Activation_Aware_Pruning

December 11, 2025

[2]:
```python
import argparse
import os
import time
import shutil

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn


import torchvision
import torchvision.transforms as transforms

from models import *


global best_prec
use_gpu = torch.cuda.is_available()
print('=> Building model...')



batch_size = 128
model_name = "VGG16_quant"
model = VGG16_quant()

#print(model)

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


train_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
```

```
    download=True,
    transform=transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
    ]))
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=2)


test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ]))

testloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=2)


print_freq = 100 # every 100 batches, accuracy printed. Here, each batch␣
 ↪includes "batch_size" data points
# CIFAR10 has 50,000 training data, and 10,000 validation data.

def train(trainloader, model, criterion, optimizer, epoch):
    batch_time = AverageMeter()
    data_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    model.train()

    end = time.time()
    for i, (input, target) in enumerate(trainloader):
        # measure data loading time
        data_time.update(time.time() - end)

        input, target = input.cuda(), target.cuda()

        # compute output
        output = model(input)
        loss = criterion(output, target)
```

```
        # measure accuracy and record loss
        prec = accuracy(output, target)[0]
        losses.update(loss.item(), input.size(0))
        top1.update(prec.item(), input.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()


        if i % print_freq == 0:
            print('Epoch: [{0}][{1}/{2}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   epoch, i, len(trainloader), batch_time=batch_time,
                   data_time=data_time, loss=losses, top1=top1))



def validate(val_loader, model, criterion ):
    batch_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    # switch to evaluate mode
    model.eval()

    end = time.time()
    with torch.no_grad():
        for i, (input, target) in enumerate(val_loader):

            input, target = input.cuda(), target.cuda()

            # compute output
            output = model(input)
            loss = criterion(output, target)

            # measure accuracy and record loss
            prec = accuracy(output, target)[0]
            losses.update(loss.item(), input.size(0))
```

```
            top1.update(prec.item(), input.size(0))

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            if i % print_freq == 0:  # This line shows how frequently print out␣
↪the status. e.g., i%5 => every 5 batch, prints out
                print('Test: [{0}/{1}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   i, len(val_loader), batch_time=batch_time, loss=losses,
                   top1=top1))

    print(' * Prec {top1.avg:.3f}% '.format(top1=top1))
    return top1.avg


def accuracy(output, target, topk=(1,)):
    """Computes the precision@k for the specified values of k"""
    maxk = max(topk)
    batch_size = target.size(0)

    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res


class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
```

```
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count


def save_checkpoint(state, is_best, fdir):
    filepath = os.path.join(fdir, 'checkpoint.pth')
    torch.save(state, filepath)
    if is_best:
        shutil.copyfile(filepath, os.path.join(fdir, 'model_best.pth.tar'))


def adjust_learning_rate(optimizer, epoch):
    """For VGGNet, the lr starts from 0.01, and is divided by 10 at 50 and 100␣
 ↪epochs"""
    adjust_list = [80, 120]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1

#model = nn.DataParallel(model).cuda()
#all_params = checkpoint['state_dict']
#model.load_state_dict(all_params, strict=False)
#criterion = nn.CrossEntropyLoss().cuda()
#validate(testloader, model, criterion)
```

```
=> Building model…
Files already downloaded and verified
Files already downloaded and verified
```

[3]:
```
PATH = "result/VGG16_quant/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda")

model.cuda()
model.eval()

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in testloader:
        data, target = data.to(device), target.to(device) # loading to GPU
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
```

```
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(testloader.dataset)

print('\nTest set: Accuracy: {}/{} ({:.0f}%)\n'.format(
        correct, len(testloader.dataset),
        100. * correct / len(testloader.dataset)))
```

```
Test set: Accuracy: 9033/10000 (90%)
```

[4]:
```
# Additional imports for activation-aware pruning
import copy
import numpy as np
from collections import defaultdict
from models.quant_layer import QuantConv2d, weight_quantization,␣
 ↪act_quantization
```

[5]:
```
# Activation Collector - hooks into model to collect activation statistics
class ActivationCollector:
    def __init__(self):
        self.stats = defaultdict(list)
        self.hooks = []

    def register_hooks(self, model):
        for name, module in model.named_modules():
            if isinstance(module, QuantConv2d):
                hook = module.register_forward_pre_hook(
                    lambda m, inp, name=name: self._collect(name, inp[0])
                )
                self.hooks.append(hook)

    def _collect(self, name, activation):
        with torch.no_grad():
            mean_act = activation.abs().mean(dim=(0, 2, 3))
            self.stats[name].append(mean_act.cpu())

    def compute_stats(self):
        result = {}
        for name, act_list in self.stats.items():
            result[name] = torch.stack(act_list, dim=0).mean(dim=0)
        return result

    def remove_hooks(self):
        for hook in self.hooks:
            hook.remove()
        self.hooks = []
```

```python
    def clear(self):
        self.stats.clear()
```

[6]:
```python
# Weight and Filter Importance Functions

def weight_importance_mag(weight):
    """Traditional magnitude-based importance"""
    return weight.abs()


def weight_importance_act_aware(weight, typical_act):
    """Activation-aware importance: weight * typical activation"""
    act_exp = typical_act.view(1, -1, 1, 1).to(weight.device)
    return weight.abs() * act_exp


def weight_importance_hybrid(weight, typical_act, alpha=0.5):
    """Hybrid importance: combination of magnitude and activation-aware"""
    mag = weight.abs()
    act = weight_importance_act_aware(weight, typical_act)

    mag_norm = (mag - mag.min()) / (mag.max() - mag.min() + 1e-8)
    act_norm = (act - act.min()) / (act.max() - act.min() + 1e-8)

    return alpha * mag_norm + (1 - alpha) * act_norm


def filter_importance_mag(weight):
    """Filter-level magnitude importance (for structured pruning)"""
    return weight.view(weight.size(0), -1).norm(dim=1)


def filter_importance_act_aware(weight, typical_act):
    """Filter-level activation-aware importance"""
    importance = weight_importance_act_aware(weight, typical_act)
    return importance.sum(dim=(1, 2, 3))
```

[7]:
```python
# Activation-Aware Pruner Class
class ActivationAwarePruner:
    def __init__(self, model, sparsity=0.5, structured=False, 
 ↪normalize_per_layer=False,
                 mode='activation_aware', hybrid_alpha=0.5):
        self.model = model
        self.sparsity = sparsity
        self.structured = structured
        self.normalize_per_layer = normalize_per_layer
```

```
        self.mode = mode
        self.hybrid_alpha = hybrid_alpha
        self.masks = {}
        self.typical_acts = {}

    def collect_stats(self, dataloader, num_batches=200):
        print(f"Collecting activation stats from {num_batches} batches...")

        collector = ActivationCollector()
        collector.register_hooks(self.model)

        self.model.eval()
        device = next(self.model.parameters()).device

        with torch.no_grad():
            for i, (inputs, _) in enumerate(dataloader):
                if i >= num_batches:
                    break
                inputs = inputs.to(device)
                _ = self.model(inputs)

        self.typical_acts = collector.compute_stats()
        collector.remove_hooks()

        print(f"Collected stats for {len(self.typical_acts)} layers")

    def compute_importance(self):
        scores = {}

        for name, module in self.model.named_modules():
            if isinstance(module, QuantConv2d):
                weight = module.weight.data

                if self.structured:
                    if self.mode == 'traditional':
                        imp = filter_importance_mag(weight)
                    elif self.mode == 'activation_aware':
                        if name in self.typical_acts:
                            imp = filter_importance_act_aware(weight, self.
↪typical_acts[name])
                        else:
                            print(f"Warning: No stats for {name}, using␣
↪magnitude")
                            imp = filter_importance_mag(weight)
                    elif self.mode == 'hybrid':
                        if name in self.typical_acts:
                            mag_imp = filter_importance_mag(weight)
```

```
                            act_imp = filter_importance_act_aware(weight, self.
↪typical_acts[name])
                            mag_norm = (mag_imp - mag_imp.min()) / (mag_imp.
↪max() - mag_imp.min() + 1e-8)
                            act_norm = (act_imp - act_imp.min()) / (act_imp.
↪max() - act_imp.min() + 1e-8)
                            imp = self.hybrid_alpha * mag_norm + (1 - self.
↪hybrid_alpha) * act_norm
                        else:
                            imp = filter_importance_mag(weight)
                else:
                    if self.mode == 'traditional':
                        imp = weight_importance_mag(weight)
                    elif self.mode == 'activation_aware':
                        if name in self.typical_acts:
                            imp = weight_importance_act_aware(weight, self.
↪typical_acts[name])
                        else:
                            print(f"Warning: No stats for {name}, using␣
↪magnitude")
                            imp = weight_importance_mag(weight)
                    elif self.mode == 'hybrid':
                        if name in self.typical_acts:
                            imp = weight_importance_hybrid(weight, self.
↪typical_acts[name], self.hybrid_alpha)
                        else:
                            imp = weight_importance_mag(weight)

                scores[name] = imp

        return scores

    def compute_threshold(self, scores):
        all_imp = []

        for name, imp in scores.items():
            if self.normalize_per_layer:
                imp_min = imp.min()
                imp_max = imp.max()
                if imp_max > imp_min:
                    norm = (imp - imp_min) / (imp_max - imp_min)
                else:
                    norm = torch.zeros_like(imp)
                all_imp.append(norm.flatten())
            else:
                all_imp.append(imp.flatten())
```

```
        all_imp = torch.cat(all_imp)
        k = max(1, int(len(all_imp) * self.sparsity))
        threshold = torch.kthvalue(all_imp, k).values.item()

        return threshold

    def create_masks(self, scores, threshold):
        masks = {}

        for name, module in self.model.named_modules():
            if isinstance(module, QuantConv2d) and name in scores:
                imp = scores[name]
                weight = module.weight.data

                if self.normalize_per_layer:
                    imp_min = imp.min()
                    imp_max = imp.max()
                    if imp_max > imp_min:
                        imp = (imp - imp_min) / (imp_max - imp_min)
                    else:
                        imp = torch.zeros_like(imp)

                if self.structured:
                    filter_mask = (imp >= threshold).float()
                    mask = filter_mask.view(-1, 1, 1, 1).expand_as(weight)
                else:
                    mask = (imp >= threshold).float()

                masks[name] = mask.to(weight.device)

        return masks

    def apply_pruning(self):
        print(f"\nApplying {self.mode} pruning with {self.sparsity*100:.1f}% sparsity...")

        scores = self.compute_importance()
        threshold = self.compute_threshold(scores)
        print(f"Threshold: {threshold:.6f}")

        self.masks = self.create_masks(scores, threshold)

        total = 0
        pruned = 0

        for name, module in self.model.named_modules():
```

```python
            if isinstance(module, QuantConv2d) and name in self.masks:
                mask = self.masks[name]
                with torch.no_grad():
                    module.weight.data *= mask

                total += mask.numel()
                pruned += (mask == 0).sum().item()

        actual = pruned / total
        print(f"Actual sparsity: {actual*100:.2f}%")

        return actual

    def get_layer_stats(self):
        stats = {}
        for name, module in self.model.named_modules():
            if isinstance(module, QuantConv2d) and name in self.masks:
                mask = self.masks[name]
                sp = (mask == 0).sum().item() / mask.numel()
                stats[name] = {
                    'sparsity': sp,
                    'total': mask.numel(),
                    'pruned': (mask == 0).sum().item()
                }
        return stats
```

[8]:

```python
# Evaluate and Fine-tune Functions

def evaluate(model, dataloader, device):
    """Evaluate model accuracy on a dataset"""
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return 100. * correct / total


def finetune(model, trainloader, testloader, masks, epochs=5, lr=0.001):
    """Fine-tune the pruned model while maintaining sparsity"""
```

```
    device = next(model.parameters()).device
    criterion = nn.CrossEntropyLoss().to(device)
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9,␣
 ↪weight_decay=1e-4)

    best_acc = 0

    for epoch in range(epochs):
        model.train()
        for inputs, targets in trainloader:
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

            # Re-apply masks to maintain sparsity
            with torch.no_grad():
                for name, module in model.named_modules():
                    if isinstance(module, QuantConv2d) and name in masks:
                        module.weight.data *= masks[name]

        acc = evaluate(model, testloader, device)
        if acc > best_acc:
            best_acc = acc
        print(f"  Epoch {epoch+1}/{epochs}: Accuracy = {acc:.2f}%")

    return best_acc
```

[9]:
```
# Compare Pruning Methods Function

def compare_pruning_methods(model, trainloader, testloader, sparsity=0.5,␣
 ↪finetune_epochs=5):
    """Compare traditional, activation-aware, and hybrid pruning methods"""
    print("="*70)
    print(f"Comparing pruning methods at {sparsity*100:.0f}% sparsity (with␣
 ↪{finetune_epochs} epochs fine-tuning)")
    print("="*70)

    device = next(model.parameters()).device
    original_state = copy.deepcopy(model.state_dict())

    results = {}

    # Traditional Magnitude Pruning
```

```
    print("\n[1] Traditional Magnitude Pruning")
    model.load_state_dict(original_state)

    pruner = ActivationAwarePruner(model, sparsity=sparsity, mode='traditional')
    sp = pruner.apply_pruning()
    acc_before = evaluate(model, testloader, device)

    print(f"Accuracy before fine-tune: {acc_before:.2f}%")
    print(f"Fine-tuning...")
    acc_after = finetune(model, trainloader, testloader, pruner.masks,
↪epochs=finetune_epochs)
    print(f"Accuracy after fine-tune: {acc_after:.2f}%")

    results['traditional'] = {
        'sparsity': sp,
        'acc_before': acc_before,
        'acc_after': acc_after,
    }

    # Activation-aware Pruning
    print("\n[2] Activation-Aware Pruning")
    model.load_state_dict(original_state)

    pruner = ActivationAwarePruner(model, sparsity=sparsity,
↪mode='activation_aware')
    pruner.collect_stats(trainloader, num_batches=200)
    sp = pruner.apply_pruning()
    acc_before = evaluate(model, testloader, device)

    print(f"Accuracy before fine-tune: {acc_before:.2f}%")
    print(f"Fine-tuning...")
    acc_after = finetune(model, trainloader, testloader, pruner.masks,
↪epochs=finetune_epochs)
    print(f"Accuracy after fine-tune: {acc_after:.2f}%")

    results['activation_aware'] = {
        'sparsity': sp,
        'acc_before': acc_before,
        'acc_after': acc_after,
    }

    # Hybrid Pruning
    print("\n[3] Hybrid Pruning (alpha=0.5)")
    model.load_state_dict(original_state)

    pruner = ActivationAwarePruner(model, sparsity=sparsity, mode='hybrid',
↪hybrid_alpha=0.5)
```

```
    pruner.collect_stats(trainloader, num_batches=200)
    sp = pruner.apply_pruning()
    acc_before = evaluate(model, testloader, device)

    print(f"Accuracy before fine-tune: {acc_before:.2f}%")
    print(f"Fine-tuning...")
    acc_after = finetune(model, trainloader, testloader, pruner.masks,␣
 ↪epochs=finetune_epochs)
    print(f"Accuracy after fine-tune: {acc_after:.2f}%")

    results['hybrid'] = {
        'sparsity': sp,
        'acc_before': acc_before,
        'acc_after': acc_after,
    }

    # Summary
    print("\n" + "="*70)
    print("SUMMARY")
    print("="*70)
    print(f"{'Method':<25} {'Sparsity':>12} {'Before':>12} {'After':>12}")
    print("-"*65)
    for method, stats in results.items():
        print(f"{method:<25} {stats['sparsity']*100:>10.2f}%␣
 ↪{stats['acc_before']:>10.2f}% {stats['acc_after']:>10.2f}%")

    best = max(results.keys(), key=lambda x: results[x]['acc_after'])
    print(f"\nBest method: {best} ({results[best]['acc_after']:.2f}%)")

    # Restore original state
    model.load_state_dict(original_state)
    return results
```

