74%, drop=-0.06%
  Layer  8: acc=90.58%, drop=0.10%
  Layer 11: acc=90.56%, drop=0.12%
  Layer 12: acc=90.56%, drop=0.12%
  Layer 10: acc=90.52%, drop=0.16%
  Layer  6: acc=90.14%, drop=0.54%
  Layer  7: acc=90.04%, drop=0.64%
  Layer  3: acc=89.97%, drop=0.71%
  Layer  2: acc=89.82%, drop=0.86%
  Layer  4: acc=89.68%, drop=1.00%
  Layer  1: acc=89.53%, drop=1.15%
  Layer  5: acc=89.31%, drop=1.37%
  Layer  0: acc=81.18%, drop=9.50%
```

In [10]:

```python
# maximum allowed accuracy drop
epsilon = 1.0

num_layers = len(qconvs)
precisions = [4] * num_layers

set_all_act_bits(model, 4)
current_acc = acc_all4

print(f"=> Start greedy search from all-4bit baseline: acc={acc_all4:.2f}%, epsilon={epsilon:.2f}%\n")
```

```python
for r in sensitivity_sorted:
    idx = r["layer"]

    set_layer_act_bit(model, idx, 2)
    precisions[idx] = 2

    acc = eval_acc(model)
    drop = acc_all4 - acc

    if drop <= epsilon:
        current_acc = acc
        print(f"  ACCEPT layer {idx:2d} -> 2bit | acc={acc:.2f}%, drop={drop:.2f}%")
    else:
        set_layer_act_bit(model, idx, 4)
        precisions[idx] = 4
        print(f"  REJECT layer {idx:2d} -> 2bit | acc={acc:.2f}%, drop={drop:.2f}%  (> {epsilon:.2f}%)")

print("\n==================== Final Mixed-Precision Config ====================")
print("Per-layer activation bits (per QuantConv2d):")
print(precisions)

num_2bit = sum(1 for b in precisions if b == 2)
print(f"\nTotal conv layers = {num_layers}")
print(f"2bit layers       = {num_2bit} ({num_2bit/num_layers*100:.1f}%)")
print(f"\nFinal mixed-precision accuracy     : {current_acc:.2f}%")
print(f"Accuracy drop vs all-4bit baseline : {acc_all4 - current_acc:.2f}%")
```

```
=> Start greedy search from all-4bit baseline: acc=90.68%, epsilon=1.00%

  ACCEPT layer  9 -> 2bit | acc=90.74%, drop=-0.06%
  ACCEPT layer  8 -> 2bit | acc=90.62%, drop=0.06%
  ACCEPT layer 11 -> 2bit | acc=90.30%, drop=0.38%
  ACCEPT layer 12 -> 2bit | acc=90.22%, drop=0.46%
  ACCEPT layer 10 -> 2bit | acc=90.35%, drop=0.33%
  REJECT layer  6 -> 2bit | acc=89.32%, drop=1.36%  (> 1.00%)
  REJECT layer  7 -> 2bit | acc=89.41%, drop=1.27%  (> 1.00%)
  REJECT layer  3 -> 2bit | acc=89.65%, drop=1.03%  (> 1.00%)
  REJECT layer  2 -> 2bit | acc=89.21%, drop=1.47%  (> 1.00%)
  REJECT layer  4 -> 2bit | acc=89.00%, drop=1.68%  (> 1.00%)
  REJECT layer  1 -> 2bit | acc=89.06%, drop=1.62%  (> 1.00%)
  REJECT layer  5 -> 2bit | acc=88.88%, drop=1.80%  (> 1.00%)
  REJECT layer  0 -> 2bit | acc=80.42%, drop=10.26%  (> 1.00%)

==================== Final Mixed-Precision Config ====================
Per-layer activation bits (per QuantConv2d):
[4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 2]

Total conv layers = 13
2bit layers       = 5 (38.5%)

Final mixed-precision accuracy     : 90.35%
Accuracy drop vs all-4bit baseline : 0.33%
```

Mixed-Precision Study (2-bit / 4-bit)

Here we perform a per-layer mixed-precision study on the `VGG16_quant` model (CIFAR-10), which:

- We keep weight quantization unchanged
- We only change activation bit-width of each `QuantConv2d` layer between 4-bit and 2-bit.
- The baseline is all layers with 4-bit activations.

Baseline (all 4-bit activations)

- Test accuracy: **90.68%**

Per-layer Sensitivity & Greedy Algorithm

1. Per-layer sensitivity
    - For each of the 13 `QuantConv2d` layers:
        - Set all layers to 4-bit.
        - Change only that layer to 2-bit.
        - Measure the new accuracy and record the accuracy drop vs. the all-4bit baseline.
    - This tells us how sensitive each layer is to reducing its activation precision from 4-bit to 2-bit.
2. Greedy mixed-precision search
    - Start from all-4bit configuration.
    - Sort layers by their individual accuracy drop (least sensitive → most sensitive).
    - Traverse this ordered list; for each candidate layer:
        - Tentatively switch it from 4-bit → 2-bit.
        - Re-evaluate accuracy with all previously accepted 2-bit layers included.
        - Accept the change if the total accuracy drop (vs. all-4bit) is ≤ ε (here ε = 1.0%); otherwise reject it.

Final Mixed-Precision Configuration

The final per-layer activation bit-widths (over 13 convolution layers) are:

```
[4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 2]
```

In [ ]:

In [ ]: