Imagenet

December 14, 2025

1 CS 441 final project

[3]:

```python
import os
import numpy as np
from PIL import Image
import time
import torch
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms, models

from sklearn.metrics import classification_report, confusion_matrix
from sklearn.svm import LinearSVC
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier
from sklearn.decomposition import PCA

import torch.nn as nn
import torch.optim as optim
DRIVE_BASE = "/content/drive/My Drive/CS441/Final/garbage_dataset"
LOCAL_TRAIN = "/content/garbage_train"
LOCAL_TEST  = "/content/garbage_test"

FORCE_SYNC = False

from google.colab import drive
drive.mount("/content/drive")

!pip -q install pillow-heif
from PIL import Image
import pillow_heif
pillow_heif.register_heif_opener()

import os, subprocess

IMG_EXTS = (".jpg", ".jpeg", ".png", ".bmp", ".webp", ".heic", ".heif")

def count_images(root):
    if not os.path.exists(root):
        return 0
    c = 0
    for _, _, files in os.walk(root):
        for f in files:
            if f.lower().endswith(IMG_EXTS):
                c += 1
    return c

def rsync_dir(src, dst):
    cmd = ["rsync", "-a", "--delete", "--info=progress2", src.rstrip("/") + "/
  ↪", dst.rstrip("/") + "/"]
    print("Running:", " ".join([f'"{x}"' if " " in x else x for x in cmd]))
    subprocess.run(cmd, check=True)

drive_train = os.path.join(DRIVE_BASE, "train")
drive_test  = os.path.join(DRIVE_BASE, "test")

if not os.path.exists(drive_train):
    raise FileNotFoundError(f"Not found: {drive_train}")
if not os.path.exists(drive_test):
    raise FileNotFoundError(f"Not found: {drive_test}")

drive_train_cnt = count_images(drive_train)
drive_test_cnt  = count_images(drive_test)

local_train_cnt = count_images(LOCAL_TRAIN)
local_test_cnt  = count_images(LOCAL_TEST)

print(f"Drive train images: {drive_train_cnt} | Local train images:␣
  ↪{local_train_cnt}")
print(f"Drive test  images: {drive_test_cnt}  | Local test  images:␣
  ↪{local_test_cnt}")

need_sync = FORCE_SYNC or (drive_train_cnt != local_train_cnt) or␣
  ↪(drive_test_cnt != local_test_cnt)
print("Need sync:", need_sync)

if need_sync:
    rsync_dir(drive_train, LOCAL_TRAIN)
    rsync_dir(drive_test,  LOCAL_TEST)

train_root = LOCAL_TRAIN
test_root  = LOCAL_TEST

local_train_cnt2 = count_images(train_root)
local_test_cnt2  = count_images(test_root)
```

```

print("\n=== READY ===")
print("train_root =", train_root, "| images:", local_train_cnt2)
print("test_root  =", test_root,  "| images:", local_test_cnt2)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
Drive train images: 20371 | Local train images: 15943
Drive test  images: 251   | Local test  images: 0
Need sync: True
Running: rsync -a --delete --info=progress2 "/content/drive/My
Drive/CS441/Final/garbage_dataset/train/" /content/garbage_train/
Running: rsync -a --delete --info=progress2 "/content/drive/My
Drive/CS441/Final/garbage_dataset/test/" /content/garbage_test/

=== READY ===
train_root = /content/garbage_train | images: 20371
test_root  = /content/garbage_test | images: 251
```

[4]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Device:", device)
```

```
Device: cuda
```

[5]:
```
class GarbageFineDataset(Dataset):
    def __init__(self, root_dir, transform=None,
                 fine_class_to_idx=None, big_class_to_idx=None,
                 strict=False,
                 extensions=('.jpg', '.jpeg', '.png', '.bmp', '.webp', '.heic',␣
↪'.heif')):
        self.root_dir = root_dir
        self.transform = transform
        self.extensions = extensions
        self.strict = strict

        self.fine_class_to_idx = {} if fine_class_to_idx is None else␣
↪dict(fine_class_to_idx)
        self.big_class_to_idx  = {} if big_class_to_idx  is None else␣
↪dict(big_class_to_idx)

        self.samples = []  # (image_path, fine_idx, big_idx)

        big_names = sorted([d for d in os.listdir(root_dir) if os.path.isdir(os.
↪path.join(root_dir, d))])
        for big_name in big_names:
            big_path = os.path.join(root_dir, big_name)