[16]:

```
# Hardware Efficiency Analysis Functions

def compute_mac_contribution(weight, typical_act, mask):
    """Compute average MAC contribution for kept weights"""
    act_exp = typical_act.view(1, -1, 1, 1).to(weight.device)
    contrib = weight.abs() * act_exp

    kept = mask > 0
    if kept.sum() > 0:
        avg = contrib[kept].mean().item()
        total = contrib[kept].sum().item()
    else:
        avg = 0
        total = 0
```

```
    return avg, total


def compute_pe_util(mask, tile_size=8):
    """Compute PE (Processing Element) utilization for systolic array - VECTORIZED"""
    flat_mask = mask.view(mask.size(0), -1).float()
    out_dim, in_dim = flat_mask.shape

    # Pad to make dimensions divisible by tile_size
    pad_out = (tile_size - out_dim % tile_size) % tile_size
    pad_in = (tile_size - in_dim % tile_size) % tile_size

    if pad_out > 0 or pad_in > 0:
        flat_mask = F.pad(flat_mask, (0, pad_in, 0, pad_out), value=0)

    # Reshape into tiles and compute mean per tile
    new_out, new_in = flat_mask.shape
    tiles = flat_mask.view(new_out // tile_size, tile_size, new_in // tile_size, tile_size)
    tile_utils = tiles.mean(dim=(1, 3)).flatten()

    return tile_utils.mean().item(), tile_utils.min().item(), tile_utils.var().item()


def compute_zero_clustering(mask):
    """Compute zero clustering score - VECTORIZED (approximate but fast)"""
    flat = mask.flatten()
    total_zeros = (flat == 0).sum().item()
    total_elements = flat.numel()

    if total_zeros == 0:
        return 0.0, 0.0, 0

    # Approximate run length by looking at transitions
    # A transition occurs when consecutive elements differ
    transitions = (flat[1:] != flat[:-1]).sum().item()

    # Estimate number of zero runs (roughly half the transitions if sparsity ~50%)
    num_runs = max(1, (transitions + 1) // 2)

    # Estimate average run length
    avg_run = total_zeros / num_runs
```

```python
    # Approximate max run (heuristic: assume max is ~2-3x average for random␣
↪sparsity)
    max_run = min(avg_run * 2.5, total_zeros)

    run_length_score = avg_run / max(total_zeros, 1)
    run_count_score = 1.0 / (1.0 + np.log(num_runs + 1))

    score = 0.5 * run_length_score + 0.5 * run_count_score

    return score, avg_run, int(max_run)


def compute_2d_clustering(mask, block_size=4):
    """Compute 2D block sparsity pattern - VECTORIZED"""
    flat = mask.view(mask.size(0), -1).float()
    out_dim, in_dim = flat.shape

    # Pad to make dimensions divisible by block_size
    pad_out = (block_size - out_dim % block_size) % block_size
    pad_in = (block_size - in_dim % block_size) % block_size

    if pad_out > 0 or pad_in > 0:
        flat = F.pad(flat, (0, pad_in, 0, pad_out), value=1)  # Pad with 1s␣
↪(non-zero)

    new_out, new_in = flat.shape

    # Reshape into blocks
    blocks = flat.view(new_out // block_size, block_size, new_in // block_size,␣
↪block_size)

    # Sum each block to find zeros per block
    block_sums = blocks.sum(dim=(1, 3))  # Shape: (num_blocks_out,␣
↪num_blocks_in)
    block_size_sq = block_size * block_size

    # Full zero blocks have sum == 0, partial have 0 < sum < block_size^2
    full_zero = (block_sums == 0).sum().item()
    partial_zero = ((block_sums > 0) & (block_sums < block_size_sq)).sum().
↪item()
    total = block_sums.numel()

    block_score = full_zero / max(total, 1)
    partial_ratio = partial_zero / max(total, 1)

    return block_score, partial_ratio
```

```
def estimate_tops_per_watt(sparsity, contrib_ratio, pe_util, clust_score,␣
↪base=10.0):
    """Estimate TOPS/Watt improvement from pruning"""
    density = 1.0 - sparsity

    compute_pwr = density
    data_mvmt = density * 0.8 + 0.2
    ctrl_ovhd = 0.1 * (1.0 - clust_score)

    overall_pwr = 0.4 * compute_pwr + 0.5 * data_mvmt + 0.1 * (1.0 + ctrl_ovhd)

    eff_throughput = min(1.0 + 0.2 * (contrib_ratio - 1.0), 1.5)

    improvement = eff_throughput / overall_pwr
    tops = base * improvement

    breakdown = {
        'compute_pwr': compute_pwr,
        'data_mvmt': data_mvmt,
        'ctrl_ovhd': ctrl_ovhd,
        'overall_pwr': overall_pwr,
        'eff_throughput': eff_throughput
    }

    return tops, improvement, breakdown
```

[11]:

```
# Comprehensive Hardware Efficiency Analysis

def analyze_hw_efficiency(model, pruner, dataloader):
    """Analyze hardware efficiency metrics comparing traditional vs␣
↪activation-aware pruning"""
    print("\n" + "="*70)
    print("Hardware Efficiency Analysis")
    print("="*70)

    device = next(model.parameters()).device

    if not pruner.typical_acts:
        pruner.collect_stats(dataloader, num_batches=200)

    total_orig_macs = 0
    total_eff_trad = 0
    total_eff_act = 0

    total_c_trad = 0
    total_c_act = 0
```

```
    kept_w_trad = 0
    kept_w_act = 0

    pe_utils_trad = []
    pe_utils_act = []
    clust_trad = []
    clust_act = []

    layer_metrics = {}

    for name, module in model.named_modules():
        if isinstance(module, QuantConv2d):
            weight = module.weight.data
            out_ch, in_ch, kH, kW = weight.shape

            spatial = 4
            macs = out_ch * in_ch * kH * kW * spatial * spatial
            total_orig_macs += macs

            if name in pruner.typical_acts:
                typ_act = pruner.typical_acts[name].to(device)

                # Traditional mask (magnitude-based)
                w_imp = weight.abs()
                thresh_trad = w_imp.flatten().quantile(pruner.sparsity)
                mask_trad = (w_imp >= thresh_trad).float()

                # Activation-aware mask
                a_imp = weight_importance_act_aware(weight, typ_act)
                thresh_act = a_imp.flatten().quantile(pruner.sparsity)
                mask_act = (a_imp >= thresh_act).float()

                # Compute metrics
                avg_c_trad, tot_c_trad = compute_mac_contribution(weight, typ_act, mask_trad)
                avg_c_act, tot_c_act = compute_mac_contribution(weight, typ_act, mask_act)

                total_c_trad += tot_c_trad
                total_c_act += tot_c_act
                kept_w_trad += mask_trad.sum().item()
                kept_w_act += mask_act.sum().item()

                pe_u_trad, _, _ = compute_pe_util(mask_trad)
                pe_u_act, _, _ = compute_pe_util(mask_act)

                pe_utils_trad.append(pe_u_trad)
```

```
                pe_utils_act.append(pe_u_act)

                cl_trad, _, _ = compute_zero_clustering(mask_trad)
                cl_act, _, _ = compute_zero_clustering(mask_act)

                clust_trad.append(cl_trad)
                clust_act.append(cl_act)

                bs_trad, _ = compute_2d_clustering(mask_trad)
                bs_act, _ = compute_2d_clustering(mask_act)

                layer_metrics[name] = {
                    'contrib_per_mac': {'trad': avg_c_trad, 'act': avg_c_act},
                    'pe_util': {'trad': pe_u_trad, 'act': pe_u_act},
                    'clust': {'trad': cl_trad, 'act': cl_act},
                    'block_sp': {'trad': bs_trad, 'act': bs_act}
                }

                total_eff_trad += mask_trad.sum().item() * spatial * spatial
                total_eff_act += mask_act.sum().item() * spatial * spatial

    # Aggregate metrics
    print("\nAggregate Metrics:")

    avg_c_trad = total_c_trad / max(kept_w_trad, 1)
    avg_c_act = total_c_act / max(kept_w_act, 1)
    c_imp = avg_c_act / max(avg_c_trad, 1e-8)

    print(f"  Output contribution/MAC:")
    print(f"    Traditional: {avg_c_trad:.6f}")
    print(f"    Act-aware:   {avg_c_act:.6f} ({c_imp:.2f}x)")

    avg_pe_trad = np.mean(pe_utils_trad)
    avg_pe_act = np.mean(pe_utils_act)
    pe_diff = (avg_pe_act - avg_pe_trad) * 100

    print(f"  PE utilization:")
    print(f"    Traditional: {avg_pe_trad:.2%}")
    print(f"    Act-aware:   {avg_pe_act:.2%} ({pe_diff:+.2f}%)")

    avg_cl_trad = np.mean(clust_trad)
    avg_cl_act = np.mean(clust_act)
    cl_imp = (avg_cl_act - avg_cl_trad) / max(avg_cl_trad, 0.01) * 100

    print(f"  Zero clustering:")
    print(f"    Traditional: {avg_cl_trad:.4f}")
    print(f"    Act-aware:   {avg_cl_act:.4f} ({cl_imp:+.1f}%)")
```

```
    tops_trad, _, _ = estimate_tops_per_watt(pruner.sparsity, 1.0, avg_pe_trad,␣
↪avg_cl_trad)
    tops_act, _, breakdown = estimate_tops_per_watt(pruner.sparsity, c_imp,␣
↪avg_pe_act, avg_cl_act)

    rel_imp = (tops_act - tops_trad) / tops_trad * 100

    print(f"  TOPS/Watt:")
    print(f"    Baseline:    10.00")
    print(f"    Traditional: {tops_trad:.2f}")
    print(f"    Act-aware:   {tops_act:.2f} ({rel_imp:+.1f}%)")

    return {
        'contrib_imp': c_imp,
        'pe_util_diff': pe_diff,
        'clust_imp': cl_imp,
        'tops_imp': rel_imp,
        'layer_metrics': layer_metrics
    }
```