            if big_name in self.big_class_to_idx:
                big_idx = self.big_class_to_idx[big_name]
            else:
                if self.strict:
                    raise ValueError(f"[STRICT] test/show train/not exist big␣
↪category: {big_name}")
                big_idx = len(self.big_class_to_idx)
                self.big_class_to_idx[big_name] = big_idx

            fine_names = sorted([d for d in os.listdir(big_path) if os.path.
↪isdir(os.path.join(big_path, d))])
            for fine_name in fine_names:
                fine_path = os.path.join(big_path, fine_name)

                fine_full_name = f"{big_name}/{fine_name}"

                if fine_full_name in self.fine_class_to_idx:
                    fine_idx = self.fine_class_to_idx[fine_full_name]
                else:
                    if self.strict:
                        raise ValueError(f"[[STRICT] test/show train/not exist␣
↪small category: {fine_full_name}")
                    fine_idx = len(self.fine_class_to_idx)
                    self.fine_class_to_idx[fine_full_name] = fine_idx


                for fname in os.listdir(fine_path):
                    if fname.lower().endswith(self.extensions):
                        img_path = os.path.join(fine_path, fname)
                        self.samples.append((img_path, fine_idx, big_idx))

        print(f"[{root_dir}] samples={len(self.samples)}, big={len(self.
↪big_class_to_idx)}, fine={len(self.fine_class_to_idx)}")

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        img_path, fine_idx, big_idx = self.samples[idx]
        img = Image.open(img_path).convert("RGB")
        if self.transform:
            img = self.transform(img)
        return img, fine_idx, big_idx

    @property
    def fine_idx_to_name(self):
        return {v: k for k, v in self.fine_class_to_idx.items()}
    @property
    def big_idx_to_name(self):
        return {v: k for k, v in self.big_class_to_idx.items()}

    @property
    def fine_to_big(self):
        # fine_idx -> big_idx
        mapping = {}
        for _, f_idx, b_idx in self.samples:
            mapping[f_idx] = b_idx
        return mapping
```

[6]:

```
img_size = 224
batch_size = 32
num_workers = 4

train_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomResizedCrop(img_size),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

val_test_transform = transforms.Compose([
    transforms.Resize((img_size, img_size)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

# 0.9 / 0.1
full_train_dataset = GarbageFineDataset(train_root, transform=train_transform)

full_train_for_val = GarbageFineDataset(
    train_root,
    transform=val_test_transform,
    fine_class_to_idx=full_train_dataset.fine_class_to_idx,
    big_class_to_idx=full_train_dataset.big_class_to_idx,
    strict=True
)

seed = 42
g = torch.Generator().manual_seed(seed)
```

```python
N = len(full_train_dataset)
n_train = int(0.9 * N)
n_val = N - n_train
train_subset, val_subset = random_split(range(N), [n_train, n_val], generator=g)


class IndexSubset(Dataset):
    def __init__(self, base_dataset, indices):
        self.base = base_dataset
        self.indices = list(indices)
    def __len__(self):
        return len(self.indices)
    def __getitem__(self, i):
        return self.base[self.indices[i]]

train_dataset = IndexSubset(full_train_dataset, train_subset)
val_dataset   = IndexSubset(full_train_for_val, val_subset)

test_dataset = GarbageFineDataset(
    test_root,
    transform=val_test_transform,
    fine_class_to_idx=full_train_dataset.fine_class_to_idx,
    big_class_to_idx=full_train_dataset.big_class_to_idx,
    strict=True
)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=num_workers)
val_loader   = DataLoader(val_dataset,   batch_size=batch_size, shuffle=False, num_workers=num_workers)
test_loader  = DataLoader(test_dataset,  batch_size=batch_size, shuffle=False, num_workers=num_workers)

fine_idx_to_name = full_train_dataset.fine_idx_to_name
big_idx_to_name  = full_train_dataset.big_idx_to_name
fine_to_big      = full_train_dataset.fine_to_big

print("Big classes:", big_idx_to_name)
print("Fine classes (examples):", list(fine_idx_to_name.items())[:11])
```

```
[/content/garbage_train] samples=20371, big=4, fine=16
[/content/garbage_train] samples=20371, big=4, fine=16
[/content/garbage_test] samples=251, big=4, fine=16
Big classes: {0: 'recycling', 1: 'special', 2: 'trash', 3: 'yard_waste'}
Fine classes (examples): [(0, 'recycling/cardboard'), (1, 'recycling/glass'),
(2, 'recycling/metal'), (3, 'recycling/paper'), (4, 'recycling/plastic'), (5,
'special/battery'), (6, 'special/cables'), (7, 'special/keyboard'), (8,
'special/mouse'), (9, 'special/tire'), (10, 'trash/biological')]
```

2 1. SVM

[7]:

```
svm_transform = transforms.Compose([
    transforms.Resize((96, 96)),
    transforms.ToTensor(),
])
svm_full_train = GarbageFineDataset(
    train_root,
    transform=svm_transform,
    fine_class_to_idx=full_train_dataset.fine_class_to_idx,
    big_class_to_idx=full_train_dataset.big_class_to_idx,
    strict=True
)
svm_train_dataset = IndexSubset(svm_full_train, train_subset)
svm_val_dataset   = IndexSubset(svm_full_train, val_subset)

svm_test_dataset = GarbageFineDataset(
    test_root,
    transform=svm_transform,
    fine_class_to_idx=full_train_dataset.fine_class_to_idx,
    big_class_to_idx=full_train_dataset.big_class_to_idx,
    strict=True
)

svm_batch_size = 64
svm_train_loader = DataLoader(svm_train_dataset, batch_size=svm_batch_size,␣
 ↪shuffle=False, num_workers=num_workers)
svm_val_loader   = DataLoader(svm_val_dataset,   batch_size=svm_batch_size,␣
 ↪shuffle=False, num_workers=num_workers)
svm_test_loader  = DataLoader(svm_test_dataset,  batch_size=svm_batch_size,␣
 ↪shuffle=False, num_workers=num_workers)



@torch.no_grad()
def extract_flat_features(loader):
    feats, labels = [], []
    for images, fine_labels, _ in loader:
        flat = images.view(images.size(0), -1)
        feats.append(flat.cpu().numpy())
        labels.append(fine_labels.numpy())
    return np.concatenate(feats), np.concatenate(labels)
```

```
svm_train_feats, svm_train_labels = extract_flat_features(svm_train_loader)
svm_val_feats, svm_val_labels = extract_flat_features(svm_val_loader)
svm_test_feats, svm_test_labels = extract_flat_features(svm_test_loader)

scaler = StandardScaler()
pca_components = 256
svm_train_feats = scaler.fit_transform(svm_train_feats)
svm_val_feats = scaler.transform(svm_val_feats)
svm_test_feats = scaler.transform(svm_test_feats)

pca = PCA(n_components=pca_components, random_state=42)
svm_train_feats = pca.fit_transform(svm_train_feats)
svm_val_feats = pca.transform(svm_val_feats)
svm_test_feats = pca.transform(svm_test_feats)

svm_clf = SGDClassifier(
    loss="hinge",
    alpha=1e-5,
    max_iter=500,
    tol=1e-4,
    n_jobs=-1,
    class_weight="balanced",
    random_state=42,
    verbose=0,
)
t0 = time.time()
svm_clf.fit(svm_train_feats, svm_train_labels)
svm_val_pred = svm_clf.predict(svm_val_feats)
svm_test_pred = svm_clf.predict(svm_test_feats)

svm_val_acc = (svm_val_pred == svm_val_labels).mean()
svm_test_acc = (svm_test_pred == svm_test_labels).mean()

print(f"[SVM] Val acc={svm_val_acc:.4f} | Test acc={svm_test_acc:.4f}")
```

```
[/content/garbage_train] samples=20371, big=4, fine=16
[/content/garbage_test] samples=251, big=4, fine=16
[SVM] Val acc=0.3734 | Test acc=0.1195
```

3 2. CNN

[8]:

```
num_fine_classes = len(full_train_dataset.fine_class_to_idx)
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.AdaptiveAvgPool2d((1, 1)),
        )
        self.classifier = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        return self.classifier(x)


def train_cnn_one_epoch(model, loader, optimizer, criterion, device):
    model.train()
    total_loss, correct, total = 0.0, 0, 0
    for images, fine_labels, _ in loader:
        images = images.to(device)
        fine_labels = fine_labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, fine_labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * images.size(0)
        pred = outputs.argmax(dim=1)
        correct += (pred == fine_labels).sum().item()
        total += fine_labels.size(0)
    return total_loss / total, correct / total


@torch.no_grad()
def eval_cnn_one_epoch(model, loader, criterion, device):
    model.eval()
    total_loss, correct, total = 0.0, 0, 0
    for images, fine_labels, _ in loader:
        images = images.to(device)
        fine_labels = fine_labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, fine_labels)

        total_loss += loss.item() * images.size(0)
        pred = outputs.argmax(dim=1)
        correct += (pred == fine_labels).sum().item()
        total += fine_labels.size(0)
    return total_loss / total, correct / total


cnn_model = SimpleCNN(num_fine_classes).to(device)
cnn_criterion = nn.CrossEntropyLoss()
cnn_optimizer = optim.Adam(cnn_model.parameters(), lr=1e-3)
cnn_epochs = 10
cnn_best_acc = 0.0
cnn_best_state = None

for epoch in range(cnn_epochs):
    tr_loss, tr_acc = train_cnn_one_epoch(cnn_model, train_loader,␣
 ↪cnn_optimizer, cnn_criterion, device)
    va_loss, va_acc = eval_cnn_one_epoch(cnn_model, val_loader, cnn_criterion,␣
 ↪device)
    print(f"[CNN] Epoch {epoch+1}/{cnn_epochs} | Train loss={tr_loss:.4f},␣
 ↪acc={tr_acc:.4f} | Val loss={va_loss:.4f}, acc={va_acc:.4f}")
    if va_acc > cnn_best_acc:
        cnn_best_acc = va_acc
        cnn_best_state = {k: v.cpu() for k, v in cnn_model.state_dict().items()}

if cnn_best_acc > 0:
    cnn_model.load_state_dict(cnn_best_state)
    cnn_model = cnn_model.to(device)
print("[CNN] Best val acc:", cnn_best_acc)
```

```
[CNN] Epoch 1/10 | Train loss=1.8990, acc=0.3824 | Val loss=1.7342, acc=0.4642
[CNN] Epoch 2/10 | Train loss=1.7634, acc=0.4262 | Val loss=1.6741, acc=0.4769
[CNN] Epoch 3/10 | Train loss=1.7054, acc=0.4435 | Val loss=1.6038, acc=0.5020
[CNN] Epoch 4/10 | Train loss=1.6543, acc=0.4612 | Val loss=1.5930, acc=0.5079
[CNN] Epoch 5/10 | Train loss=1.6063, acc=0.4768 | Val loss=1.6048, acc=0.4902
[CNN] Epoch 6/10 | Train loss=1.5679, acc=0.4903 | Val loss=1.4820, acc=0.5206
[CNN] Epoch 7/10 | Train loss=1.5332, acc=0.5009 | Val loss=1.4561, acc=0.5402
[CNN] Epoch 8/10 | Train loss=1.5196, acc=0.5062 | Val loss=1.3990, acc=0.5559
[CNN] Epoch 9/10 | Train loss=1.4680, acc=0.5202 | Val loss=1.4069, acc=0.5648
[CNN] Epoch 10/10 | Train loss=1.4526, acc=0.5232 | Val loss=1.2763, acc=0.5942
[CNN] Best val acc: 0.5942100098135427
```

4 3. Fine tuned Resnet 18

Stage 1: Freeze the backbone and only train the last layer

[9]:

```
weights = models.ResNet18_Weights.IMAGENET1K_V1
model = models.resnet18(weights=weights)

for p in model.parameters():
    p.requires_grad = False

num_features = model.fc.in_features
num_fine_classes = len(full_train_dataset.fine_class_to_idx)
model.fc = nn.Linear(num_features, num_fine_classes)

model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to
/root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
```

```
100%|          | 44.7M/44.7M [00:00<00:00, 97.2MB/s]
```

[10]:

```
def train_one_epoch(model, loader, optimizer, criterion, device):
    model.train()
    total_loss, correct, total = 0.0, 0, 0

    for images, fine_labels, _ in loader:
        images = images.to(device)
        fine_labels = fine_labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, fine_labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * images.size(0)
        pred = outputs.argmax(dim=1)
        correct += (pred == fine_labels).sum().item()
        total += fine_labels.size(0)

    return total_loss / total, correct / total

@torch.no_grad()
def eval_one_epoch(model, loader, criterion, device):
    model.eval()
    total_loss, correct, total = 0.0, 0, 0

    for images, fine_labels, _ in loader:
        images = images.to(device)
        fine_labels = fine_labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, fine_labels)

        total_loss += loss.item() * images.size(0)
        pred = outputs.argmax(dim=1)
        correct += (pred == fine_labels).sum().item()
        total += fine_labels.size(0)

    return total_loss / total, correct / total
```

[12]:

```
num_epochs = 10
best_val_acc = 0.0
best_state = None

for epoch in range(num_epochs):
    tr_loss, tr_acc = train_one_epoch(model, train_loader, optimizer,␣
 ↪criterion, device)
    va_loss, va_acc = eval_one_epoch(model, val_loader, criterion, device)

    print(f"Epoch {epoch+1}/{num_epochs} | "
          f"Train loss={tr_loss:.4f}, acc={tr_acc:.4f} | "
          f"Val loss={va_loss:.4f}, acc={va_acc:.4f}")

    if va_acc > best_val_acc:
        best_val_acc = va_acc
        best_state = {k: v.cpu() for k, v in model.state_dict().items()}

if best_state is not None:
    model.load_state_dict(best_state)
    model = model.to(device)

print("Best val acc:", best_val_acc)
```

```
Epoch 1/10 | Train loss=0.8239, acc=0.7554 | Val loss=0.3957, acc=0.8817
Epoch 2/10 | Train loss=0.5305, acc=0.8326 | Val loss=0.3148, acc=0.9033
Epoch 3/10 | Train loss=0.4863, acc=0.8439 | Val loss=0.3107, acc=0.9028
Epoch 4/10 | Train loss=0.4674, acc=0.8491 | Val loss=0.2777, acc=0.9136
Epoch 5/10 | Train loss=0.4524, acc=0.8536 | Val loss=0.2988, acc=0.9048
Epoch 6/10 | Train loss=0.4460, acc=0.8553 | Val loss=0.2949, acc=0.9063
Epoch 7/10 | Train loss=0.4397, acc=0.8573 | Val loss=0.2955, acc=0.9097
Epoch 8/10 | Train loss=0.4334, acc=0.8598 | Val loss=0.2877, acc=0.9112
Epoch 9/10 | Train loss=0.4326, acc=0.8582 | Val loss=0.2906, acc=0.9024
Epoch 10/10 | Train loss=0.4291, acc=0.8612 | Val loss=0.3078, acc=0.9048
Best val acc: 0.9136408243375859
```

Stage 2: Unfreeze layer 4 and fc

[13]:

```
for name, p in model.named_parameters():
    p.requires_grad = name.startswith("layer4") or name.startswith("fc")

optimizer = optim.AdamW([
    {"params": model.layer4.parameters(), "lr": 1e-4},
    {"params": model.fc.parameters(),     "lr": 5e-4},
], weight_decay=1e-4)

scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=5)

num_epochs_ft = 5
best_val_acc_ft = 0.0
best_state_ft = None

for epoch in range(num_epochs_ft):
    tr_loss, tr_acc = train_one_epoch(model, train_loader, optimizer,␣
 ↪criterion, device)
    va_loss, va_acc = eval_one_epoch(model, val_loader, criterion, device)

    print(f"[FT] Epoch {epoch+1}/{num_epochs_ft} | "
          f"Train loss={tr_loss:.4f}, acc={tr_acc:.4f} | "
          f"Val loss={va_loss:.4f}, acc={va_acc:.4f}", flush=True)

    scheduler.step()

    if va_acc > best_val_acc_ft:
        best_val_acc_ft = va_acc
        best_state_ft = {k: v.cpu() for k, v in model.state_dict().items()}

if best_state_ft is not None:
    model.load_state_dict(best_state_ft)
    model = model.to(device)

print("Best FT val acc:", best_val_acc_ft)
```

```
[FT] Epoch 1/5 | Train loss=0.4423, acc=0.8595 | Val loss=0.2472, acc=0.9249
[FT] Epoch 2/5 | Train loss=0.3057, acc=0.9002 | Val loss=0.2236, acc=0.9338
[FT] Epoch 3/5 | Train loss=0.2463, acc=0.9201 | Val loss=0.1948, acc=0.9460
[FT] Epoch 4/5 | Train loss=0.1929, acc=0.9364 | Val loss=0.1782, acc=0.9460
[FT] Epoch 5/5 | Train loss=0.1688, acc=0.9443 | Val loss=0.1680, acc=0.9509
Best FT val acc: 0.9509322865554465
```

5 4. Fine tuned Mobile Net V3

[14]:

```
mb_weights = models.MobileNet_V3_Large_Weights.IMAGENET1K_V1
mobile_model = models.mobilenet_v3_large(weights=mb_weights)
mobile_model.classifier[-1] = nn.Linear(mobile_model.classifier[-1].
 ↪in_features, num_fine_classes)

for p in mobile_model.features.parameters():
    p.requires_grad = False

mobile_model = mobile_model.to(device)
mobile_criterion = nn.CrossEntropyLoss()
mobile_optimizer = optim.Adam(mobile_model.classifier.parameters(), lr=1e-3)
mobile_epochs = 10
mobile_best_acc = 0.0
mobile_best_state = None

for epoch in range(mobile_epochs):
    tr_loss, tr_acc = train_one_epoch(mobile_model, train_loader,␣
 ↪mobile_optimizer, mobile_criterion, device)
    va_loss, va_acc = eval_one_epoch(mobile_model, val_loader,␣
 ↪mobile_criterion, device)
    print(f"[MobileNet] Epoch {epoch+1}/{mobile_epochs} | Train loss={tr_loss:.
 ↪4f}, acc={tr_acc:.4f} | Val loss={va_loss:.4f}, acc={va_acc:.4f}")
    if va_acc > mobile_best_acc:
        mobile_best_acc = va_acc
        mobile_best_state = {k: v.cpu() for k, v in mobile_model.state_dict().
 ↪items()}

if mobile_best_state is not None:
    mobile_model.load_state_dict(mobile_best_state)
    mobile_model = mobile_model.to(device)
print("[MobileNet] Best val acc:", mobile_best_acc)

# fine-tune: unfreeze backbone with smaller LR for a few epochs
for p in mobile_model.features.parameters():
    p.requires_grad = True

mobile_ft_optimizer = optim.AdamW([
    {"params": mobile_model.features.parameters(), "lr": 3e-5, "weight_decay":␣
 ↪1e-4},
    {"params": mobile_model.classifier.parameters(), "lr": 3e-4, "weight_decay":
 ↪ 1e-4},
])
mobile_ft_epochs = 5
mobile_ft_best_acc = 0.0
mobile_ft_best_state = None
```

```python
for epoch in range(mobile_ft_epochs):
    tr_loss, tr_acc = train_one_epoch(mobile_model, train_loader, mobile_ft_optimizer, mobile_criterion, device)
    va_loss, va_acc = eval_one_epoch(mobile_model, val_loader, mobile_criterion, device)
    print(f"[MobileNet-FT] Epoch {epoch+1}/{mobile_ft_epochs} | Train loss={tr_loss:.4f}, acc={tr_acc:.4f} | Val loss={va_loss:.4f}, acc={va_acc:.4f}")
    if va_acc > mobile_ft_best_acc:
        mobile_ft_best_acc = va_acc
        mobile_ft_best_state = {k: v.cpu() for k, v in mobile_model.state_dict().items()}

if mobile_ft_best_state is not None:
    mobile_model.load_state_dict(mobile_ft_best_state)
    mobile_model = mobile_model.to(device)
print("[MobileNet-FT] Best val acc:", mobile_ft_best_acc)
```

```
Downloading:
"https://download.pytorch.org/models/mobilenet_v3_large-8738ca79.pth" to
/root/.cache/torch/hub/checkpoints/mobilenet_v3_large-8738ca79.pth
```

```
100%|          | 21.1M/21.1M [00:00<00:00, 70.7MB/s]
```

```
[MobileNet] Epoch 1/10 | Train loss=0.6210, acc=0.8090 | Val loss=0.3032,
acc=0.9053
[MobileNet] Epoch 2/10 | Train loss=0.4433, acc=0.8573 | Val loss=0.2660,
acc=0.9171
[MobileNet] Epoch 3/10 | Train loss=0.4037, acc=0.8709 | Val loss=0.2370,
acc=0.9303
[MobileNet] Epoch 4/10 | Train loss=0.3640, acc=0.8824 | Val loss=0.2605,
acc=0.9190
[MobileNet] Epoch 5/10 | Train loss=0.3515, acc=0.8888 | Val loss=0.2331,
acc=0.9323
[MobileNet] Epoch 6/10 | Train loss=0.3385, acc=0.8909 | Val loss=0.2535,
acc=0.9195
[MobileNet] Epoch 7/10 | Train loss=0.3201, acc=0.8978 | Val loss=0.2629,
acc=0.9195
[MobileNet] Epoch 8/10 | Train loss=0.3129, acc=0.8996 | Val loss=0.2550,
acc=0.9220
[MobileNet] Epoch 9/10 | Train loss=0.3127, acc=0.9023 | Val loss=0.2167,
acc=0.9342
[MobileNet] Epoch 10/10 | Train loss=0.2988, acc=0.9058 | Val loss=0.2332,
acc=0.9347
[MobileNet] Best val acc: 0.9347399411187438
[MobileNet-FT] Epoch 1/5 | Train loss=0.2264, acc=0.9279 | Val loss=0.1811,
acc=0.9534
```