[12]:
```
# Evaluate Original Model
print("="*70)
print("Evaluating Original Model")
print("="*70)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")

# Load the model checkpoint
PATH = "result/VGG16_quant/model_best.pth.tar"
checkpoint = torch.load(PATH, map_location=device)
model.load_state_dict(checkpoint['state_dict'])
model = model.to(device)

# Evaluate original accuracy
orig_acc = evaluate(model, testloader, device)
print(f"Original model accuracy: {orig_acc:.2f}%")
```

```
======================================================================
Evaluating Original Model
======================================================================
Device: cuda
Original model accuracy: 90.33%
```

[13]:
```
# Compare Pruning Methods at Different Sparsity Levels
print("\n" + "="*70)
```

```python
print("Comparing Pruning Methods at Different Sparsity Levels")
print("="*70)

sparsity_levels = [0.3, 0.5, 0.7]
all_results = {}

for sp in sparsity_levels:
    print(f"\n{'='*70}")
    print(f"Sparsity: {sp*100:.0f}%")
    print("="*70)

    # Reload original model state for fair comparison
    model.load_state_dict(checkpoint['state_dict'])
    model = model.to(device)

    results = compare_pruning_methods(model, trainloader, testloader,
                                      sparsity=sp, finetune_epochs=5)
    all_results[sp] = results
```

```
======================================================================
Comparing Pruning Methods at Different Sparsity Levels
======================================================================

======================================================================
Sparsity: 30%
======================================================================
======================================================================
Comparing pruning methods at 30% sparsity (with 5 epochs fine-tuning)
======================================================================

[1] Traditional Magnitude Pruning

Applying traditional pruning with 30.0% sparsity…
Threshold: 0.003790
Actual sparsity: 30.00%
Accuracy before fine-tune: 88.07%
Fine-tuning…
  Epoch 1/5: Accuracy = 90.05%
  Epoch 2/5: Accuracy = 90.37%
  Epoch 3/5: Accuracy = 90.20%
  Epoch 4/5: Accuracy = 90.18%
  Epoch 5/5: Accuracy = 90.05%
Accuracy after fine-tune: 90.37%

[2] Activation-Aware Pruning
Collecting activation stats from 200 batches…
Collected stats for 13 layers
```

```
Applying activation_aware pruning with 30.0% sparsity…
Threshold: 0.001241
Actual sparsity: 30.00%
Accuracy before fine-tune: 85.39%
Fine-tuning…
  Epoch 1/5: Accuracy = 89.83%
  Epoch 2/5: Accuracy = 90.16%
  Epoch 3/5: Accuracy = 90.10%
  Epoch 4/5: Accuracy = 90.12%
  Epoch 5/5: Accuracy = 90.25%
Accuracy after fine-tune: 90.25%

[3] Hybrid Pruning (alpha=0.5)
Collecting activation stats from 200 batches…
Collected stats for 13 layers

Applying hybrid pruning with 30.0% sparsity…
Threshold: 0.145644
Actual sparsity: 30.00%
Accuracy before fine-tune: 24.17%
Fine-tuning…
  Epoch 1/5: Accuracy = 89.67%
  Epoch 2/5: Accuracy = 89.94%
  Epoch 3/5: Accuracy = 90.05%
  Epoch 4/5: Accuracy = 90.11%
  Epoch 5/5: Accuracy = 89.93%
Accuracy after fine-tune: 90.11%

======================================================================
SUMMARY
======================================================================
Method                        Sparsity      Before       After
------------------------------------------------------------------
traditional                     30.00%      88.07%      90.37%
activation_aware                30.00%      85.39%      90.25%
hybrid                          30.00%      24.17%      90.11%

Best method: traditional (90.37%)

======================================================================
Sparsity: 50%
======================================================================
======================================================================
Comparing pruning methods at 50% sparsity (with 5 epochs fine-tuning)