```
[MobileNet-FT] Epoch 2/5 | Train loss=0.1710, acc=0.9435 | Val loss=0.1731,
acc=0.9509
[MobileNet-FT] Epoch 3/5 | Train loss=0.1692, acc=0.9464 | Val loss=0.1521,
acc=0.9588
[MobileNet-FT] Epoch 4/5 | Train loss=0.1400, acc=0.9555 | Val loss=0.1504,
acc=0.9598
[MobileNet-FT] Epoch 5/5 | Train loss=0.1349, acc=0.9561 | Val loss=0.1582,
acc=0.9578
[MobileNet-FT] Best val acc: 0.9597644749754661
```

6 5. Strategies evaluation and comparison

[15]:

```python
import numpy as np
import torch
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
from matplotlib.colors import PowerNorm, LogNorm
import matplotlib.pyplot as plt
import numpy as np

@torch.no_grad()
def collect_fine_preds(model, loader, device):
    model.eval()
    y_true, y_pred = [], []

    for images, fine_labels, _ in loader:
        images = images.to(device)
        outputs = model(images)
        pred = outputs.argmax(dim=1).cpu().numpy()

        y_true.extend(fine_labels.numpy())
        y_pred.extend(pred)

    return np.array(y_true), np.array(y_pred)


def _plot_confusion_matrix(cm, class_names, title, normalize=False,
                           cmap="Blues", use_log_for_counts=True,␣
 ↪vmax_percentile=99):

    cm_plot = cm.astype(np.float64)

    if normalize:
        row_sums = cm_plot.sum(axis=1, keepdims=True)
        cm_plot = np.divide(cm_plot, row_sums, out=np.zeros_like(cm_plot),␣
 ↪where=row_sums != 0) * 100.0
```

```
    n = len(class_names)

    fig_w = min(18, max(10, 0.65 * n))
    fig_h = min(18, max(7,  0.60 * n))
    plt.figure(figsize=(fig_w, fig_h))

    if normalize:

        vmax = 100.0
        norm = PowerNorm(gamma=0.5, vmin=0.0, vmax=vmax)
        im = plt.imshow(cm_plot, interpolation="nearest", cmap=cmap, norm=norm)
    else:

        nonzero = cm_plot[cm_plot > 0]
        vmax = np.percentile(nonzero, vmax_percentile) if nonzero.size else 1.0

        if use_log_for_counts:

            masked = np.ma.masked_where(cm_plot == 0, cm_plot)
            norm = LogNorm(vmin=1, vmax=max(1, vmax))
            im = plt.imshow(masked, interpolation="nearest", cmap=cmap, norm=norm)
        else:

            norm = PowerNorm(gamma=0.5, vmin=0.0, vmax=vmax)
            im = plt.imshow(cm_plot, interpolation="nearest", cmap=cmap, norm=norm)

    plt.title(title)
    plt.colorbar(im, fraction=0.046, pad=0.04)

    tick_marks = np.arange(n)
    plt.xticks(tick_marks, class_names, rotation=45, ha="right", fontsize=9)
    plt.yticks(tick_marks, class_names, fontsize=9)


    display_max = np.nanmax(cm_plot) if normalize else (np.nanmax(cm_plot[cm_plot > 0]) if np.any(cm_plot > 0) else 1.0)
    thresh = display_max * 0.5

    for i in range(n):
        for j in range(n):
            val = cm_plot[i, j]
            if normalize:
                text = f"{val:.1f}"
                show = (val > 0)
            else:
                text = str(int(cm[i, j]))
                show = (cm[i, j] > 0)

            if show:
                plt.text(j, i, text,
                         ha="center", va="center",
                         fontsize=8,
                         color="white" if val > thresh else "black")

    plt.ylabel("True label")
    plt.xlabel("Predicted label")
    plt.tight_layout()
    plt.show()


def evaluate_report(y_fine_true, y_fine_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name=""):

    fine_names = [fine_idx_to_name[i] for i in range(len(fine_idx_to_name))]

    print(f"\n=== {set_name} Fine-class Report  ===")
    print(classification_report(y_fine_true, y_fine_pred,␣
 ↪target_names=fine_names, zero_division=0))

    cm_fine = confusion_matrix(y_fine_true, y_fine_pred, labels=np.
 ↪arange(len(fine_names)))
    _plot_confusion_matrix(cm_fine, fine_names, title=f"{set_name} Fine␣
 ↪Confusion Matrix (Counts)", normalize=False)
    _plot_confusion_matrix(cm_fine, fine_names, title=f"{set_name} Fine␣
 ↪Confusion Matrix (Row %)", normalize=True)

    y_big_true = np.array([fine_to_big[int(i)] for i in y_fine_true])
    y_big_pred = np.array([fine_to_big[int(i)] for i in y_fine_pred])

    big_names = [big_idx_to_name[i] for i in range(len(big_idx_to_name))]

    print(f"\n=== {set_name} Big-class Report  ===")
    print(classification_report(y_big_true, y_big_pred, target_names=big_names,␣
 ↪zero_division=0))

    cm_big = confusion_matrix(y_big_true, y_big_pred, labels=np.
 ↪arange(len(big_names)))
    _plot_confusion_matrix(cm_big, big_names, title=f"{set_name} Big Confusion␣
 ↪Matrix (Counts)", normalize=False)
```

```
    _plot_confusion_matrix(cm_big, big_names, title=f"{set_name} Big Confusion␣
 ↪Matrix (Row %)", normalize=True)

print("\n######## SVM VAL SET EVAL ########")
evaluate_report(svm_val_labels, svm_val_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="SVM VAL")

print("\n######## SVM TEST SET EVAL ########")
evaluate_report(svm_test_labels, svm_test_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="SVM TEST")

print("\n######## CNN VAL SET EVAL ########")
y_cnn_val_true, y_cnn_val_pred = collect_fine_preds(cnn_model, val_loader,␣
 ↪device)
evaluate_report(y_cnn_val_true, y_cnn_val_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="CNN VAL")

print("\n######## CNN TEST SET EVAL ########")
y_cnn_test_true, y_cnn_test_pred = collect_fine_preds(cnn_model, test_loader,␣
 ↪device)
evaluate_report(y_cnn_test_true, y_cnn_test_pred, fine_idx_to_name,␣
 ↪fine_to_big, big_idx_to_name, set_name="CNN TEST")

y_val_true, y_val_pred = collect_fine_preds(model, val_loader, device)
print("\n######## ResNet VAL SET EVAL ########")
evaluate_report(y_val_true, y_val_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="VAL")

y_test_true, y_test_pred = collect_fine_preds(model, test_loader, device)
print("\n######## ResNet TEST SET EVAL ########")
evaluate_report(y_test_true, y_test_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="TEST")

print("\n######## MobileNet VAL SET EVAL ########")
y_mb_val_true, y_mb_val_pred = collect_fine_preds(mobile_model, val_loader,␣
 ↪device)
evaluate_report(y_mb_val_true, y_mb_val_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="MobileNet VAL")

print("\n######## MobileNet TEST SET EVAL ########")
y_mb_test_true, y_mb_test_pred = collect_fine_preds(mobile_model, test_loader,␣
 ↪device)
evaluate_report(y_mb_test_true, y_mb_test_pred, fine_idx_to_name, fine_to_big,␣
 ↪big_idx_to_name, set_name="MobileNet TEST")
```

```
Output hidden; open in https://colab.research.google.com to view.
```

```
[ ]: # from https://gist.github.com/jonathanagustin/b67b97ef12c53a8dec27b343dca4abba
     # install can take a minute

     import os
     # @title Convert Notebook to PDF. Save Notebook to given directory
     NOTEBOOKS_DIR = "/content/drive/My Drive/CS441/Final" # @param {type:"string"}
     NOTEBOOK_NAME = "Imagenet.ipynb" # @param {type:"string"}
     #------------------------------------------------------------------------------#
     from google.colab import drive
     drive.mount("/content/drive/", force_remount=True)
     NOTEBOOK_PATH = f"{NOTEBOOKS_DIR}/{NOTEBOOK_NAME}"
     assert os.path.exists(NOTEBOOK_PATH), f"NOTEBOOK NOT FOUND: {NOTEBOOK_PATH}"
     !apt install -y texlive-xetex texlive-fonts-recommended texlive-plain-generic >␣
      ↪/dev/null 2>&1
     !jupyter nbconvert "$NOTEBOOK_PATH" --to pdf > /dev/null 2>&1
     NOTEBOOK_PDF = NOTEBOOK_PATH.rsplit('.', 1)[0] + '.pdf'
     assert os.path.exists(NOTEBOOK_PDF), f"ERROR MAKING PDF: {NOTEBOOK_PDF}"
     print(f"PDF CREATED: {NOTEBOOK_PDF}")
```

```
Mounted at /content/drive/
```