======================================================================

[1] Traditional Magnitude Pruning
```

```
Applying traditional pruning with 50.0% sparsity…
Threshold: 0.006388
Actual sparsity: 50.00%
Accuracy before fine-tune: 84.33%
Fine-tuning…
  Epoch 1/5: Accuracy = 90.01%
  Epoch 2/5: Accuracy = 90.15%
  Epoch 3/5: Accuracy = 90.21%
  Epoch 4/5: Accuracy = 90.21%
  Epoch 5/5: Accuracy = 90.45%
Accuracy after fine-tune: 90.45%

[2] Activation-Aware Pruning
Collecting activation stats from 200 batches…
Collected stats for 13 layers

Applying activation_aware pruning with 50.0% sparsity…
Threshold: 0.002112
Actual sparsity: 50.00%
Accuracy before fine-tune: 71.02%
Fine-tuning…
  Epoch 1/5: Accuracy = 89.83%
  Epoch 2/5: Accuracy = 90.11%
  Epoch 3/5: Accuracy = 90.25%
  Epoch 4/5: Accuracy = 90.15%
  Epoch 5/5: Accuracy = 90.22%
Accuracy after fine-tune: 90.25%

[3] Hybrid Pruning (alpha=0.5)
Collecting activation stats from 200 batches…
Collected stats for 13 layers

Applying hybrid pruning with 50.0% sparsity…
Threshold: 0.248203
Actual sparsity: 50.00%
Accuracy before fine-tune: 10.00%
Fine-tuning…
  Epoch 1/5: Accuracy = 88.24%
  Epoch 2/5: Accuracy = 88.88%
  Epoch 3/5: Accuracy = 89.06%
  Epoch 4/5: Accuracy = 89.13%
  Epoch 5/5: Accuracy = 89.25%
Accuracy after fine-tune: 89.25%

======================================================================
SUMMARY
======================================================================
```

```
Method                        Sparsity      Before       After
-----------------------------------------------------------------
traditional                    50.00%      84.33%      90.45%
activation_aware               50.00%      71.02%      90.25%
hybrid                         50.00%      10.00%      89.25%

Best method: traditional (90.45%)

======================================================================
Sparsity: 70%
======================================================================
======================================================================
Comparing pruning methods at 70% sparsity (with 5 epochs fine-tuning)
======================================================================

[1] Traditional Magnitude Pruning

Applying traditional pruning with 70.0% sparsity…
Threshold: 0.009266
Actual sparsity: 70.00%
Accuracy before fine-tune: 76.26%
Fine-tuning…
  Epoch 1/5: Accuracy = 89.89%
  Epoch 2/5: Accuracy = 89.94%
  Epoch 3/5: Accuracy = 90.22%
  Epoch 4/5: Accuracy = 90.03%
  Epoch 5/5: Accuracy = 90.14%
Accuracy after fine-tune: 90.22%

[2] Activation-Aware Pruning
Collecting activation stats from 200 batches…
Collected stats for 13 layers

Applying activation_aware pruning with 70.0% sparsity…
Threshold: 0.003150
Actual sparsity: 70.00%
Accuracy before fine-tune: 27.31%
Fine-tuning…
  Epoch 1/5: Accuracy = 89.55%
  Epoch 2/5: Accuracy = 89.70%
  Epoch 3/5: Accuracy = 89.65%
  Epoch 4/5: Accuracy = 89.50%
  Epoch 5/5: Accuracy = 89.84%
Accuracy after fine-tune: 89.84%

[3] Hybrid Pruning (alpha=0.5)
Collecting activation stats from 200 batches…
Collected stats for 13 layers
```

```
Applying hybrid pruning with 70.0% sparsity…
Threshold: 0.362611
Actual sparsity: 70.00%
Accuracy before fine-tune: 10.00%
Fine-tuning…
  Epoch 1/5: Accuracy = 82.84%
  Epoch 2/5: Accuracy = 85.12%
  Epoch 3/5: Accuracy = 85.75%
  Epoch 4/5: Accuracy = 86.38%
  Epoch 5/5: Accuracy = 86.85%
Accuracy after fine-tune: 86.85%

======================================================================
SUMMARY
======================================================================
Method                        Sparsity       Before        After
-----------------------------------------------------------------
traditional                     70.00%       76.26%       90.22%
activation_aware                70.00%       27.31%       89.84%
hybrid                          70.00%       10.00%       86.85%

Best method: traditional (90.22%)
```

[14]:

```
# Final Summary Table
print("\n" + "="*70)
print("Final Summary - Accuracy After Fine-tuning")
print("="*70)
print(f"{'Sparsity':<12} {'Traditional':>15} {'Act-Aware':>15} {'Hybrid':>15}")
print("-"*60)

for sp in sparsity_levels:
    r = all_results[sp]
    print(f"{sp*100:>6.0f}%      "
          f"{r['traditional']['acc_after']:>13.2f}% "
          f"{r['activation_aware']['acc_after']:>13.2f}% "
          f"{r['hybrid']['acc_after']:>13.2f}%")

print("\nImprovement over Traditional:")
print("-"*60)
for sp in sparsity_levels:
    r = all_results[sp]
    trad = r['traditional']['acc_after']
    act = r['activation_aware']['acc_after']
    hyb = r['hybrid']['acc_after']
    print(f"{sp*100:>6.0f}%      "
          f"{'baseline':>13} "
```

```
         f"{act - trad:>+12.2f}% "
         f"{hyb - trad:>+12.2f}%")
```

```
======================================================================
Final Summary - Accuracy After Fine-tuning
======================================================================
Sparsity         Traditional       Act-Aware          Hybrid
------------------------------------------------------------
    30%              90.37%          90.25%          90.11%
    50%              90.45%          90.25%          89.25%
    70%              90.22%          89.84%          86.85%

Improvement over Traditional:
------------------------------------------------------------
    30%           baseline         -0.12%         -0.26%
    50%           baseline         -0.20%         -1.20%
    70%           baseline         -0.38%         -3.37%
```

[17]:

```
# Hardware Efficiency Analysis at 50% Sparsity
print("\n" + "="*70)
print("Hardware Efficiency Analysis at 50% Sparsity")
print("="*70)

# Reload the original model
model.load_state_dict(checkpoint['state_dict'])
model = model.to(device)

# Create pruner and collect activation statistics
pruner = ActivationAwarePruner(model, sparsity=0.5, mode='activation_aware')
pruner.collect_stats(trainloader, num_batches=200)
pruner.apply_pruning()

# Analyze hardware efficiency
hw_metrics = analyze_hw_efficiency(model, pruner, testloader)

print("\n" + "="*70)
print("Analysis Complete")
print("="*70)
```

```
======================================================================
Hardware Efficiency Analysis at 50% Sparsity
======================================================================
Collecting activation stats from 200 batches…
Collected stats for 13 layers

Applying activation_aware pruning with 50.0% sparsity…
```

```
Threshold: 0.002111
Actual sparsity: 50.00%

======================================================================
Hardware Efficiency Analysis
======================================================================

Aggregate Metrics:
  Output contribution/MAC:
    Traditional: 0.003526
    Act-aware:   0.003541 (1.00x)
  PE utilization:
    Traditional: 60.94%
    Act-aware:   60.94% (+0.00%)
  Zero clustering:
    Traditional: 0.0405
    Act-aware:   0.0406 (+0.4%)
  TOPS/Watt:
    Baseline:    10.00
    Traditional: 16.40
    Act-aware:   16.42 (+0.1%)

======================================================================
Analysis Complete
======================================================================
```

[18]:

```python
# Layer-by-Layer Sparsity Statistics
print("\n" + "="*70)
print("Layer-by-Layer Sparsity Statistics")
print("="*70)

layer_stats = pruner.get_layer_stats()
print(f"{'Layer':<30} {'Sparsity':>12} {'Pruned':>12} {'Total':>12}")
print("-"*70)

for name, stats in layer_stats.items():
    print(f"{name:<30} {stats['sparsity']*100:>10.2f}% {stats['pruned']:>12,}␣
  ↪{stats['total']:>12,}")
```

```
======================================================================
Layer-by-Layer Sparsity Statistics
======================================================================
Layer                              Sparsity       Pruned        Total
----------------------------------------------------------------------
features.0                            0.75%           13        1,728
features.3                            9.43%        3,478       36,864
features.7                            6.41%        4,726       73,728
```

```
features.10                              18.10%       26,696      147,456
features.14                              12.45%       36,726      294,912
features.17                              42.82%      252,578      589,824
features.20                              54.70%      322,611      589,824
features.24                               8.26%        1,522       18,432
features.27                               1.74%           10          576
features.29                               0.62%          227       36,864
features.33                              31.22%      736,523    2,359,296
features.36                              64.43%    1,520,129    2,359,296
features.39                              64.80%    1,528,808    2,359,296
```

[ ]: