Clean cache (data from earlier runs)

```
# === Clean all old model artifacts ===
# This will remove checkpoints, logs, and any saved weights
# so that training starts from scratch.

import shutil
import os

# Common folders where models/logs are usually stored
folders_to_remove = [
    "/content/checkpoints",   # custom checkpoints
    "/content/logs",          # training logs (tensorboard, etc.)
    "/content/models",        # saved models
    "/content/runs",          # often used by torch or tensorboard
]
for folder in folders_to_remove:
    shutil.rmtree(folder, ignore_errors=True)
    print(f"[OK] Removed (if existed): {folder}")

# If you save weights with specific extensions (like .pth, .pt, .h5) in /content
# you can also wipe them all:
for f in os.listdir("/content"):
    if f.endswith((".pth", ".pt", ".h5", ".ckpt")):
        try:
            os.remove(os.path.join("/content", f))
            print(f"[OK] Removed file: {f}")
        except Exception as e:
            print(f"Could not remove {f}: {e}")

print("All old model files cleared. Training will start from scratch.")
```

```
[OK] Removed (if existed): /content/checkpoints
[OK] Removed (if existed): /content/logs
[OK] Removed (if existed): /content/models
[OK] Removed (if existed): /content/runs
All old model files cleared. Training will start from scratch.
```

```
# === Full cleanup of Colab working dirs ===
import shutil

folders_to_remove = [
    "/content/kagglehub_cache_1758281629",
    "/content/kagglehub_cache_1758281860",
    "/content/leaf_disease_data",
    "/content/sample_data"
]

for folder in folders_to_remove:
    shutil.rmtree(folder, ignore_errors=True)
    print(f"[OK] Removed: {folder}")

print(" Workspace cleaned. Now you have a fresh Colab environment (except Google Drive).")
```

```
[OK] Removed: /content/kagglehub_cache_1758281629
[OK] Removed: /content/kagglehub_cache_1758281860
[OK] Removed: /content/leaf_disease_data
[OK] Removed: /content/sample_data
 Workspace cleaned. Now you have a fresh Colab environment (except Google Drive).
```

```
# === Purge local Colab cache and (optionally) a dataset folder on Google Drive ===
# Run this BEFORE any import of kagglehub.

import os, shutil

# 1) Remove local kagglehub cache inside the Colab VM
local_cache = os.path.expanduser("~/.cache/kagglehub")
shutil.rmtree(local_cache, ignore_errors=True)
print(f"[OK] Removed local cache: {local_cache}")

# 2) (Optional) If you stored a copy on Google Drive, remove that too
#    Change this path if your dataset sits elsewhere in Drive.
drive_dataset_dir = "/content/drive/MyDrive/leaf-disease-dataset-combination"
shutil.rmtree(drive_dataset_dir, ignore_errors=True)
print(f"[OK] Removed Drive dataset folder (if exist `): {drive_dataset_dir}")
```

```
[OK] Removed local cache: /root/.cache/kagglehub
[OK] Removed Drive dataset folder (if existed): /content/drive/MyDrive/leaf-disease-dataset-combination
```

```
# === Force a fresh download with an isolated, empty cache dir ===
# IMPORTANT: set env vars BEFORE importing kagglehub.

import os, time
from pathlib import Path

# Put kagglehub cache into a fresh, run-specific folder (so nothing is reused)
run_cache = Path(f"/content/kagglehub_cache_{int(time.time())}")
os.environ["XDG_CACHE_HOME"] = str(run_cache)    # kagglehub honors ~/.cache via XDG_CACHE_HOME
print("[INFO] Using isolated cache dir:", run_cache)

import kagglehub   # import AFTER setting XDG_CACHE_HOME

DATASET_ID = "asheniranga/leaf-disease-dataset-combination"

# Force download to bypass any previous cache
data_path = kagglehub.dataset_download(DATASET_ID, force_download=True)

print("[OK] Fresh dataset downloaded to:", data_path)
print("[CHECK] Cache root exists:", run_cache.exists())
print("[CHECK] Cache dir content entries:", len(list(run_cache.rglob('*'))))
```

```
[INFO] Using isolated cache dir: /content/kagglehub_cache_1759067419
Downloading from https://www.kaggle.com/api/v1/datasets/download/asheniranga/leaf-disease-dataset-combination?datase
100%|██████████| 761M/761M [00:26<00:00, 30.3MB/s]Extracting files...

[OK] Fresh dataset downloaded to: /root/.cache/kagglehub/datasets/asheniranga/leaf-disease-dataset-combination/versi
[CHECK] Cache root exists: False
[CHECK] Cache dir content entries: 0
```

Import data, and GPU setup

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
import numpy as np
from collections import defaultdict

# Check GPU availability
import tensorflow as tf
print("GPU Available:", tf.config.list_physical_devices('GPU'))
if tf.config.list_physical_devices('GPU'):
    print(" GPU detected and ready!")
else:
    print(" Using CPU")

# Simple setup
plt.style.use('default')
sns.set_palette("Set2")
```

```
GPU Available: [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
 GPU detected and ready!
```

Libraries and data import

```
from sklearn.datasets import fetch_openml
from sklearn import __version__ as skver
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, log_loss
from sklearn.preprocessing import Normalizer, LabelEncoder
from sklearn import metrics
import random, glob, os
from pathlib import Path
from IPython import get_ipython
from IPython.display import display
import kagglehub
```

```
import os
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter, defaultdict
import cv2
from PIL import Image
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

# Setting up for beautiful graphs
plt.style.use('default')
sns.set_palette("husl")
```

```
from google.colab import drive
drive.mount('/content/drive')


from google.colab import drive
drive.mount('/content/drive')
zip_path = "/content/drive/MyDrive/ColabNotebooks/LeavesPhotoDF/leaf_disease.zip"

import zipfile
import os

extract_path = "/content/leaf_disease_data"
os.makedirs(extract_path, exist_ok=True)

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

data_path = Path(extract_path)

print("Data path set to:", data_path)
```

```
Mounted at /content/drive
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_re
Data path set to: /content/leaf_disease_data
```

ANALYSIS OF DATASET STRUCTURE

↳ 10 клітинок приховано

Data cleaning and preperation

↳ 16 клітинок приховано

Weights and lost funcktions

↳ 7 клітинок приховано

Train

CNN(convolution neyro networkong)

Two experiments on 50% subset, 7 epochs, A) Houdini-style margin loss B) CrossEntropyLoss + class_weights CrossEntropyLoss mesure distanse bitween two probability distribution between real probabbility destribution: p and predicted probobility distrebiton of model: q, $H(p,q) = -\Sigma p(x) \log q(x)$.

In most cases we dont need masure spcifik wites for Houdini because the Houdini weiths depends on the difference between the correct class and the "strongest" class

```
# ================== UNIVERSAL BENCHMARK (plant vs disease) ==================
# Compares Houdini vs CrossEntropy(+class weights) on chosen label granularity.
# - Pluggable metrics: pass a list of callables (y_true, y_pred, y_proba, classes)->dict
# - Supports user-provided class_weights dicts {class_name: weight}
# - Keeps the best epoch by val macro-F1; returns predictions/probas for later analysis.
```

```
import os, time, copy, math, warnings
from pathlib import Path
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, Dataset, Subset
from torchvision import transforms, models
from PIL import Image
from sklearn.metrics import f1_score, classification_report, confusion_matrix

# ---------------- User inputs ----------------
ROOT = Path(data_path) / "image data" if (Path(data_path) / "image data").exists() else Path(data_path)

SUBSAMPLE_FRAC = 0.10     # exact 20% of train
EPOCHS         = 4       # keep tag text consistent below
BATCH_SIZE     = 64
NUM_WORKERS    = 2
LR             = 3e-4
WEIGHT_DECAY   = 1e-4
RANDOM_SEED    = 42
USE_REAL_HOUDINI = True   # if you already defined global houdini_loss(logits,y)

torch.manual_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Device:", device)

# ---------------- Per-image standardization ----------------
def per_image_standardization_torch(x: torch.Tensor) -> torch.Tensor:
    """Standardize a single image tensor channel-wise after scaling to [0,255]."""
    x255 = x * 255.0
    mean = x255.mean()
    var  = x255.var(unbiased=False)
    std  = torch.sqrt(var + 1e-12)
    adjusted_std = torch.maximum(std, torch.tensor(1.0 / math.sqrt(x255.numel()), device=x.device, dtype=x.dtype))
    return (x255 - mean) / adjusted_std

class PathsDataset(Dataset):
    """Lightweight dataset built from arrays of paths and numeric labels."""
    def __init__(self, paths_np, labels_np, resize_hw=(256,256), augment=False):
        self.paths_np = paths_np
        self.labels_np = labels_np
        self.resize = transforms.Resize(resize_hw, interpolation=transforms.InterpolationMode.BILINEAR)
        self.augs = None
        if augment:
            self.augs = transforms.RandomOrder([
                transforms.RandomHorizontalFlip(),
                transforms.RandomVerticalFlip(),
                transforms.RandomRotation(15),
                transforms.ColorJitter(0.2,0.2,0.2,0.03),
            ])
    def __len__(self): return len(self.paths_np)
    def __getitem__(self,i):
        p = self.paths_np[i]; y = int(self.labels_np[i])
        img = Image.open(p).convert("RGB")
        img = self.resize(img)
        x = transforms.functional.to_tensor(img)
        if self.augs is not None:
            x_pil = transforms.functional.to_pil_image(x)
            x_pil = self.augs(x_pil)
            x = transforms.functional.to_tensor(x_pil)
        x = per_image_standardization_torch(x)
        return x, y

# ---------------- Split collection with label_mode ----------------
EXTS = {'.jpg','.jpeg','.png','.bmp','.webp','.tif','.tiff'}

def collect_split(split, label_mode="plant"):
    """
    label_mode='plant'   -> label = plant_dir.name
    label_mode='disease' -> label = disease_dir.name  (merge same-named diseases)
    """
    assert label_mode in {"plant","disease"}
    paths, labels = [], []
    split_dir = ROOT / split
    if not split_dir.exists():
        return paths, labels
    for plant_dir in sorted(p for p in split_dir.iterdir() if p.is_dir()):
        plant = plant_dir.name
        for disease_dir in sorted(p for p in plant_dir.iterdir() if p.is_dir()):
            disease = disease_dir.name
```

```
            for f in disease_dir.rglob("*"):
                if f.is_file() and f.suffix.lower() in EXTS:
                    paths.append(str(f))
                    labels.append(plant if label_mode=="plant" else disease)
    return paths, labels

def make_loaders(label_mode="plant", augment_train=False):
    """Build datasets/dataloaders + numeric encodings for a chosen label granularity."""
    tr_p, tr_l_str = collect_split("train", label_mode)
    va_p, va_l_str = collect_split("validation", label_mode)
    te_p, te_l_str = collect_split("test", label_mode)

    classes = sorted(set(tr_l_str + va_l_str + te_l_str))
    label_to_index = {c:i for i,c in enumerate(classes)}
    tr_l = np.array([label_to_index[s] for s in tr_l_str], dtype=np.int64)
    va_l = np.array([label_to_index[s] for s in va_l_str], dtype=np.int64)
    te_l = np.array([label_to_index[s] for s in te_l_str], dtype=np.int64)

    tr_p = np.array(tr_p); va_p = np.array(va_p); te_p = np.array(te_p)

    train_ds_full = PathsDataset(tr_p, tr_l, augment=augment_train)
    val_ds        = PathsDataset(va_p, va_l)
    test_ds       = PathsDataset(te_p, te_l) if len(te_p) else None

    # exact 20% subset
    n_train = len(train_ds_full)
    k = max(1, int(n_train * SUBSAMPLE_FRAC))
    idx = np.arange(n_train)
    rng = np.random.default_rng(RANDOM_SEED); rng.shuffle(idx)
    train_indices = idx[:k]
    train_ds = Subset(train_ds_full, train_indices)

    pin_mem = (device.type=='cuda')
    train_dl = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True,  num_workers=NUM_WORKERS, pin_memory=pin_m
    val_dl   = DataLoader(val_ds,   batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS, pin_memory=pin_m
    test_dl  = DataLoader(test_ds,  batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS, pin_memory=pin_m

    return dict(
        classes=classes, C=len(classes),
        train_ds_full=train_ds_full, train_ds=train_ds, train_indices=train_indices,
        val_ds=val_ds, test_ds=test_ds,
        train_dl=train_dl, val_dl=val_dl, test_dl=test_dl
    )

# ---------------- Model / metrics / loss helpers ----------------
def make_resnet18(num_classes:int)->nn.Module:
    """ResNet-18 backbone with a light dropout on the head."""
    m = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
    m.fc = nn.Sequential(nn.Dropout(0.2), nn.Linear(m.fc.in_features, num_classes))
    return m.to(device)

@torch.no_grad()
def eval_split(model, dl, criterion=None, need_probs:bool=True):
    """Evaluate on a dataloader; optionally collect probabilities for extra metrics."""
    model.eval()
    tot=corr=0; loss_sum=0.0
    y_true=[]; y_pred=[]; y_prob=[]
    for x,y in dl:
        x,y = x.to(device), y.to(device)
        logits = model(x)
        if criterion is not None:
            loss_sum += criterion(logits,y).item()*y.size(0)
        p = logits.argmax(1)
        corr += (p==y).sum().item(); tot += y.size(0)
        y_true.append(y.cpu().numpy()); y_pred.append(p.cpu().numpy())
        if need_probs:
            y_prob.append(torch.softmax(logits, dim=1).cpu().numpy())
    if tot==0:
        return dict(
            loss=0.0, acc=0.0, f1_macro=0.0, f1_weighted=0.0,
            y_true=np.array([]), y_pred=np.array([]), y_proba=np.array([])
        )
    y_true = np.concatenate(y_true); y_pred = np.concatenate(y_pred)
    y_proba = np.concatenate(y_prob) if (need_probs and len(y_prob)) else None
    acc = corr/tot
    f1m = f1_score(y_true,y_pred,average='macro',zero_division=0)
    f1w = f1_score(y_true,y_pred,average='weighted',zero_division=0)
    loss_avg = loss_sum/tot if criterion is not None else 0.0
    return dict(loss=loss_avg, acc=acc, f1_macro=f1m, f1_weighted=f1w,
                y_true=y_true, y_pred=y_pred, y_proba=y_proba)

def align_weights_from_dict(classes, weights_dict:dict|None):
```

```
    """
    Align a user-provided dict {class_name: weight} to a torch tensor in `classes` order.
    Warn if there are missing or extra keys. If missing, fallback to 1.0.
    """
    if not weights_dict:
        return None
    miss = [c for c in classes if c not in weights_dict]
    extra = [k for k in weights_dict.keys() if k not in classes]
    if miss:
        warnings.warn(f"[class weights] missing classes: {miss}; using 1.0 for them")
    if extra:
        warnings.warn(f"[class weights] extra keys not in classes: {extra} (ignored)")
    arr = np.array([weights_dict.get(c, 1.0) for c in classes], dtype=np.float32)
    return torch.tensor(arr, device=device)
"""
class HoudiniMarginLoss(nn.Module):
    #Self-contained Houdini-style margin loss (if you don't use your own)
    def __init__(self, tau: float = 1.0):
        super().__init__(); self.tau=tau; self.sigmoid=nn.Sigmoid()
    def forward(self, logits, y):
        s_true = logits.gather(1, y.view(-1,1))
        mask = torch.ones_like(logits, dtype=torch.bool); mask.scatter_(1, y.view(-1,1), False)
        max_other = logits.masked_fill(~mask, float('-inf')).amax(dim=1, keepdim=True)
        margin = (max_other - s_true)/self.tau
        return self.sigmoid(margin).mean()
"""
def run_experiment_pair(label_mode="plant",
                        class_weights_dict:dict|None=None,
                        metrics_fns:list|None=None,
                        augment_train:bool=False):
    """
    Run two training loops on the same split/subset:
      - 'houdini' : your global houdini_loss if available (or local fallback)
      - 'cew'     : CrossEntropy with aligned class weights (if provided)
    metrics_fns: list of callables (y_true, y_pred, y_proba, classes) -> dict
    Returns a dict with per-mode summaries, predictions, and models.
    """
    # loaders
    bundle = make_loaders(label_mode=label_mode, augment_train=augment_train)
    classes = bundle["classes"]; C=bundle["C"]
    val_dl=bundle["val_dl"]; test_dl=bundle["test_dl"]

    # class weights for CE aligned to class order
    CE_WEIGHTS = align_weights_from_dict(classes, class_weights_dict)

    results = {}

    for mode in ("houdini","cew"):
        model = make_resnet18(C)
        optimizer = torch.optim.AdamW(model.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
    mode='max',
    factor=0.5,
    patience=4,
    min_lr=1e-6,
    #verbose=True
)
        scaler = torch.cuda.amp.GradScaler(enabled=(device.type=='cuda'))
        pct = int(SUBSAMPLE_FRAC * 100)
        if mode=="houdini":
            if USE_REAL_HOUDINI and ('houdini_loss' in globals()):
                def criterion(logits,y): return houdini_loss(logits,y)  # your custom
            else:
                criterion = HoudiniMarginLoss(tau=1.0)
            tag = f"{label_mode.upper()} | Houdini_{pct}p_{EPOCHS}e"
        else:
            criterion = nn.CrossEntropyLoss(weight=CE_WEIGHTS) if CE_WEIGHTS is not None else nn.CrossEntropyLoss()
            tag = f"{label_mode.upper()} | CE+Weights_{pct}p_{EPOCHS}e" if CE_WEIGHTS is not None else f"{label_moc

        best_f1m=-1.0; best_state=None; best_epoch=-1

        def train_one_epoch():
            """One training epoch with AMP."""
            model.train()
            tot=corr=0; loss_sum=0.0
            for x,y in bundle["train_dl"]:
                x,y = x.to(device), y.to(device)
                optimizer.zero_grad(set_to_none=True)
                with torch.cuda.amp.autocast(enabled=(device.type=='cuda')):
                    logits = model(x); loss = criterion(logits,y)
                scaler.scale(loss).backward()
```

```
                scaler.step(optimizer); scaler.update()
                with torch.no_grad():
                    p = logits.argmax(1); corr += (p==y).sum().item(); tot += y.size(0); loss_sum += loss.item()*y.
            return (loss_sum/max(1,tot), corr/max(1,tot))

        # --- train loop
        for epoch in range(1,EPOCHS+1):
            t0=time.time()
            tr_loss, tr_acc = train_one_epoch()
            val_stats = eval_split(model, val_dl, criterion, need_probs=True)
            # после val_stats = eval_split(...)
            scheduler.step(val_stats['f1_macro'])  # для mode='max'

            print(f"[{tag}] epoch {epoch:02d} | train acc {tr_acc:.3f} loss {tr_loss:.3f} | "
                  f"val acc {val_stats['acc']:.3f} f1m {val_stats['f1_macro']:.3f} loss {val_stats['loss']:.3f} | t
            if val_stats['f1_macro'] > best_f1m:
                best_f1m = float(val_stats['f1_macro']); best_epoch=epoch; best_state=copy.deepcopy(model.state_dic

        if best_state is not None:
            model.load_state_dict(best_state)

        # --- final evals
        val_stats  = eval_split(model, val_dl,  criterion, need_probs=True)
        test_stats = eval_split(model, test_dl, criterion, need_probs=True) if test_dl is not None else None

        # --- built-in prints
        print("\nValidation classification report:")
        print(classification_report(val_stats['y_true'], val_stats['y_pred'], target_names=classes, zero_division=0
        if test_stats is not None:
            print("Test classification report:")
            print(classification_report(test_stats['y_true'], test_stats['y_pred'], target_names=classes, zero_divi

        # --- user metrics (plug-in)
        extra_val_metrics = {}
        extra_test_metrics = {}
        if metrics_fns:
            for fn in metrics_fns:
                try:
                    extra_val_metrics |= fn(val_stats['y_true'], val_stats['y_pred'], val_stats['y_proba'], classes
                    if test_stats is not None:
                        extra_test_metrics |= fn(test_stats['y_true'], test_stats['y_pred'], test_stats['y_proba'],
                except Exception as e:
                    warnings.warn(f"[metrics] '{getattr(fn,'__name__',str(fn))}' failed: {e}")

        summary = dict(
            label_mode=label_mode, tag=tag, best_epoch=best_epoch,
            final_val_f1_macro=float(val_stats['f1_macro']),
            final_val_loss=float(val_stats['loss']),
            final_val_acc=float(val_stats['acc']),
            final_test_f1_macro=float(test_stats['f1_macro']) if test_stats is not None else float('nan'),
            final_test_acc=float(test_stats['acc']) if test_stats is not None else float('nan'),
            extra_val_metrics=extra_val_metrics,
            extra_test_metrics=extra_test_metrics
        )

        results[mode] = dict(
            summary=summary,
            model=model,
            classes=classes,
            val_stats=val_stats,
            test_stats=test_stats
        )

    # comparison print
    print(f"\n=== COMPARISON [{label_mode.upper()}] (20% / {EPOCHS} epochs) ===")
    h = results["houdini"]["summary"]; c = results["cew"]["summary"]
    show = ['best_epoch','final_val_f1_macro','final_val_loss','final_val_acc','final_test_f1_macro','final_test_ac
    print("Houdini: ", {k:h[k] for k in show})
    print("CE+Wght:", {k:c[k] for k in show})
    return results

# ---------------- Example: wrap your custom metrics ----------------
# Define adapters for your already-written metrics. Each should return a dict.
def metric_confmat(y_true, y_pred, y_proba, classes):
    """Return confusion matrix as a flattened dict for logging (optional)."""
    cm = confusion_matrix(y_true, y_pred, labels=np.arange(len(classes)))
    return {"confusion_matrix": cm.tolist()}  # keep it JSON-serializable

# If you have your own functions, pass them here (example):
# metrics_list = [your_metric_fn1, your_metric_fn2, metric_confmat]
metrics_list = [metric_confmat]
```

```
# ---------------- RUNS ----------------
# IMPORTANT: pass the corresponding weights dict for each label_mode.
#   - class_weights_plants:   {plant_name: weight}
#   - class_weights_diseases: {disease_name: weight}
#
# Example (uncomment when you have both dicts in scope):
res_plant   = run_experiment_pair(label_mode="plant",
                                  class_weights_dict=class_weights_plants,
                                  metrics_fns=metrics_list,
                                  augment_train=False)
res_disease = run_experiment_pair(label_mode="disease",
                                  class_weights_dict=class_weights_disease,
                                  metrics_fns=metrics_list,
                                  augment_train=False)
```

```
           corn (maize)       1.00      0.91      0.95       280
                  grape       0.98      1.00      0.99       296
                  peach       1.00      0.92      0.96       192
           pepper, bell       1.00      0.99      0.99       180
                 potato       0.96      0.97      0.96       155
             strawberry       0.98      0.94      0.96       114
                 tomato       0.98      0.99      0.98      1317

               accuracy                           0.98      3417
              macro avg       0.98      0.97      0.97      3417
           weighted avg       0.98      0.98      0.98      3417

[PLANT | CE+Weights_10p_4e] epoch 01 | train acc 0.904 loss 0.335 | val acc 0.954 f1m 0.945 loss 0.138 | time 9.8s
[PLANT | CE+Weights_10p_4e] epoch 02 | train acc 0.992 loss 0.034 | val acc 0.989 f1m 0.985 loss 0.032 | time 9.8s
[PLANT | CE+Weights_10p_4e] epoch 03 | train acc 0.997 loss 0.016 | val acc 0.985 f1m 0.978 loss 0.049 | time 9.9s
[PLANT | CE+Weights_10p_4e] epoch 04 | train acc 0.999 loss 0.006 | val acc 0.993 f1m 0.990 loss 0.028 | time 9.8s

Validation classification report:
                         precision    recall  f1-score   support

                Cassava       1.00      1.00      1.00       257
                   Rice       1.00      1.00      1.00       240
                  apple       0.99      0.98      0.98       226
cherry (including sour)       0.99      0.98      0.99       136
           corn (maize)       0.98      1.00      0.99       276
                  grape       0.99      1.00      1.00       291
                  peach       0.97      0.98      0.98       191
           pepper, bell       0.99      1.00      1.00       177
                 potato       0.97      0.99      0.98       155
             strawberry       1.00      0.97      0.99       112
                 tomato       1.00      0.99      1.00      1303

               accuracy                           0.99      3364
              macro avg       0.99      0.99      0.99      3364
           weighted avg       0.99      0.99      0.99      3364

Test classification report:
                         precision    recall  f1-score   support

                Cassava       1.00      1.00      1.00       265
                   Rice       1.00      1.00      1.00       246
                  apple       0.98      0.97      0.98       232
cherry (including sour)       0.95      0.99      0.97       140
           corn (maize)       0.99      1.00      0.99       280
                  grape       0.99      1.00      0.99       296
                  peach       1.00      0.96      0.98       192
           pepper, bell       0.99      0.97      0.98       180
                 potato       0.93      0.99      0.96       155
             strawberry       1.00      0.97      0.99       114
                 tomato       1.00      0.99      0.99      1317

               accuracy                           0.99      3417
              macro avg       0.98      0.99      0.98      3417
           weighted avg       0.99      0.99      0.99      3417


=== COMPARISON [PLANT] (20% / 4 epochs) ===
Houdini:  {'best_epoch': 3, 'final_val_f1_macro': 0.9734772690221412, 'final_val_loss': 0.02394630381666515, 'fina
```

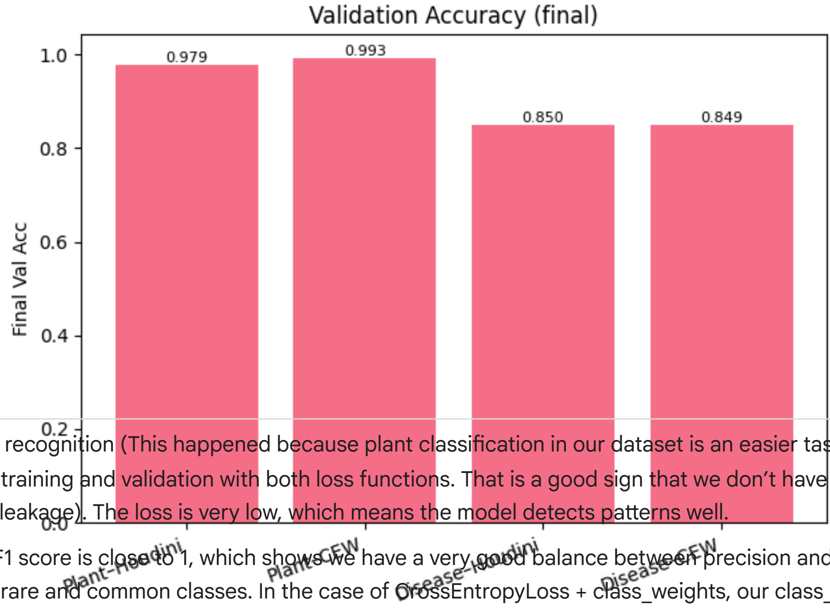
Visualisation

```
# ================== PLOTS ONLY (no training) ==================
# This cell expects the variables `res_plant` and `res_disease` to exist in scope.
# It DOES NOT train anything — it only visualizes existing results.

import numpy as np
import matplotlib.pyplot as plt

# ---- Utilities to safely extract histories (if present) ----
def _has_history(res):
```

```
        """Return True if both modes contain 'history' with per-epoch dicts."""
        try:
            for mode in ("houdini", "cew"):
                h = res[mode].get("history", None)
                if not isinstance(h, list) or len(h) == 0:
                    return False
                # Check that the first item looks like a per-epoch dict
                if not isinstance(h[0], dict) or "epoch" not in h[0]:
                    return False
            return True
        except Exception:
            return False

    def _series_from(res, label_prefix: str):
        """
        Convert results[mode]['history'] into a dict of metric-> {label: (epochs, values)}.
        Expected metric keys in history: 'train_acc','train_loss','val_acc','val_f1m','val_loss'
        """
        out = {}
        for mode, nice in (("houdini", f"{label_prefix}-Houdini"),
                           ("cew",     f"{label_prefix}-CEW")):
            hist = res[mode]["history"]
            epochs = [d.get("epoch", i+1) for i,d in enumerate(hist)]
            for key_src, key_dst in (
                ("train_acc", "train_acc"),
                ("train_loss","train_loss"),
                ("val_acc",   "val_acc"),
                ("val_f1m",   "val_f1m"),
                ("val_loss",  "val_loss"),
            ):
                vals = [d[key_src] for d in hist if key_src in d]
                if len(vals) == len(epochs):
                    out.setdefault(key_dst, {})
                    out[key_dst][nice] = (epochs, vals)
        return out

    def _plot_lines(metric_title: str, key: str, plant_series: dict, disease_series: dict):
        """Draw a line plot with 4 lines: Plant-Houdini, Plant-CEW, Disease-Houdini, Disease-CEW."""
        plt.figure()
        plotted = 0
        for label in ("Plant-Houdini","Plant-CEW","Disease-Houdini","Disease-CEW"):
            src = plant_series if label.startswith("Plant") else disease_series
            if key in src and label in src[key]:
                xs, ys = src[key][label]
                plt.plot(xs, ys, marker='o', label=label)
                plotted += 1
        plt.xlabel("Epoch")
        plt.ylabel(metric_title.split()[-1] if "Loss" not in metric_title else "Loss")
        plt.title(metric_title)
        plt.grid(True, alpha=0.3)
        if plotted:
            plt.legend()
        else:
            plt.text(0.5, 0.5, "No per-epoch history found", ha='center', va='center', transform=plt.gca().transAxes)
        plt.tight_layout()
        plt.show()

    # ---- Fallback: bar charts from summaries (no history available) ----
    def _bar_from_summaries(res_plant, res_disease, title: str, extract_key: str):
        """
        Make a 4-bar comparison (Plant-Houdini, Plant-CEW, Disease-Houdini, Disease-CEW)
        using values from results[mode]['summary'][extract_key].
        """
        labels = ["Plant-Houdini","Plant-CEW","Disease-Houdini","Disease-CEW"]
        values = []
        for res in (res_plant, res_plant, res_disease, res_disease):
            mode = "houdini" if len(values) % 2 == 0 else "cew"
            val = float(res[mode]["summary"].get(extract_key, np.nan))
            values.append(val)

        x = np.arange(len(labels))
        plt.figure()
        plt.bar(x, values)
        plt.xticks(x, labels, rotation=20, ha='right')
        plt.title(title)
        plt.ylabel(extract_key.replace("_", " ").title())
        for xi, v in zip(x, values):
            plt.text(xi, v, f"{v:.3f}" if np.isfinite(v) else "NA", ha='center', va='bottom', fontsize=8)
        plt.tight_layout()
        plt.show()

    # ---- Decide which path to take (with or without history) ----
```

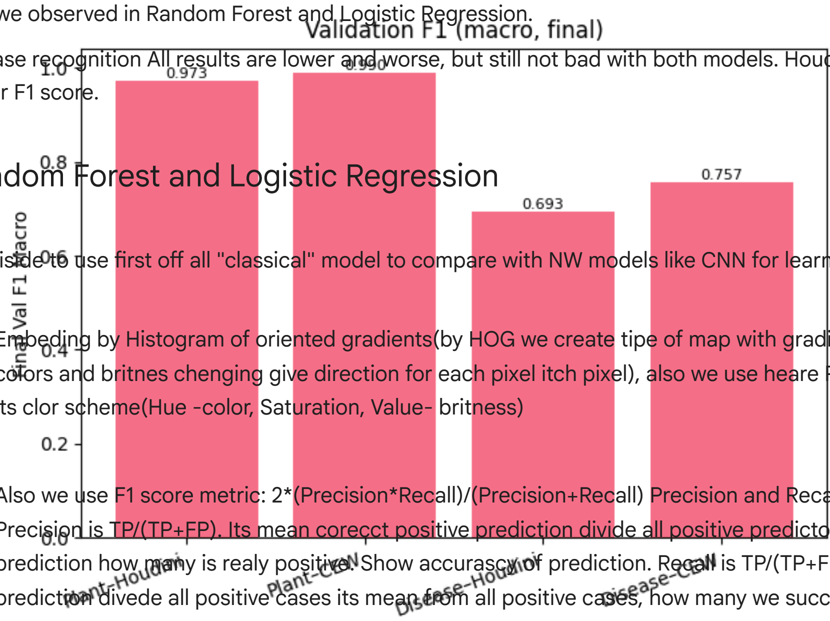
```
    plant_has_hist   = _has_history(res_plant)
    disease_has_hist = _has_history(res_disease)
    both_have_hist   = plant_has_hist and disease_has_hist

    if both_have_hist:
        # ------------- Line plots: 4 charts × 4 lines -------------
        plant_series   = _series_from(res_plant,   "Plant")
        disease_series = _series_from(res_disease, "Disease")

        # 1) Train Accuracy
        _plot_lines("Train Accuracy", "train_acc", plant_series, disease_series)
        # 2) Train Loss
        _plot_lines("Train Loss", "train_loss", plant_series, disease_series)
        # 3) Validation Accuracy
        _plot_lines("Validation Accuracy", "val_acc", plant_series, disease_series)
        # 4) Validation F1 (macro)
        _plot_lines("Validation F1 (macro)", "val_f1m", plant_series, disease_series)

    else:
        # ------------- Fallback (no history): bar comparisons by summaries -------------
        # We can't reconstruct per-epoch curves. Show final comparisons instead.
        print("[info] No per-epoch history found in results. Showing final-summary comparisons only.")
        # 1) Validation Accuracy
        _bar_from_summaries(res_plant, res_disease, "Validation Accuracy (final)", "final_val_acc")
        # 2) Validation F1 (macro)
        _bar_from_summaries(res_plant, res_disease, "Validation F1 (macro, final)", "final_val_f1_macro")
        # 3) Validation Loss
        _bar_from_summaries(res_plant, res_disease, "Validation Loss (final)", "final_val_loss")
        # 4) Test Accuracy (if NaN for some mode, it'll show NA)
        _bar_from_summaries(res_plant, res_disease, "Test Accuracy (final)", "final_test_acc")
```

```
[info] No per-epoch history found in results. Showing final-summary comparisons only.
```



Plant recognition (This happened because plant classification in our dataset is an easier task.) The model shows very high accuracy on both training and validation with both loss functions. That is a good sign that we don't have overfitting (we also verified that there is no data leakage). The loss is very low, which means the model detects patterns well.

The F1 score is close to 1, which shows we have a very good balance between precision and recall. This means the model works well with both rare and common classes. In the case of CrossEntropyLoss + class_weights, our class_weights were sufficient to fix the imbalance that we observed in Random Forest and Logistic Regression.



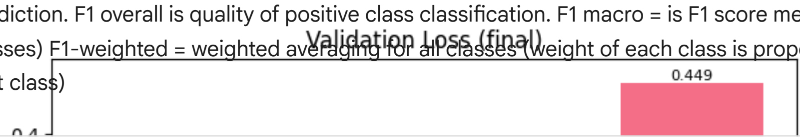
Disease recognition All results are lower and worse, but still not bad with both models. Houdini achieved a better loss, but CE+W gave a better F1 score.

Random Forest and Logistic Regression

We diside to use first off all "classical" model to compare with NW models like CNN for learning resons

> Embeding by Histogram of oriented gradients(by HOG we create tipe of map with gradient directions thet distrebiut by colors and britnes chenging give direction for each pixel itch pixel), also we use heare PCA for redusing dimantion. And HSV its clor scheme(Hue -color, Saturation, Value- britness)

> Also we use F1 score metric: 2*(Precision*Recall)/(Precision+Recall) Precision and Recall we got from confusion matrix Precision is TP/(TP+FP). Its mean corecct positive prediction divide all positive predicton, its mean from all positive prediction how many is realy positive. Show accurascy of prediction. Recall is TP/(TP+FN). Its mean correct positive prediction divede all positive cases its mean from all positive cases, how many we successfully found. Show sensitivity of prediction. F1 overall is quality of positive class classification. F1 macro = is F1 score mean for all classes (same waite for all classes) F1-weighted = weighted averaging for all classes (weight of each class is proportional to the number of examples in that class)




```
# === Slim HOG+PCA baseline (end-to-end, robust to (plant,disease) dual labels) ===
# All comments are in English, per your request.

import numpy as np, random, math, warnings
import matplotlib.pyplot as plt
from skimage import feature
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, classification_report, confusion_matrix, ConfusionMatrixDispl
from sklearn.exceptions import ConvergenceWarning

# Optional frameworks (the code works even if they're missing)
try:
    import torch
except Exception:
    torch = None
try:
    import tensorflow as tf
except Exception:
    tf = None

# ----------------------------
# Config
# ----------------------------
TRAIN_FRACTION     = 0.20          # fraction of train_ds to subsample for PCA/train
```

```
RANDOM_SEED        = 42
N_COMPONENTS       = 156
USE_RANDOM_FOREST  = True

HOG_PARAMS = dict(
    orientations=9,
    pixels_per_cell=(16, 16),
    cells_per_block=(2, 2),
    block_norm='L2-Hys',
    feature_vector=True
)

rng = np.random.RandomState(RANDOM_SEED)
random.seed(RANDOM_SEED)
if torch is not None:
    try: torch.manual_seed(RANDOM_SEED)
    except: pass

warnings.filterwarnings("ignore", category=ConvergenceWarning)

# Optional: define class_weights externally as a dict {class_id: weight} or None.
# Example:
# class_weights = {0: 1.0, 1: 2.0, 2: 1.3}
# If not defined, it will be ignored.
if 'class_weights' not in globals():
    class_weights = None


# ----------------------------
# Minimal helpers (framework-agnostic)
# ----------------------------
def _to_numpy(x):
    """Convert torch/tf/PIL/array-like to numpy array."""
    if torch is not None and isinstance(x, getattr(torch, "Tensor", ())):
        return x.detach().cpu().numpy()
    if tf is not None and isinstance(x, getattr(tf, "Tensor", ())):
        return x.numpy()
    try:
        from PIL import Image as _PILImage
        if isinstance(x, _PILImage.Image):
            return np.asarray(x)
    except Exception:
        pass
    return np.asarray(x)

def _hwc_from_any(img):
    """Ensure image shape is HWC or HW (with channel axis last)."""
    arr = _to_numpy(img)
    if arr.ndim == 2:
        return arr[..., None]
    if arr.ndim == 3:
        # Convert CHW to HWC if needed
        if arr.shape[0] in (1,3) and arr.shape[0] < arr.shape[-1]:
            return np.transpose(arr, (1,2,0))  # (C,H,W)->(H,W,C)
        return arr
    raise ValueError(f"Unsupported image shape: {arr.shape}")

def batch_hog(images):
    """Compute HOG features for a batch of images (HWC or HW)."""
    arr = _to_numpy(images)
    if arr.ndim == 2:
        arr = arr[None, ..., None]  # -> (1,H,W,1)
    elif arr.ndim == 3:
        arr = arr[None, ...]        # -> (1,H,W,C)
    elif arr.ndim != 4:
        raise ValueError(f"Expected 2D/3D/4D, got {arr.shape}")

    feats = []
    for i in range(arr.shape[0]):
        img = arr[i]
        if img.ndim == 3 and img.shape[-1] > 1:
            gray = np.mean(img.astype(np.float64), axis=-1)
        elif img.ndim == 3:
            gray = img[..., 0].astype(np.float64)
        else:
            gray = img.astype(np.float64)
        feats.append(feature.hog(gray, **HOG_PARAMS).astype(np.float32))
    return np.asarray(feats, dtype=np.float32)

def _is_indexable_dataset(ds):
    """Return True if ds supports __len__ and __getitem__."""
    return hasattr(ds, '__len__') and hasattr(ds, '__getitem__')
```

```
def _unwrap_example(example):
    """Extract (image, label) from common dataset formats."""
    if isinstance(example, (tuple, list)) and len(example) >= 2:
        return example[0], example[1]
    if isinstance(example, dict):
        for ki in ('image','img','x','features'):
            for kl in ('label','y','target','labels','targets'):
                if ki in example and kl in example:
                    return example[ki], example[kl]
    raise ValueError("Provide (image,label) or dict with image/label.")

def _split_dataset(ds, val_frac=0.15, test_frac=0.15, seed=RANDOM_SEED):
    """Split indexable dataset into (train,val,test) subsets."""
    if not _is_indexable_dataset(ds):
        raise ValueError("Need an indexable dataset (has __len__ and __getitem__). "
                         "If you have DataLoaders or tf.data, pass val/test explicitly.")
    n = len(ds)
    idx = np.arange(n)
    local_rng = np.random.RandomState(seed)
    local_rng.shuffle(idx)
    n_test = int(round(n * test_frac))
    n_val  = int(round(n * val_frac))
    test_idx = idx[:n_test]
    val_idx  = idx[n_test:n_test+n_val]
    train_idx = idx[n_test+n_val:]

    # Try torch Subset; else use a lightweight fallback
    try:
        import torch.utils.data as _tud
        return (_tud.Subset(ds, train_idx), _tud.Subset(ds, val_idx), _tud.Subset(ds, test_idx))
    except Exception:
        class _Subset:
            def __init__(self, base, indices): self.base, self.indices = base, list(indices)
            def __len__(self): return len(self.indices)
            def __getitem__(self, i): return self.base[self.indices[i]]
        base = ds
        return (_Subset(base, train_idx), _Subset(base, val_idx), _Subset(base, test_idx))

def _select_label(lab, label_mode):
    """
    Return a single integer label from many possible shapes:
      - scalar
      - 1D vector with >=2 elements (index 0=plant, 1=disease)
      - dict {'plant':..., 'disease':...}
      - tuple/list of two scalars (plant, disease)
    """
    # tuple/list of two scalars
    if isinstance(lab, (tuple, list)) and len(lab) >= 2 and not hasattr(lab[0], "__len__"):
        return int(lab[0] if label_mode == 'plant' else lab[1])

    if isinstance(lab, dict):
        key = 'plant' if label_mode == 'plant' else 'disease'
        return int(_to_numpy(lab[key]))

    arr = _to_numpy(lab)
    if arr.ndim == 0:
        return int(arr)
    if arr.ndim == 1 and arr.size >= 2:
        idx = 0 if label_mode == 'plant' else 1
        return int(arr[idx])
    return int(arr.reshape(()))


def _labels_from_any(yb, label_mode):
    """
    Convert batched labels of many shapes to a 1D numpy int64 vector of length B.
    Supported inputs (batched):
      - yb shape (B,) : already single task
      - yb shape (B,2) : [:,0]=plant, [:,1]=disease
      - tuple/list (plant_batch, disease_batch), each shape (B,)
      - dict {'plant': batch, 'disease': batch}
    """
    # tuple/list of two batched tensors
    if isinstance(yb, (tuple, list)) and len(yb) >= 2 and hasattr(yb[0], "__len__"):
        a = _to_numpy(yb[0])
        b = _to_numpy(yb[1])
        y = a if label_mode == 'plant' else b
        return y.astype(np.int64).reshape(-1)

    if isinstance(yb, dict):
        key = 'plant' if label_mode == 'plant' else 'disease'
```

```
            y = _to_numpy(yb[key])
            return y.astype(np.int64).reshape(-1)

        arr = _to_numpy(yb)
        # (B,2) -> pick a column
        if arr.ndim == 2 and arr.shape[1] >= 2:
            idx = 0 if label_mode == 'plant' else 1
            arr = arr[:, idx]
        # (B,1) or other >1D → flatten to (B,)
        if arr.ndim > 1:
            arr = arr.reshape(-1)
        return arr.astype(np.int64)


    def collect_subset_hog(ds, take_fraction, rng, label_mode="plant"):
        """
        Collect a subsample of HOG features and labels from 'ds'.
        Works with indexable datasets and iterable/batched sources (DataLoader / tf.data).
        """
        X_list, y_list = [], []

        def _push(img, y_scalar):
            feat = batch_hog(_hwc_from_any(img)[None, ...])
            X_list.append(feat)
            y_list.append(np.int64(y_scalar).reshape(-1))

        if _is_indexable_dataset(ds):
            n = len(ds)
            k = max(1, int(n * take_fraction))
            for i in rng.choice(n, size=k, replace=False):
                img, lab = _unwrap_example(ds[i])
                y_scalar = _select_label(lab, label_mode)
                _push(img, y_scalar)
        else:
            target = 5000  # soft cap to avoid scanning entire infinite streams
            collected = 0
            for ex in ds:
                try:
                    xb, yb = _unwrap_example(ex)
                    xb = _to_numpy(xb)
                    # Normalize images to (B,H,W,C?)
                    if xb.ndim == 4 and xb.shape[1] in (1, 3) and xb.shape[1] < xb.shape[-1]:
                        xb = np.transpose(xb, (0, 2, 3, 1))  # CHW -> HWC
                    elif xb.ndim == 3:
                        xb = xb[None, ...]
                    if xb.ndim != 4:
                        continue

                    # Get a 1D label vector of length B
                    yb_vec = _labels_from_any(yb, label_mode)
                    B = xb.shape[0]
                    if yb_vec.shape[0] != B:
                        # Skip malformed batch
                        continue

                    for i in range(B):
                        if rng.rand() <= take_fraction:
                            _push(xb[i], yb_vec[i])
                            collected += 1
                            if collected >= target:
                                raise StopIteration
                except StopIteration:
                    break
                except Exception:
                    continue

        if not X_list:
            dummy = feature.hog(np.zeros((32,32), dtype=np.float64), **HOG_PARAMS)
            F = dummy.size
            return np.empty((0, F), np.float32), np.empty((0,), np.int64)

        X = np.concatenate(X_list, axis=0).astype(np.float32)
        y = np.concatenate(y_list, axis=0).astype(np.int64)
        return X, y


    def _iter_batches(dl, label_mode="plant"):
        """
        Yield (xb, yb) as numpy arrays:
          xb -> (B,H,W,C?), yb -> (B,)
        """
        for batch in dl:
```

```python
        xb, yb = _unwrap_example(batch)
        xb_np = _to_numpy(xb)
        # Normalize image layout to (B,H,W,C?)
        if xb_np.ndim == 4 and (xb_np.shape[1] in (1,3) and xb_np.shape[1] < xb_np.shape[-1]):
            xb_np = np.transpose(xb_np, (0, 2, 3, 1))
        elif xb_np.ndim == 3:
            xb_np = xb_np[None, ...]

        # Get (B,) labels robustly
        yb_np = _labels_from_any(yb, label_mode)

        # Final sanity
        if xb_np.ndim != 4 or yb_np.ndim != 1 or xb_np.shape[0] != yb_np.shape[0]:
            # Skip malformed batch
            continue

        yield xb_np, yb_np


def eval_stream(ds, pca, clf, name="split", label_mode="plant"):
    """
    Evaluate split by streaming batches: HOG -> PCA -> predict.
    Prints metrics and returns (acc, f1_macro, f1_weighted).
    """
    y_true_all, y_pred_all = [], []
    for xb, yb in _iter_batches(ds, label_mode=label_mode):
        Xb = batch_hog(xb)
        if Xb.shape[0] == 0:
            continue
        preds = clf.predict(pca.transform(Xb)).astype(np.int64)
        y_true_all.append(yb)
        y_pred_all.append(preds)

    if not y_true_all:
        print(f"[{name}] no data")
        return 0.0, 0.0, 0.0

    y_true = np.concatenate(y_true_all)
    y_pred = np.concatenate(y_pred_all)

    acc = accuracy_score(y_true, y_pred)
    f1m = f1_score(y_true, y_pred, average='macro',    zero_division=0)
    f1w = f1_score(y_true, y_pred, average='weighted', zero_division=0)

    print(f"[{name}] acc={acc:.4f}  f1_macro={f1m:.4f}  f1_weighted={f1w:.4f}")
    print(classification_report(y_true, y_pred, digits=3))
    return acc, f1m, f1w

def plot_confmat(y_true, y_pred, class_names, title, normalize='true'):
    """Utility to plot a confusion matrix."""
    cm = confusion_matrix(y_true, y_pred, normalize=normalize)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
    fig, ax = plt.subplots(figsize=(6.5,6))
    disp.plot(ax=ax, cmap='Blues', colorbar=True, values_format='.2f' if normalize else 'd')
    ax.set_title(title); plt.tight_layout(); plt.show()

def plot_f1_bars(y_true, y_pred, class_names, title):
    """Per-class F1 bar plot."""
    rep = classification_report(y_true, y_pred, output_dict=True, zero_division=0)
    f1s, labels = [], []
    for i, name in enumerate(class_names):
        key = str(i)
        if key in rep:
            f1s.append(rep[key]['f1-score']); labels.append(name)
    fig, ax = plt.subplots(figsize=(10,4.2))
    ax.bar(np.arange(len(f1s)), f1s)
    ax.set_xticks(np.arange(len(f1s))); ax.set_xticklabels(labels, rotation=45, ha='right')
    ax.set_ylim(0,1.0); ax.set_ylabel("F1"); ax.set_title(title)
    plt.tight_layout(); plt.show()


# ---------------------------
# Main runner (legacy signature)
# ---------------------------
def make_loaders(label_mode="plant", augment_train=False):
    """
    Legacy entrypoint kept for compatibility with your notebook.
    Expects the following globals to exist:
      - train_ds : training dataset or loader
      - val_ds   : validation dataset or loader (optional)
      - test_ds  : test dataset or loader (optional)
```

```
    If val_ds/test_ds are missing and train_ds is indexable, we will split it.
    """
    # Resolve datasets/loaders from globals
    if 'train_ds' not in globals():
        raise RuntimeError("Global 'train_ds' is required. Please define train_ds before calling make_loaders().")

    local_train = globals()['train_ds']
    local_val   = globals().get('val_ds', None)
    local_test  = globals().get('test_ds', None)

    # If val/test not provided: attempt to split
    if (local_val is None or local_test is None) and _is_indexable_dataset(local_train):
        local_train, local_val, local_test = _split_dataset(local_train, val_frac=0.15, test_frac=0.15, seed=RANDOM
    elif local_val is None or local_test is None:
        raise RuntimeError("val_ds/test_ds not found. Provide them or make train_ds indexable so we can split.")

    # 1) Subsample train for HOG+PCA fit + training
    X_train_raw, y_train = collect_subset_hog(local_train, TRAIN_FRACTION, rng, label_mode=label_mode)
    if X_train_raw.shape[0] == 0:
        raise RuntimeError("No training samples collected. Ensure your train_ds yields (image,label) and that TRAIN

    # 2) PCA on train subset
    pca = PCA(n_components=N_COMPONENTS, svd_solver='randomized', random_state=RANDOM_SEED)
    X_train_pca = pca.fit_transform(X_train_raw.astype(np.float32))

    # 3) Train baseline models
    lr = LogisticRegression(
        multi_class='multinomial', solver='saga', max_iter=2000, n_jobs=-1,
        class_weight=class_weights, verbose=0
    ).fit(X_train_pca, y_train)

    rf = None
    if USE_RANDOM_FOREST:
        rf = RandomForestClassifier(
            n_estimators=400, max_depth=None, n_jobs=-1,
            class_weight=class_weights, random_state=RANDOM_SEED
        ).fit(X_train_pca, y_train)

    # 4) Evaluate
    lr_val_acc,  lr_val_f1m,  lr_val_f1w = eval_stream(local_val,  pca, lr, "VAL / LogReg",  label_mode=label_mode)
    lr_test_acc, lr_test_f1m, lr_test_f1w = eval_stream(local_test, pca, lr, "TEST / LogReg", label_mode=label_mode

    if rf is not None:
        rf_val_acc,  rf_val_f1m,  rf_val_f1w = eval_stream(local_val,  pca, rf, "VAL / RF",  label_mode=label_mode)
        rf_test_acc, rf_test_f1m, rf_test_f1w = eval_stream(local_test, pca, rf, "TEST / RF", label_mode=label_mode

    # You can optionally return more (e.g., pca, models) if you want.
    return lr_test_f1m, lr_test_f1w


# ============================
# Example calls (keep as-is):
# ============================
# You must already have these in your environment:
#   train_ds, val_ds, test_ds
# If you don't have val_ds/test_ds, remove them and the code will split train_ds automatically.

print("Classic_plant")
classic_plant = make_loaders(label_mode="plant")

print("Classic_disease")
classic_disease = make_loaders(label_mode="disease")
```

```
weighted avg      0.497     0.454     0.382      3364

[TEST / RF] acc=0.4457  f1_macro=0.2554  f1_weighted=0.3787
              precision    recall  f1-score   support

           0      0.000     0.000     0.000        48
           1      0.353     0.211     0.264        57
           2      0.176     0.077     0.107        39
           3      0.500     0.021     0.040        48
           4      0.398     0.523     0.452       176
           5      0.000     0.000     0.000        42
           6      0.371     0.228     0.283        57
           7      1.000     0.023     0.044        44
           8      0.000     0.000     0.000        46
           9      0.409     0.621     0.493       393
          10      0.542     0.198     0.291       131
          11      0.000     0.000     0.000        21
          12      1.000     0.027     0.053        37
          13      0.944     0.977     0.960        87
          14      0.824     0.097     0.174       144
          15      0.685     0.604     0.642       101
          16      0.329     0.792     0.465       591
          17      0.550     0.210     0.303       210
          18      0.844     0.684     0.755        79
          19      0.857     0.086     0.156        70
          20      1.000     0.013     0.025        80
          21      0.636     0.296     0.404        71
          22      0.000     0.000     0.000        77
          23      1.000     0.008     0.015       129
          24      0.877     0.410     0.559       122
          25      0.000     0.000     0.000       102
          26      0.000     0.000     0.000        28
          27      0.553     0.840     0.667       387

    accuracy                          0.446      3417
   macro avg      0.495     0.248     0.255      3417
weighted avg      0.508     0.446     0.379      3417
```

```
import inspect
print(inspect.signature(make_loaders))
```

```
(label_mode='plant', augment_train=False)
```

Both Classical model show bed result in both cases. In disisise clasification the do not usefull at all

Second cheking og edgeliks by dublikates ore sami dublicater recognithion

↳ 4 клітинки приховано

Models Compitishion

Now we'll check other models, to compare our own with.

```
# ==========================================================
# Fair sweep on 20% TRAIN subset (no disk saves)
# - Works with label_mode in {"plant","disease"}
# - Plug-in metrics (your own metrics can be passed in)
# - Supports your class weight dicts {class_name: weight}
# - Uses prepared splits at ROOT/train|validation|test
# - TF-like per-image standardization (on 0..255 scale)
# - Auto image size per model (fixes DeiT 224 vs 256)
# - Colab-safe DataLoader (num_workers=0) to avoid MP assertion
# ==========================================================

# If needed in your environment:
# !pip -q install timm

from pathlib import Path
import os, gc, time, math, random, numpy as np, warnings
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, Dataset
from torchvision import transforms
from PIL import Image
from sklearn.metrics import accuracy_score, f1_score, classification_report, confusion_matrix
import timm

# -----------------------
```

```
# Config
# -----------------------
ROOT             = Path(data_path) / "image data" if (Path(data_path) / "image data").exists() else Path(data_path)
SUBSAMPLE_FRAC   = 0.10         # exact 20% of TRAIN
EPOCHS_SWEEP     = 4
BATCH_SIZE       = 64
NUM_WORKERS_SAFE = 0            # avoids Colab "can only test a child process" assertion
PIN_MEM          = (torch.cuda.is_available())
LR               = 3e-4
WEIGHT_DECAY     = 1e-4
RANDOM_SEED      = 42

# Strategy for CE weights when houdini_loss is NOT provided:
USE_TRAIN_BALANCED_WEIGHTS = False  # if False -> prefer your dict weights when provided

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
g = torch.Generator().manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED); np.random.seed(RANDOM_SEED); torch.manual_seed(RANDOM_SEED)

print("Device:", device)
print("Root:", str(ROOT))

# -----------------------
# TF-like per-image standardization (on 0..255 scale)
# -----------------------
def per_image_standardization_torch(x: torch.Tensor) -> torch.Tensor:
    """
    x: float tensor in [0,1], shape [3,H,W].
    Emulates tf.image.per_image_standardization(image * 255.0):
      - mean/std over ALL pixels and channels on 0..255 scale
      - adjusted_std = max(std, 1/sqrt(N))
      - output = (x*255 - mean) / adjusted_std
    """
    x255 = x * 255.0
    mean = x255.mean()
    var  = x255.var(unbiased=False)
    std  = torch.sqrt(var + 1e-12)
    adjusted_std = torch.maximum(std, torch.tensor(1.0 / math.sqrt(x255.numel()), device=x.device, dtype=x.dtype))
    return (x255 - mean) / adjusted_std

class PerImageStandardize(object):
    def __call__(self, x: torch.Tensor) -> torch.Tensor:
        return per_image_standardization_torch(x)

def build_transforms(img_size: int):
    """Build train/eval transforms for a given model's required img_size."""
    train_tfms = transforms.Compose([
        transforms.Resize((img_size, img_size), interpolation=transforms.InterpolationMode.BILINEAR),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.RandomRotation(15),
        transforms.ColorJitter(0.2,0.2,0.2,0.03),
        transforms.ToTensor(),
        PerImageStandardize(),  # TF-like standardization
    ])
    eval_tfms = transforms.Compose([
        transforms.Resize((img_size, img_size), interpolation=transforms.InterpolationMode.BILINEAR),
        transforms.ToTensor(),
        PerImageStandardize(),
    ])
    return train_tfms, eval_tfms

# -----------------------
# Dataset that supports label_mode={"plant","disease"}
# Expected tree: ROOT/split/PLANT/DISEASE/*.jpg
# -----------------------
EXTS = {'.jpg','.jpeg','.png','.bmp','.webp','.tif','.tiff'}

def collect_split(split, label_mode="plant"):
    """
    Build (paths, labels_str) for split.
    label_mode='plant'   -> label = plant_dir.name
    label_mode='disease' -> label = disease_dir.name (merged across plants)
    """
    assert label_mode in {"plant","disease"}
    paths, labels = [], []
    split_dir = ROOT / split
    if not split_dir.exists():
        return paths, labels
    for plant_dir in sorted(p for p in split_dir.iterdir() if p.is_dir()):
        plant = plant_dir.name
        for disease_dir in sorted(p for p in plant_dir.iterdir() if p.is_dir()):
```

```
            disease = disease_dir.name
            for f in disease_dir.rglob("*"):
                if f.is_file() and f.suffix.lower() in EXTS:
                    paths.append(str(f))
                    labels.append(plant if label_mode=="plant" else disease)
    return paths, labels

class PathsDataset(Dataset):
    """Lightweight dataset built from arrays of paths and numeric labels."""
    def __init__(self, paths_np, labels_np, transform):
        self.paths_np = paths_np
        self.labels_np = labels_np
        self.transform = transform
    def __len__(self): return len(self.paths_np)
    def __getitem__(self,i):
        p = self.paths_np[i]; y = int(self.labels_np[i])
        img = Image.open(p).convert("RGB")
        if self.transform is not None:
            img = self.transform(img)
        # transform ends with ToTensor + PerImageStandardize, so 'img' is torch.Tensor
        return img, y

# -----------------------
# Build base label space and stratified subset for TRAIN
# -----------------------
def make_base_and_subset(label_mode="plant"):
    """Collect splits, build class->index mapping, and create 20% stratified subset indices."""
    tr_p, tr_l_s = collect_split("train", label_mode)
    va_p, va_l_s = collect_split("validation", label_mode)
    te_p, te_l_s = collect_split("test", label_mode)

    classes = sorted(set(tr_l_s + va_l_s + te_l_s))
    lbl2idx = {c:i for i,c in enumerate(classes)}

    tr_y = np.array([lbl2idx[s] for s in tr_l_s], dtype=np.int64)
    va_y = np.array([lbl2idx[s] for s in va_l_s], dtype=np.int64)
    te_y = np.array([lbl2idx[s] for s in te_l_s], dtype=np.int64)

    tr_p = np.array(tr_p); va_p = np.array(va_p); te_p = np.array(te_p)

    # build stratified 20% from TRAIN ONLY
    by_c = {}
    for i,t in enumerate(tr_y): by_c.setdefault(int(t), []).append(i)
    rng = np.random.default_rng(RANDOM_SEED)
    subset_idx = []
    for idxs in by_c.values():
        k = max(1, int(round(len(idxs)*SUBSAMPLE_FRAC)))
        subset_idx.extend(rng.choice(idxs, size=k, replace=False).tolist())
    rng.shuffle(subset_idx)

    base = dict(
        classes=classes, C=len(classes),
        train_paths=tr_p, train_targets=tr_y,
        val_paths=va_p,   val_targets=va_y,
        test_paths=te_p,  test_targets=te_y,
        train_subset_idx=subset_idx
    )
    return base

# -----------------------
# Rebuild DataLoaders for a given img_size (per model)
# -----------------------
def build_dls_for_imgsize(img_size, base):
    tr_tfms, ev_tfms = build_transforms(img_size)

    train_full = PathsDataset(base['train_paths'], base['train_targets'], transform=tr_tfms)
    val_full   = PathsDataset(base['val_paths'],   base['val_targets'],   transform=ev_tfms)
    test_full  = PathsDataset(base['test_paths'],  base['test_targets'],  transform=ev_tfms)

    train_ds = Subset(train_full, base['train_subset_idx'])
    val_ds   = val_full
    test_ds  = test_full

    train_dl = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True,
                          num_workers=NUM_WORKERS_SAFE, pin_memory=PIN_MEM, generator=g)
    val_dl   = DataLoader(val_ds,   batch_size=BATCH_SIZE, shuffle=False,
                          num_workers=NUM_WORKERS_SAFE, pin_memory=PIN_MEM)
    test_dl  = DataLoader(test_ds,  batch_size=BATCH_SIZE, shuffle=False,
                          num_workers=NUM_WORKERS_SAFE, pin_memory=PIN_MEM)
    return train_dl, val_dl, test_dl

def infer_img_size_from_model(m):
```

```
    """Read target input size from timm model default_cfg; fallback to 224."""
    cfg = getattr(m, 'default_cfg', None) or {}
    inp = cfg.get('input_size', (3, 224, 224))
    return int(inp[1])

# -----------------------
# Weights alignment and criterion builder
# -----------------------
def align_weights_from_dict(classes, weights_dict:dict|None):
    """
    Align a user-provided dict {class_name: weight} to a torch tensor in `classes` order.
    Warn if there are missing or extra keys. If missing, fallback to 1.0.
    """
    if not weights_dict:
        return None
    miss = [c for c in classes if c not in weights_dict]
    extra = [k for k in weights_dict.keys() if k not in classes]
    if miss:
        warnings.warn(f"[class weights] missing classes: {miss}; using 1.0 for them")
    if extra:
        warnings.warn(f"[class weights] extra keys not in classes: {extra} (ignored)")
    arr = np.array([weights_dict.get(c, 1.0) for c in classes], dtype=np.float32)
    return torch.tensor(arr, device=device)

def build_ce_balanced_weights_from_subset(targets_full, subset_idx, C):
    """Compute classic balanced weights from TRAIN subset labels."""
    sub_targets = targets_full[subset_idx]
    counts = np.bincount(sub_targets, minlength=C).astype(np.float32)
    class_w = (counts.sum() / (C * np.clip(counts, 1, None))).astype(np.float32)
    return torch.tensor(class_w, device=device)

# -----------------------
# Eval helper: returns (loss, acc, f1_macro) + y_true/y_pred/y_proba
# -----------------------
@torch.no_grad()
def eval_split(model, dl, criterion):
    model.eval(); y_true = []; y_pred = []; y_prob = []; loss_sum = 0.0; n_tot = 0
    for x,y in dl:
        x,y = x.to(device), y.to(device)
        logits = model(x); loss = criterion(logits, y)
        loss_sum += loss.item()*y.size(0); n_tot += y.size(0)
        y_true.extend(y.cpu().numpy().tolist())
        p = logits.argmax(1)
        y_pred.extend(p.cpu().numpy().tolist())
        y_prob.append(torch.softmax(logits, dim=1).cpu().numpy())
    acc = accuracy_score(y_true, y_pred)
    f1m = f1_score(y_true, y_pred, average='macro', zero_division=0)
    y_prob = np.concatenate(y_prob, axis=0) if len(y_prob) else None
    return (loss_sum/n_tot if n_tot>0 else 0.0), acc, f1m, np.array(y_true), np.array(y_pred), y_prob

# -----------------------
# Train a timm model for the sweep (per-model img size)
# -----------------------
def train_model(name, base, classes, criterion_builder, epochs=EPOCHS_SWEEP, lr=LR, wd=WEIGHT_DECAY):
    # free memory between models
    if device.type == 'cuda':
        torch.cuda.empty_cache()
    gc.collect()

    # create model first to get its suggested input size
    m = timm.create_model(name, pretrained=True, num_classes=len(classes), drop_rate=0.2).to(device)
    img_size = infer_img_size_from_model(m)

    # dataloaders for this img_size
    train_dl, val_dl, test_dl = build_dls_for_imgsize(img_size, base)

    # criterion (closure over device/weights)
    crit = criterion_builder(device)

    # optional tweak for short-run stability
    local_lr = lr
    if 'convnext_tiny' in name:
        local_lr = min(lr, 2e-4)

    opt  = torch.optim.AdamW(m.parameters(), lr=local_lr, weight_decay=wd)
    sch  = torch.optim.lr_scheduler.CosineAnnealingLR(opt, T_max=epochs)
    scaler = torch.cuda.amp.GradScaler(enabled=(device.type=='cuda'))

    best_f1 = -1.0; best_state = None
    epoch_times = []
    t0 = time.time()
```

```
    for ep in range(1, epochs+1):
        m.train()
        ep_start = time.time()
        for x,y in train_dl:
            x,y = x.to(device), y.to(device)
            opt.zero_grad(set_to_none=True)
            with torch.cuda.amp.autocast(enabled=(device.type=='cuda')):
                logits = m(x); loss = crit(logits, y)
            scaler.scale(loss).backward()
            # torch.nn.utils.clip_grad_norm_(m.parameters(), 1.0)  # uncomment if needed
            scaler.step(opt); scaler.update()
        sch.step()
        ep_secs = time.time() - ep_start
        epoch_times.append(ep_secs)

        # validation metrics
        v_loss, v_acc, v_f1, _, _, _ = eval_split(m, val_dl, crit)
        if v_f1 > best_f1:
            best_f1   = v_f1
            best_state = {k: v.detach().cpu() for k,v in m.state_dict().items()}

        print(f"[{name}] epoch {ep:02d}/{epochs} | "
              f"val acc {v_acc:.3f} f1M {v_f1:.3f} loss {v_loss:.3f} | "
              f"time {ep_secs:.1f}s (img_size={img_size})")

    total_secs = time.time() - t0

    # test with best state (in-memory only)
    if best_state is not None:
        m.load_state_dict(best_state, strict=True)

    t_loss, t_acc, t_f1, y_true, y_pred, y_proba = eval_split(m, test_dl, crit)

    params_M = sum(p.numel() for p in m.parameters())/1e6
    return {
        'name': name,
        'val_f1_best': float(best_f1),
        'test_acc': float(t_acc),
        'test_f1': float(t_f1),
        'params_M': float(params_M),
        'total_secs': float(total_secs),
        'avg_epoch_secs': float(np.mean(epoch_times)) if epoch_times else float('nan'),
        'epoch_secs_list': [float(s) for s in epoch_times],
        'y_true': y_true.tolist(),
        'y_pred': y_pred.tolist(),
        # y_proba can be large; include if you need calibration/AUC style metrics
        # 'y_proba': y_proba.tolist() if y_proba is not None else None,
    }, (m, crit, (train_dl, val_dl, test_dl))

# -----------------------
# Random Forest baseline on pooled features (optional)
# -----------------------
def random_forest_baseline(base, classes, backbone='resnet18'):
    from sklearn.ensemble import RandomForestClassifier

    # feature extractor @224
    _, ev_tfms = build_transforms(224)
    feat_train = PathsDataset(base['train_paths'][base['train_subset_idx']], base['train_targets'][base['train_subs
    feat_val   = PathsDataset(base['val_paths'],  base['val_targets'],  transform=ev_tfms)
    feat_test  = PathsDataset(base['test_paths'], base['test_targets'], transform=ev_tfms)

    dl_train = DataLoader(feat_train, batch_size=128, shuffle=False, num_workers=NUM_WORKERS_SAFE, pin_memory=PIN_M
    dl_val   = DataLoader(feat_val,   batch_size=128, shuffle=False, num_workers=NUM_WORKERS_SAFE, pin_memory=PIN_M
    dl_test  = DataLoader(feat_test,  batch_size=128, shuffle=False, num_workers=NUM_WORKERS_SAFE, pin_memory=PIN_M

    extractor = timm.create_model(backbone, pretrained=True, num_classes=0, global_pool='avg').to(device).eval()

    def collect(dl):
        feats, ys = [], []
        with torch.no_grad():
            for x,y in dl:
                x = x.to(device)
                f = extractor(x).detach().cpu().numpy()
                feats.append(f); ys.extend(y.numpy().tolist())
        return np.concatenate(feats, axis=0), np.array(ys, dtype=np.int64)

    t0 = time.time()
    Xtr, ytr = collect(dl_train)
    Xva, yva = collect(dl_val)
    Xte, yte = collect(dl_test)

    rf = RandomForestClassifier(n_estimators=300, max_features='sqrt', n_jobs=-1, random_state=RANDOM_SEED)
```

```
        rf.fit(Xtr, ytr)
        secs = time.time() - t0

        vpred = rf.predict(Xva); tpred = rf.predict(Xte)
        vacc  = accuracy_score(yva, vpred)
        tacc  = accuracy_score(yte, tpred)
        tf1   = f1_score(yte, tpred, average='macro', zero_division=0)

        return {
            'name': f'random_forest({backbone}-feats)',
            'val_f1_best': float('nan'),
            'test_acc': float(tacc),
            'test_f1': float(tf1),
            'params_M': 0.0,
            'total_secs': float(secs),
            'avg_epoch_secs': float('nan'),
            'epoch_secs_list': [],
        }

    # -----------------------
    # Metrics plug-in examples
    # Each fn: (y_true, y_pred, y_proba, classes) -> dict
    # -----------------------
    def metric_confmat(y_true, y_pred, y_proba, classes):
        cm = confusion_matrix(y_true, y_pred, labels=np.arange(len(classes)))
        return {"confusion_matrix": cm.tolist()}

    def metric_report(y_true, y_pred, y_proba, classes):
        rep = classification_report(y_true, y_pred, target_names=classes, zero_division=0, output_dict=True)
        return {"classification_report": rep}  # nested dict is OK to log

    # Put your own custom metrics here (they will be called after each model test)
    metrics_list = [metric_confmat]  # extend with your metrics

    # -----------------------
    # Criterion builder factory
    # Uses your houdini_loss if present; else CrossEntropy with either your weights dict
    # or balanced weights computed from the TRAIN subset (configurable).
    # -----------------------
    def make_criterion_builder(base, classes, class_weights_dict=None):
        aligned_user_w = align_weights_from_dict(classes, class_weights_dict) if class_weights_dict else None
        ce_balanced_w  = build_ce_balanced_weights_from_subset(base['train_targets'], base['train_subset_idx'], len(cla

        if 'houdini_loss' in globals():
            print("Using external houdini_loss.")
            def builder(_device):
                def criterion(logits, y): return houdini_loss(logits, y)
                return criterion
            return builder
        else:
            if not USE_TRAIN_BALANCED_WEIGHTS and aligned_user_w is not None:
                print("Using CrossEntropyLoss with YOUR aligned class weights.")
                def builder(_device):
                    return nn.CrossEntropyLoss(weight=aligned_user_w)
                return builder
            else:
                print("Using CrossEntropyLoss with balanced class weights (computed on TRAIN subset).")
                def builder(_device):
                    return nn.CrossEntropyLoss(weight=ce_balanced_w)
                return builder

    # -----------------------
    # Sweep runner (per label_mode)
    # -----------------------
    def run_sweep(label_mode="plant", class_weights_dict=None, candidates=None, epochs=EPOCHS_SWEEP, add_rf=True):
        base = make_base_and_subset(label_mode=label_mode)
        classes = base['classes']
        print(f"Label mode: {label_mode} | Classes: {len(classes)} "
              f"| Train={len(base['train_paths'])} | Val={len(base['val_paths'])} | Test={len(base['test_paths'])}")
        print(f"Train subset: {len(base['train_subset_idx'])} ({SUBSAMPLE_FRAC*100:.0f}% of TRAIN)")

        if candidates is None:
            candidates = [
                'resnet18',
                'resnet50',
                'efficientnet_b2',
                'convnext_tiny',
                'deit_small_patch16_224',
            ]

        criterion_builder = make_criterion_builder(base, classes, class_weights_dict=class_weights_dict)
```

```
    results = []
    models_cache = {}

    for name in candidates:
        try:
            r, pack = train_model(name, base, classes, criterion_builder, epochs=epochs, lr=LR, wd=WEIGHT_DECAY)
            results.append(r)
            models_cache[name] = pack  # (model, crit, (train_dl,val_dl,test_dl))
            print(f"{name:24s} best_val_f1={r['val_f1_best']:.3f} test_acc={r['test_acc']:.3f} "
                  f"f1M={r['test_f1']:.3f} params={r['params_M']:.1f} "
                  f"time={r['total_secs']:.1f}s (avg/epoch={r['avg_epoch_secs']:.1f}s)")
        except Exception as e:
            print(f"[ERROR] {name}: {e}")

    if add_rf:
        try:
            rf_res = random_forest_baseline(base, classes, backbone='resnet18')
            results.append(rf_res)
            print(f"{rf_res['name']:24s} test_acc={rf_res['test_acc']:.3f} "
                  f"f1M={rf_res['test_f1']:.3f} time={rf_res['total_secs']:.1f}s")
        except Exception as e:
            print(f"[ERROR] random_forest: {e}")

    # Sort by best validation F1; fallback to test_f1
    def sort_key(d):
        return (d.get('val_f1_best', float('-inf')), d.get('test_f1', float('-inf')))
    results = sorted(results, key=sort_key, reverse=True)

    # Run plug-in metrics on the BEST model (optional; you can loop over all if needed)
    if metrics_list and len(models_cache):
        best_name = results[0]['name']
        model, crit, (tr_dl, va_dl, te_dl) = models_cache[best_name]
        # Eval again to get arrays for metrics
        _, _, _, y_true, y_pred, y_proba = eval_split(model, te_dl, crit)
        extra = {}
        for fn in metrics_list:
            try:
                extra |= fn(y_true, y_pred, y_proba, classes)
            except Exception as e:
                warnings.warn(f"[metrics] '{getattr(fn,'__name__',str(fn))}' failed: {e}")
        results[0]['extra_metrics_test'] = extra

        # Optional: print classification report for the best model
        try:
            rep = classification_report(y_true, y_pred, target_names=classes, zero_division=0)
            print("\n[Best model test classification report]\n", rep)
        except Exception:
            pass

    sweep_out = {
        'label_mode': label_mode,
        'classes': classes,
        'subset_sizes': {
            'train_subset': len(base['train_subset_idx']),
            'val': len(base['val_paths']),
            'test': len(base['test_paths'])
        },
        'results': results
    }

    print("\nTop-5:")
    for row in results[:5]:
        print(row)
    return sweep_out

# -----------------------
# HOW TO RUN
# (Provide your dicts: class_weights_plants / class_weights_diseases)
# Each dict must map CLASS NAME (as appears in 'classes') -> weight
# -----------------------
# Example:
sweep_plants   = run_sweep(label_mode="plant",   class_weights_dict=class_weights_plants,   candidates=None, epochs
sweep_diseases = run_sweep(label_mode="disease", class_weights_dict=class_weights_disease, candidates=None, epochs=
```

```
Device: cuda
Root: /content/leaf_disease_data/image data
Label mode: plant | Classes: 11 | Train=27045 | Val=3364 | Test=3417
Train subset: 2703 (10% of TRAIN)
Using external houdini_loss.
```

model.safetensors: 100% 46.8M/46.8M [00:00<00:00, 49.6MB/s]

```
[resnet18] epoch 01/4 | val acc 0.739 f1M 0.581 loss 0.310 | time 26.6s (img_size=224)
[resnet18] epoch 02/4 | val acc 0.920 f1M 0.877 loss 0.150 | time 26.2s (img_size=224)
[resnet18] epoch 03/4 | val acc 0.952 f1M 0.923 loss 0.101 | time 26.1s (img_size=224)
[resnet18] epoch 04/4 | val acc 0.967 f1M 0.948 loss 0.089 | time 26.1s (img_size=224)
resnet18               best_val_f1=0.948 test_acc=0.965 f1M=0.946 params=11.2 time=150.0s (avg/epoch=26.2s)
```

model.safetensors: 100% 102M/102M [00:00<00:00, 175MB/s]

```
[resnet50] epoch 01/4 | val acc 0.714 f1M 0.510 loss 0.311 | time 32.1s (img_size=224)
[resnet50] epoch 02/4 | val acc 0.917 f1M 0.866 loss 0.162 | time 32.2s (img_size=224)
[resnet50] epoch 03/4 | val acc 0.954 f1M 0.926 loss 0.118 | time 32.2s (img_size=224)
[resnet50] epoch 04/4 | val acc 0.966 f1M 0.946 loss 0.104 | time 31.8s (img_size=224)
resnet50               best_val_f1=0.946 test_acc=0.965 f1M=0.948 params=23.5 time=198.3s (avg/epoch=32.1s)
```

model.safetensors: 100% 36.8M/36.8M [00:00<00:00, 170MB/s]

```
[efficientnet_b2] epoch 01/4 | val acc 0.901 f1M 0.800 loss 0.099 | time 57.2s (img_size=256)
[efficientnet_b2] epoch 02/4 | val acc 0.940 f1M 0.862 loss 0.061 | time 36.4s (img_size=256)
[efficientnet_b2] epoch 03/4 | val acc 0.946 f1M 0.868 loss 0.056 | time 35.9s (img_size=256)
[efficientnet_b2] epoch 04/4 | val acc 0.949 f1M 0.872 loss 0.051 | time 36.2s (img_size=256)
efficientnet_b2        best_val_f1=0.872 test_acc=0.949 f1M=0.873 params=7.7 time=215.7s (avg/epoch=41.4s)
```

model.safetensors: 100% 114M/114M [00:00<00:00, 162MB/s]

```
[convnext_tiny] epoch 01/4 | val acc 0.387 f1M 0.051 loss 0.613 | time 54.6s (img_size=224)
[convnext_tiny] epoch 02/4 | val acc 0.387 f1M 0.051 loss 0.613 | time 33.0s (img_size=224)
[convnext_tiny] epoch 03/4 | val acc 0.387 f1M 0.051 loss 0.613 | time 33.1s (img_size=224)
[convnext_tiny] epoch 04/4 | val acc 0.387 f1M 0.051 loss 0.613 | time 32.5s (img_size=224)
convnext_tiny          best_val_f1=0.051 test_acc=0.385 f1M=0.051 params=27.8 time=240.6s (avg/epoch=38.3s)
```

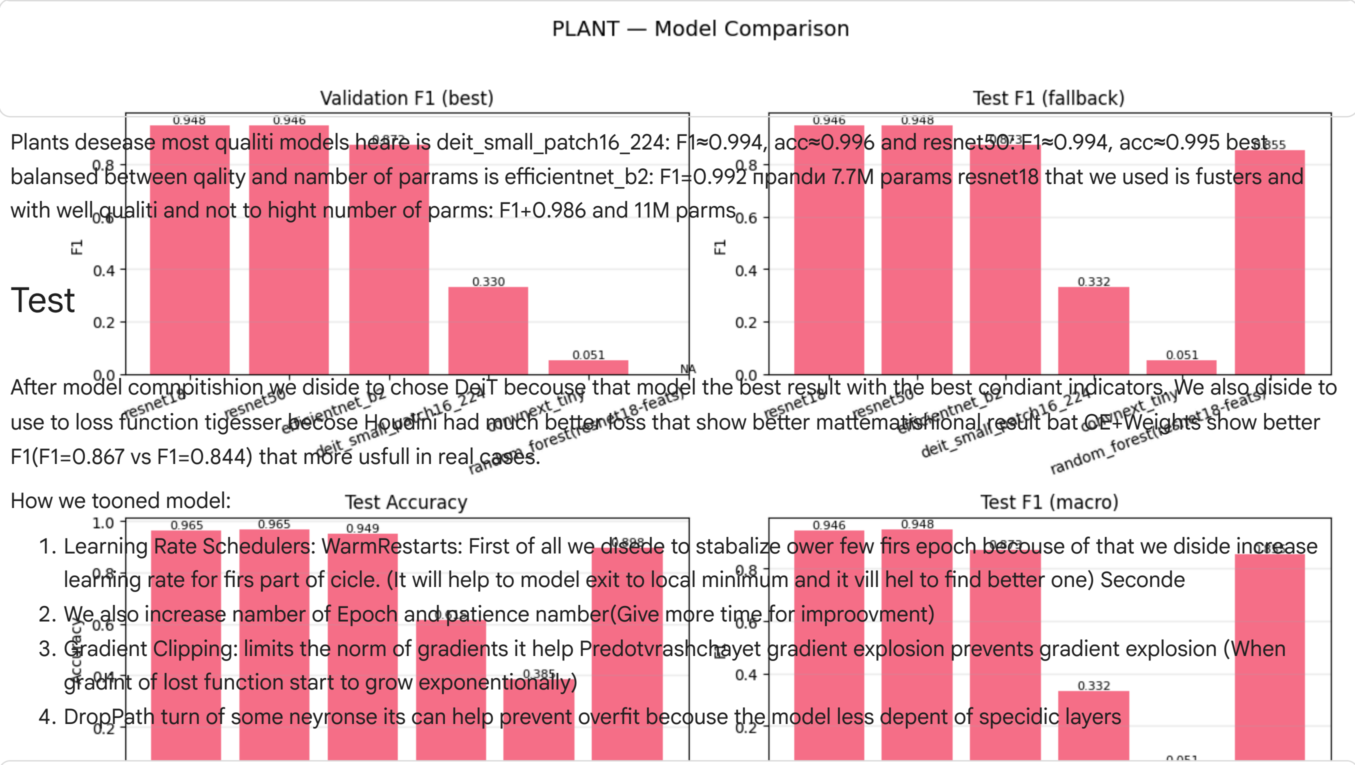
model.safetensors: 100% 88.2M/88.2M [00:00<00:00, 149MB/s]

```
[deit_small_patch16_224] epoch 01/4 | val acc 0.565 f1M 0.281 loss 0.436 | time 29.3s (img_size=224)
[deit_small_patch16_224] epoch 02/4 | val acc 0.594 f1M 0.316 loss 0.406 | time 30.0s (img_size=224)
[deit_small_patch16_224] epoch 03/4 | val acc 0.614 f1M 0.322 loss 0.386 | time 30.4s (img_size=224)
[deit_small_patch16_224] epoch 04/4 | val acc 0.614 f1M 0.330 loss 0.385 | time 30.1s (img_size=224)
deit_small_patch16_224   best_val_f1=0.330 test_acc=0.615 f1M=0.332 params=21.7 time=193.4s (avg/epoch=29.9s)
random_forest(resnet18-feats) test_acc=0.898 f1M=0.855 time=33.4s

[Best model test classification report]
                          precision    recall  f1-score   support

                 Cassava       0.97      1.00      0.98       265
                    Rice       1.00      1.00      1.00       246
                   apple       0.95      0.85      0.90       232
  cherry (including sour)       0.89      0.86      0.88       140
             corn (maize)       0.99      0.99      0.99       280
                   grape       0.98      0.97      0.97       296
                   peach       0.93      0.98      0.95       192
```

```
# ==================== Sweep comparison plots (NO TRAINING) ====================
# Expects:
#   - sweep_plants   = run_sweep(... label_mode="plant" ...)
#   - sweep_diseases = run_sweep(... label_mode="disease" ...)
# Builds 4 bar charts per sweep (all models shown on each chart).

import matplotlib.pyplot as plt
import numpy as np

def _extract_metric_list(sweep_out, key, default_key=None):
    """
    Extracts (labels, values) for a metric from sweep_out['results'].
    If 'key' does not exist in rows and default_key is provided, use it instead.
    """
    rows = sweep_out.get('results', [])
    names = [r.get('name', f'model_{i}') for i, r in enumerate(rows)]
    # If metric not present, try default_key
    if rows and key not in rows[0] and default_key is not None:
        key = default_key
    vals = [float(r.get(key, np.nan)) for r in rows]
    return names, vals, key

def _bar_plot(ax, names, vals, title, ylabel):
    x = np.arange(len(names))
    ax.bar(x, vals)
    ax.set_title(title)
    ax.set_ylabel(ylabel)
    ax.set_xticks(x)
    ax.set_xticklabels(names, rotation=20, ha='right')
    # Annotate bars
    for xi, v in zip(x, vals):
        label = "NA" if not np.isfinite(v) else f"{v:.3f}"
        ax.text(xi, v if np.isfinite(v) else 0.0, label, ha='center', va='bottom', fontsize=8)
    ax.grid(axis='y', alpha=0.25)
```

```
def plot_sweep_four(sweep_out, title_prefix=""):
    """
    Makes 4 bar charts for a given sweep_out:
      1) Validation F1 (best)
      2) Validation Loss  (or Test F1 if val_loss not recorded)
      3) Test Accuracy
      4) Test F1
    Shows all models on each chart.
    """
    # 1) Validation F1 (best across epochs)
    names1, vals1, _ = _extract_metric_list(sweep_out, "val_f1_best")

    # 2) Validation Loss (fallback → test_f1)
    names2, vals2, used2 = _extract_metric_list(sweep_out, "val_loss_best", default_key="test_f1")
    title2 = "Validation Loss" if used2 == "val_loss_best" else "Test F1 (fallback)"

    # 3) Test Accuracy
    names3, vals3, _ = _extract_metric_list(sweep_out, "test_acc")

    # 4) Test F1
    names4, vals4, _ = _extract_metric_list(sweep_out, "test_f1")

    # Ensure consistent model ordering across plots (use order of results)
    fig, axes = plt.subplots(2, 2, figsize=(12, 8))
    plt.suptitle(f"{title_prefix} — Model Comparison", y=1.02, fontsize=14)

    _bar_plot(axes[0,0], names1, vals1, "Validation F1 (best)", "F1")
    _bar_plot(axes[0,1], names2, vals2, title2,            "Loss" if used2 == "val_loss_best" else "F1")
    _bar_plot(axes[1,0], names3, vals3, "Test Accuracy",       "Accuracy")
    _bar_plot(axes[1,1], names4, vals4, "Test F1 (macro)",     "F1")

    plt.tight_layout()
    plt.show()

# --------- Run for both sweeps (each will show 4 charts with all 5 models) ----------
plot_sweep_four(sweep_plants,   title_prefix="PLANT")
plot_sweep_four(sweep_diseases, title_prefix="DISEASE")
```

```
Top-5:
{'name': 'deit_small_patch16_224', 'val_f1_best': 0.7864325252358413, 'test_acc': 0.8797190517998245, 'test_f1': 0.7757950154
{'name': 'resnet50', 'val_f1_best': 0.5414712131022963, 'test_acc': 0.7175885279484928, 'test_f1': 0.5460467532702683, 'param
{'name': 'efficientnet_b2', 'val_f1_best': 0.5087609853443139, 'test_acc': 0.762071992976295, 'test_f1': 0.5132321136582311,
{'name': 'resnet18', 'val_f1_best': 0.4563747576351335, 'test_acc': 0.6695932104184957, 'test_f1': 0.44419792170094535, 'para
{'name': 'convnext_tiny', 'val_f1_best': 0.05204427375429046, 'test_acc': 0.3160667251975417, 'test_f1': 0.05157707462297638,
```



Plants desease most qualiti models heare is deit_small_patch16_224: F1≈0.994, acc≈0.996 and resnet50: F1≈0.994, acc≈0.995 best balansed between qality and namber of parrams is efficientnet_b2: F1=0.992 прandи 7.7M params resnet18 that we used is fusters and with well qualiti and not to hight number of parms: F1+0.986 and 11M parms.

Test

After model compipitishion we diside to chose DeiT becouse that model the best result with the best condiant indicators. We also diside to use to loss function tigesser becose Houdini had much better loss that show better mattematishional result bat CE+Weights show better F1(F1=0.867 vs F1=0.844) that more usfull in real cases.

How we tooned model:

1. Learning Rate Schedulers: WarmRestarts: First of all we disede to stabalize ower few firs epoch becouse of that we diside increase learning rate for firs part of cicle. (It will help to model exit to local minimum and it vill hel to find better one) Seconde
2. We also increase namber of Epoch and patience namber(Give more time for improovment)
3. Gradient Clipping: limits the norm of gradients it help Predotvrashchayet gradient explosion prevents gradient explosion (When gradint of lost function start to grow exponentionally)
4. DropPath turn of some neyronse its can help prevent overfit becouse the model less depent of specidic layers

```
# === DeiT multi-task (plant, disease) with mixed loss (0.3*Houdini + 0.7*CEweights) ===
# - Data tree: ROOT/split/PLANT/DISEASE/*.jpg
# - Two heads: plant_head, disease_head
# - Proper regularization: drop_path (DeiT), label smoothing, dropout in heads, grad clipping
# - Per-image standardization (TF-like), no ImageNet normalize, as in your pipeline

import os, time, math, warnings, numpy as np
from pathlib import Path
from typing import Optional, Dict, List, Tuple

import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import transforms
from PIL import Image

from sklearn.metrics import f1_score, classification_report, accuracy_score
import timm

# ------------------------
# Config
# ------------------------
ROOT            = Path(data_path) / "image data" if (Path(data_path) / "image data").exists() else Path(data_path)
MODEL_NAME      = "deit_small_mt_houdini0.3_cew0.7"
EPOCHS          = 20
PATIENCE        = 4
BATCH_SIZE      = 64
NUM_WORKERS     = 2
LR              = 3e-4
WEIGHT_DECAY    = 1e-4
LABEL_SMOOTHING = 0.1
SUBSAMPLE_FRAC  = 0.2          # set to 0.20 for 20% of train
RANDOM_SEED     = 42
TEST_EVERY_EPOCH = True
DROPPATH_RATE   = 0.1          # stochastic depth inside DeiT
HEAD_DROPOUT    = 0.2          # dropout before linear heads
MAX_GRAD_NORM   = 1.0          # gradient clipping
IMG_SIZE        = 224          # DeiT default

torch.manual_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
pin_mem = (device.type == 'cuda')
print("Device:", device)
print("Root:", str(ROOT))

# ------------------------
# Optional: your weights dicts (class_name -> weight)
# Fill these from your code, or leave None to auto-balance on subset:
# ------------------------
class_weights_plants   = None  # e.g., {"tomato": 0.8, "rice": 1.2, ...}
class_weights_diseases = None  # e.g., {"late blight": 1.3, ...}
```

```
# -----------------------
# TF-like per-image standardization (on 0..255 scale)
# -----------------------
def per_image_standardization_torch(x: torch.Tensor) -> torch.Tensor:
    """Standardize a single image tensor channel-wise after scaling to [0,255]."""
    x255 = x * 255.0
    mean = x255.mean()
    var  = x255.var(unbiased=False)
    std  = torch.sqrt(var + 1e-12)
    adjusted_std = torch.maximum(std, torch.tensor(1.0 / math.sqrt(x255.numel()), device=x.device, dtype=x.dtype))
    return (x255 - mean) / adjusted_std

class PerImageStandardize(object):
    def __call__(self, x: torch.Tensor) -> torch.Tensor:
        return per_image_standardization_torch(x)

train_tfms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE), interpolation=transforms.InterpolationMode.BILINEAR),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(0.2,0.2,0.2,0.03),
    transforms.ToTensor(),
    PerImageStandardize(),
])
eval_tfms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE), interpolation=transforms.InterpolationMode.BILINEAR),
    transforms.ToTensor(),
    PerImageStandardize(),
])

# -----------------------
# Build paths + multi-task labels from file tree
# -----------------------
EXTS = {'.jpg','.jpeg','.png','.bmp','.webp','.tif','.tiff'}

def collect_split_mt(split: str) -> Tuple[List[str], List[str], List[str]]:
    """
    Return (paths, plant_labels_str, disease_labels_str) for given split.
    Expect: ROOT/split/PLANT/DISEASE/*.*
    """
    split_dir = ROOT / split
    paths, plants, diseases = [], [], []
    if not split_dir.exists():
        return paths, plants, diseases
    for plant_dir in sorted(p for p in split_dir.iterdir() if p.is_dir()):
        plant = plant_dir.name
        for disease_dir in sorted(p for p in plant_dir.iterdir() if p.is_dir()):
            disease = disease_dir.name
            for f in disease_dir.rglob("*"):
                if f.is_file() and f.suffix.lower() in EXTS:
                    paths.append(str(f)); plants.append(plant); diseases.append(disease)
    return paths, plants, diseases

def build_label_spaces() -> dict:
    tr_p, tr_pl, tr_di = collect_split_mt("train")
    va_p, va_pl, va_di = collect_split_mt("validation")
    te_p, te_pl, te_di = collect_split_mt("test")

    plant_classes   = sorted(set(tr_pl + va_pl + te_pl))
    disease_classes = sorted(set(tr_di + va_di + te_di))
    p2i = {c:i for i,c in enumerate(plant_classes)}
    d2i = {c:i for i,c in enumerate(disease_classes)}

    tr_y_p = np.array([p2i[s] for s in tr_pl], dtype=np.int64)
    tr_y_d = np.array([d2i[s] for s in tr_di], dtype=np.int64)
    va_y_p = np.array([p2i[s] for s in va_pl], dtype=np.int64)
    va_y_d = np.array([d2i[s] for s in va_di], dtype=np.int64)
    te_y_p = np.array([p2i[s] for s in te_pl], dtype=np.int64)
    te_y_d = np.array([d2i[s] for s in te_di], dtype=np.int64)

    return dict(
        train_paths=np.array(tr_p),  train_plant=tr_y_p,  train_disease=tr_y_d,
        val_paths=np.array(va_p),    val_plant=va_y_p,    val_disease=va_y_d,
        test_paths=np.array(te_p),   test_plant=te_y_p,   test_disease=te_y_d,
        plant_classes=plant_classes, disease_classes=disease_classes
    )

class MTPathsDataset(Dataset):
    """Returns (image_tensor, plant_idx, disease_idx)."""
    def __init__(self, paths, y_plant, y_disease, transform):
        self.paths = paths; self.y_p = y_plant; self.y_d = y_disease
```

```
        self.transform = transform
    def __len__(self): return len(self.paths)
    def __getitem__(self, i):
        img = Image.open(self.paths[i]).convert("RGB")
        if self.transform is not None: img = self.transform(img)
        return img, int(self.y_p[i]), int(self.y_d[i])

base = build_label_spaces()
print(f"Plants: {len(base['plant_classes'])} | Diseases: {len(base['disease_classes'])}")
print(f"Train={len(base['train_paths'])} | Val={len(base['val_paths'])} | Test={len(base['test_paths'])}")

# -----------------------
# Subsample 20% of TRAIN if requested
# -----------------------
if SUBSAMPLE_FRAC < 1.0:
    # stratified by PLANT (could also stratify jointly; plant is fine here)
    by_c = {}
    for i,t in enumerate(base['train_plant']): by_c.setdefault(int(t), []).append(i)
    rng = np.random.default_rng(RANDOM_SEED)
    idx_sub = []
    for idxs in by_c.values():
        k = max(1, int(round(len(idxs)*SUBSAMPLE_FRAC)))
        idx_sub.extend(rng.choice(idxs, size=k, replace=False).tolist())
    rng.shuffle(idx_sub)
else:
    idx_sub = np.arange(len(base['train_paths'])).tolist()

train_ds_full = MTPathsDataset(base['train_paths'], base['train_plant'], base['train_disease'], transform=train_tfms
val_ds        = MTPathsDataset(base['val_paths'],   base['val_plant'],   base['val_disease'],   transform=eval_tfms)
test_ds       = MTPathsDataset(base['test_paths'],  base['test_plant'],  base['test_disease'],  transform=eval_tfms)

train_ds = Subset(train_ds_full, idx_sub)
train_dl = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True,  num_workers=NUM_WORKERS, pin_memory=pin_mem)
val_dl   = DataLoader(val_ds,   batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS, pin_memory=pin_mem)
test_dl  = DataLoader(test_ds,  batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS, pin_memory=pin_mem)

print(f"Train subset: {len(train_ds)} (of {len(train_ds_full)})")

# -----------------------
# Align weights helpers
# -----------------------
def align_weights_from_dict(classes: List[str], weights_dict: Optional[Dict[str, float]]):
    """Map user {class_name: weight} -> tensor aligned to classes order."""
    if not weights_dict:
        return None
    miss = [c for c in classes if c not in weights_dict]
    extra = [k for k in weights_dict if k not in classes]
    if miss:  warnings.warn(f"[class weights] missing: {miss}; using 1.0")
    if extra: warnings.warn(f"[class weights] extra keys ignored: {extra}")
    arr = np.array([weights_dict.get(c, 1.0) for c in classes], dtype=np.float32)
    return torch.tensor(arr, device=device)

def balanced_weights_from_subset(targets: np.ndarray, num_classes: int) -> torch.Tensor:
    """Classic balanced weights computed from subset labels."""
    counts = np.bincount(targets, minlength=num_classes).astype(np.float32)
    w = (counts.sum() / (num_classes * np.clip(counts, 1, None))).astype(np.float32)
    return torch.tensor(w, device=device)

# Build per-head CE weights
sub_idx_np = np.array(idx_sub, dtype=np.int64)
train_pl_subset = base['train_plant'][sub_idx_np]
train_di_subset = base['train_disease'][sub_idx_np]

w_pl_user = align_weights_from_dict(base['plant_classes'],   class_weights_plants)
w_di_user = align_weights_from_dict(base['disease_classes'], class_weights_diseases)

w_pl_bal  = balanced_weights_from_subset(train_pl_subset, len(base['plant_classes']))
w_di_bal  = balanced_weights_from_subset(train_di_subset, len(base['disease_classes']))

w_pl = w_pl_user if w_pl_user is not None else w_pl_bal
w_di = w_di_user if w_di_user is not None else w_di_bal

# -----------------------
# Houdini loss (fallback if not provided globally)
# -----------------------
class HoudiniMarginLoss(nn.Module):
    """Safe Houdini-like margin loss."""
    def __init__(self, tau: float = 1.0):
        super().__init__()
        self.tau = tau
        self.sigmoid = nn.Sigmoid()
    def forward(self, logits, y):
```

```
        s_true = logits.gather(1, y.view(-1,1))
        mask = torch.ones_like(logits, dtype=torch.bool)
        mask.scatter_(1, y.view(-1,1), False)
        s_oth = logits.masked_fill(~mask, float('-inf')).amax(dim=1, keepdim=True)
        margin = (s_oth - s_true) / self.tau
        return self.sigmoid(margin).mean()

USE_HOUDINI = ('houdini_loss' in globals())
if not USE_HOUDINI:
    houdini_loss = HoudiniMarginLoss(tau=1.0)  # local fallback
print("Using", "external houdini_loss." if USE_HOUDINI else "local HoudiniMarginLoss(tau=1.0).")

# -----------------------
# Model: DeiT backbone + two task heads
# -----------------------
# num_classes=0 returns features; set drop_path_rate for stochastic depth regularization
backbone = timm.create_model('deit_small_patch16_224', pretrained=True, num_classes=0, drop_path_rate=DROPPATH_RATE)
backbone.to(device).train()

# infer feature dimension
with torch.no_grad():
    dummy = torch.zeros(1,3,IMG_SIZE,IMG_SIZE).to(device)
    feat = backbone(dummy)  # shape [B, D]
feat_dim = feat.shape[-1]

plant_head = nn.Sequential(
    nn.Dropout(HEAD_DROPOUT),
    nn.Linear(feat_dim, len(base['plant_classes']))
).to(device)

disease_head = nn.Sequential(
    nn.Dropout(HEAD_DROPOUT),
    nn.Linear(feat_dim, len(base['disease_classes']))
).to(device)

params_M = (sum(p.numel() for p in backbone.parameters())
            + sum(p.numel() for p in plant_head.parameters())
            + sum(p.numel() for p in disease_head.parameters()))/1e6

# -----------------------
# Criterion: mixed per head  (0.3 * Houdini + 0.7 * CE(weights, label_smoothing))
# -----------------------
ce_pl = nn.CrossEntropyLoss(weight=w_pl, label_smoothing=LABEL_SMOOTHING)
ce_di = nn.CrossEntropyLoss(weight=w_di, label_smoothing=LABEL_SMOOTHING)

def mixed_loss(logits_pl, y_pl, logits_di, y_di):
    # Houdini terms
    h_pl = houdini_loss(logits_pl, y_pl)
    h_di = houdini_loss(logits_di, y_di)
    # CE terms
    ce_p = ce_pl(logits_pl, y_pl)
    ce_d = ce_di(logits_di, y_di)
    # mix per head, then average across heads
    loss_pl = 0.3 * h_pl + 0.7 * ce_p
    loss_di = 0.3 * h_di + 0.7 * ce_d
    return 0.5 * (loss_pl + loss_di), (h_pl.item(), ce_p.item(), h_di.item(), ce_d.item())

# -----------------------
# Optimizer / Scheduler / AMP
# -----------------------
all_params = list(backbone.parameters()) + list(plant_head.parameters()) + list(disease_head.parameters())
optimizer = torch.optim.AdamW(all_params, lr=LR, weight_decay=WEIGHT_DECAY)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts( optimizer, T_0=3, T_mult=2, eta_min=1e-6)
 # T_0  number of epoch for first cicle/ T_mult=x its mean xT_i/ eta_min min Learning Rate
scaler    = torch.cuda.amp.GradScaler(enabled=(device.type=='cuda'))

# -----------------------
# Eval helpers (per-head metrics + combined macro-F1)
# -----------------------
@torch.no_grad()
def eval_split(dl, name="split"):
    backbone.eval(); plant_head.eval(); disease_head.eval()
    tot = 0; loss_sum = 0.0
    ytp=[]; ypp=[]; ytd=[]; ypd=[]
    for x, yp, yd in dl:
        x, yp, yd = x.to(device), yp.to(device), yd.to(device)
        with torch.cuda.amp.autocast(enabled=(device.type=='cuda')):
            feats = backbone(x)
            lp = plant_head(feats)
            ld = disease_head(feats)
            loss, _ = mixed_loss(lp, yp, ld, yd)
        loss_sum += loss.item() * x.size(0)
        tot += x.size(0)
```

```
            ytp.append(yp.cpu().numpy()); ypp.append(lp.argmax(1).cpu().numpy())
            ytd.append(yd.cpu().numpy()); ypd.append(ld.argmax(1).cpu().numpy())
        if tot == 0:
            return dict(loss=0.0, acc_p=0.0, f1m_p=0.0, acc_d=0.0, f1m_d=0.0, f1m_avg=0.0)
        ytp = np.concatenate(ytp); ypp = np.concatenate(ypp)
        ytd = np.concatenate(ytd); ypd = np.concatenate(ypd)
        acc_p = accuracy_score(ytp, ypp); f1m_p = f1_score(ytp, ypp, average='macro', zero_division=0)
        acc_d = accuracy_score(ytd, ypd); f1m_d = f1_score(ytd, ypd, average='macro', zero_division=0)
        return dict(loss=loss_sum/tot, acc_p=acc_p, f1m_p=f1m_p, acc_d=acc_d, f1m_d=f1m_d, f1m_avg=(f1m_p+f1m_d)/2)

    # -----------------------
    # Train one epoch
    # -----------------------
    def train_one_epoch():
        backbone.train(); plant_head.train(); disease_head.train()
        tot = 0; loss_sum = 0.0
        hpl_sum=0.0; cepl_sum=0.0; hdi_sum=0.0; cedi_sum=0.0
        corr_p=0; corr_d=0
        for x, yp, yd in train_dl:
            x, yp, yd = x.to(device), yp.to(device), yd.to(device)
            optimizer.zero_grad(set_to_none=True)
            with torch.cuda.amp.autocast(enabled=(device.type=='cuda')):
                feats = backbone(x)
                lp = plant_head(feats)
                ld = disease_head(feats)
                loss, (hpl, cepl, hdi, cedi) = mixed_loss(lp, yp, ld, yd)
            scaler.scale(loss).backward()
            if MAX_GRAD_NORM is not None:
                scaler.unscale_(optimizer)
                torch.nn.utils.clip_grad_norm_(all_params, MAX_GRAD_NORM)
            scaler.step(optimizer); scaler.update()

            with torch.no_grad():
                bs = x.size(0); tot += bs
                loss_sum += loss.item()*bs
                hpl_sum += hpl*bs; cepl_sum += cepl*bs
                hdi_sum += hdi*bs; cedi_sum += cedi*bs
                corr_p += (lp.argmax(1)==yp).sum().item()
                corr_d += (ld.argmax(1)==yd).sum().item()

        return dict(
            loss=loss_sum/max(1,tot),
            houdini_p=hpl_sum/max(1,tot),
            ce_p=cepl_sum/max(1,tot),
            houdini_d=hdi_sum/max(1,tot),
            ce_d=cedi_sum/max(1,tot),
            acc_p=corr_p/max(1,tot),
            acc_d=corr_d/max(1,tot),
        )

    # -----------------------
    # Train loop with early stopping on avg(val macro-F1)
    # -----------------------
    best_score, wait = -1.0, 0
    best_state = None
    history = []

    train_wall_start = time.time()
    for epoch in range(1, EPOCHS+1):
        t0 = time.time()
        # train
        t_train0 = time.time()
        tr = train_one_epoch()
        train_secs = time.time() - t_train0

        # val
        va = eval_split(val_dl, name="val")

        # optional test
        if TEST_EVERY_EPOCH and test_dl is not None:
            te = eval_split(test_dl, name="test")
        else:
            te = dict(loss=0.0, acc_p=0.0, f1m_p=0.0, acc_d=0.0, f1m_d=0.0, f1m_avg=0.0)

        scheduler.step()
        epoch_secs = time.time() - t0
        denom = len(train_ds)
        train_ips = denom / max(1e-9, train_secs)

        history.append({
            'epoch': epoch,
            'train': tr,
            'val': va,
```

```
            'test': te,
            'train_secs': float(train_secs),
            'epoch_secs': float(epoch_secs),
            'train_ips':  float(train_ips),
        })

        print(f"epoch {epoch:02d} | "
              f"train loss {tr['loss']:.3f} accP {tr['acc_p']:.3f} accD {tr['acc_d']:.3f} "
              f"({train_secs:.1f}s, {train_ips:.1f} img/s) | "
              f"val f1P {va['f1m_p']:.3f} f1D {va['f1m_d']:.3f} avg {va['f1m_avg']:.3f} loss {va['loss']:.3f} | "
              f"test f1P {te['f1m_p']:.3f} f1D {te['f1m_d']:.3f} avg {te['f1m_avg']:.3f} | "
              f"epoch {epoch_secs:.1f}s")

        # early stopping on average macro-F1 across the two heads
        score = va['f1m_avg']
        if score > best_score:
            best_score, wait = float(score), 0
            best_state = {
                'backbone': {k: v.detach().cpu().clone() for k,v in backbone.state_dict().items()},
                'plant_head': {k: v.detach().cpu().clone() for k,v in plant_head.state_dict().items()},
                'disease_head': {k: v.detach().cpu().clone() for k,v in disease_head.state_dict().items()},
            }
        else:
            wait += 1
            if wait >= PATIENCE:
                print("Early stopping.")
                break

    total_train_secs = time.time() - train_wall_start

    # -----------------------
    # Final eval with best weights + reports
    # -----------------------
    if best_state is not None:
        backbone.load_state_dict(best_state['backbone'], strict=True)
        plant_head.load_state_dict(best_state['plant_head'], strict=True)
        disease_head.load_state_dict(best_state['disease_head'], strict=True)
        backbone.to(device).eval(); plant_head.to(device).eval(); disease_head.to(device).eval()

        va = eval_split(val_dl,  name="val(final)")
        te = eval_split(test_dl, name="test(final)") if test_dl is not None else dict(loss=0, acc_p=0, f1m_p=0, acc_d=0,

        print("Best weights restored from memory. Best val avg F1-macro =", round(best_score, 4))
        print("Final VAL:  accP={:.4f} f1P={:.4f} accD={:.4f} f1D={:.4f} avgF1={:.4f} loss={:.4f}".format(
            va['acc_p'], va['f1m_p'], va['acc_d'], va['f1m_d'], va['f1m_avg'], va['loss']))
        print("Final TEST: accP={:.4f} f1P={:.4f} accD={:.4f} f1D={:.4f} avgF1={:.4f} loss={:.4f}".format(
            te['acc_p'], te['f1m_p'], te['acc_d'], te['f1m_d'], te['f1m_avg'], te['loss']))

        # Detailed per-head classification reports on TEST
        @torch.no_grad()
        def test_reports():
            if test_dl is None:
                print("TEST split not found; skipping reports."); return
            ytp=[]; ypp=[]; ytd=[]; ypd=[]
            for x, yp, yd in test_dl:
                x = x.to(device)
                feats = backbone(x)
                lp = plant_head(feats); ld = disease_head(feats)
                ypp.append(lp.argmax(1).cpu().numpy()); ytp.append(yp.numpy())
                ypd.append(ld.argmax(1).cpu().numpy()); ytd.append(yd.numpy())
            ytp=np.concatenate(ytp); ypp=np.concatenate(ypp)
            ytd=np.concatenate(ytd); ypd=np.concatenate(ypd)
            print("\n[TEST report — PLANT]")
            print(classification_report(ytp, ypp, target_names=base['plant_classes'], digits=3, zero_division=0))
            print("\n[TEST report — DISEASE]")
            print(classification_report(ytd, ypd, target_names=base['disease_classes'], digits=3, zero_division=0))
        test_reports()

        output_summary = {
            'name': MODEL_NAME,
            'params_M': float(params_M),
            'best_val_avg_f1_macro': float(best_score),
            'final_val': va,
            'final_test': te,
            'total_train_secs': float(total_train_secs),
            'avg_epoch_secs': float(np.mean([h['epoch_secs'] for h in history])) if history else float('nan'),
        }
        output = {'history': history, 'summary': output_summary}
        print(output_summary)
    else:
        print("Warning: No best_state stored in memory. Skipping final evaluation.")
        output_summary = {
            'name': MODEL_NAME
```

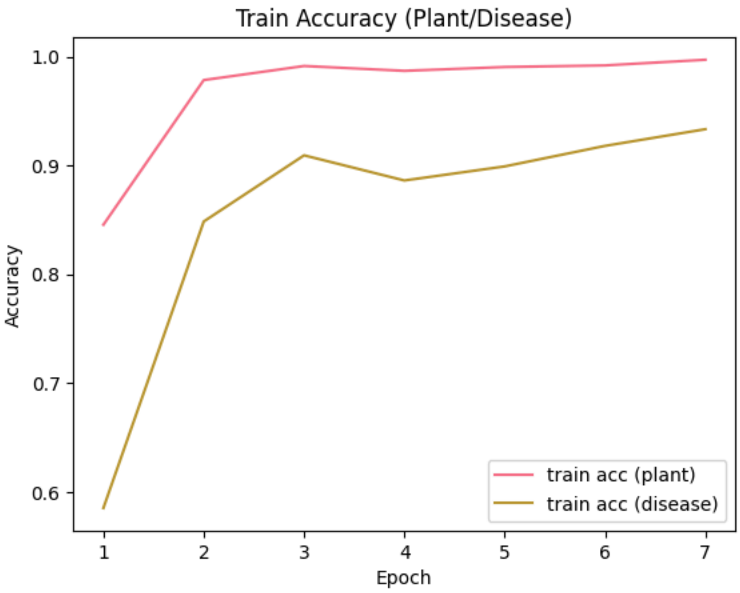
```
        'name': MODEL_NAME,
        'params_M': float(params_M),
        'best_val_avg_f1_macro': float('nan'),
        'final_val': None,
        'final_test': None,
        'total_train_secs': float(total_train_secs),
        'avg_epoch_secs': float(np.mean([h['epoch_secs'] for h in history])) if history else float('nan'),
    }
    output = {'history': history, 'summary': output_summary}
    print(output_summary)
```

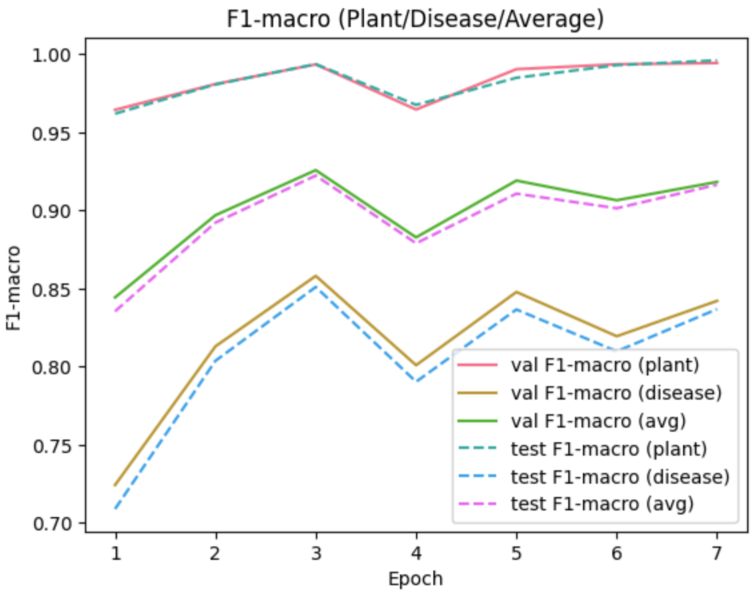
```
[TEST report — PLANT]
                         precision    recall  f1-score   support

              Cassava      0.996     1.000     0.998       265
                 Rice      1.000     1.000     1.000       246
                apple      1.000     1.000     1.000       232
cherry (including sour)      0.986     0.993     0.989       140
          corn (maize)      1.000     1.000     1.000       280
                grape      0.997     1.000     0.998       296
                peach      0.985     1.000     0.992       192
          pepper, bell      0.989     0.978     0.983       180
               potato      0.993     0.968     0.980       155
           strawberry      1.000     0.982     0.991       114
               tomato      0.996     0.998     0.997      1317

             accuracy                          0.996      3417
            macro avg      0.995     0.993     0.994      3417
         weighted avg      0.996     0.996     0.996      3417


[TEST report — DISEASE]
                                    precision    recall  f1-score   support

             Bacterial Blight (CBB)      0.484     0.625     0.545        48
        Brown Streak Disease (CBSD)      0.565     0.842     0.676        57
                          BrownSpot      0.420     0.872     0.567        39
                 Green Mottle (CGM)      0.464     0.667     0.547        48
                            Healthy      0.702     0.415     0.521       176
                              Hispa      0.462     0.286     0.353        42
                          LeafBlast      0.725     0.509     0.598        57
                Mosaic Disease (CMD)      0.733     0.750     0.742        44
                         apple scab      0.979     1.000     0.989        46
                     bacterial spot      0.972     0.980     0.976       393
                          black rot      0.970     0.985     0.977       131
                   cedar apple rust      1.000     1.000     1.000        21
   cercospora leaf spot gray leaf spot      0.854     0.946     0.897        37
                        common rust      1.000     1.000     1.000        87
                       early blight      0.977     0.896     0.935       144
                esca (black measles)      0.980     0.980     0.980       101
                            healthy      0.988     0.992     0.990       591
                        late blight      0.981     0.962     0.971       210
    leaf blight (isariopsis leaf spot)      1.000     1.000     1.000        79
                          leaf mold      1.000     0.986     0.993        70
                        leaf scorch      1.000     0.988     0.994        80
                northern leaf blight      0.970     0.915     0.942        71
                     powdery mildew      1.000     0.987     0.993        77
                 septoria leaf spot      0.916     0.930     0.923       129
spider mites two-spotted spider mite      0.982     0.885     0.931       122
                        target spot      0.838     0.961     0.895       102
                tomato mosaic virus      0.800     1.000     0.889        28
        tomato yellow leaf curl virus      0.995     0.990     0.992       387

                           accuracy                          0.912      3417
                          macro avg      0.848     0.870     0.851      3417
                       weighted avg      0.919     0.912     0.911      3417

{'name': 'deit_small_mt_houdini0.3_cew0.7', 'params_M': 21.680679, 'best_val_avg_f1_macro': 0.9257667441815131, 'f:
```

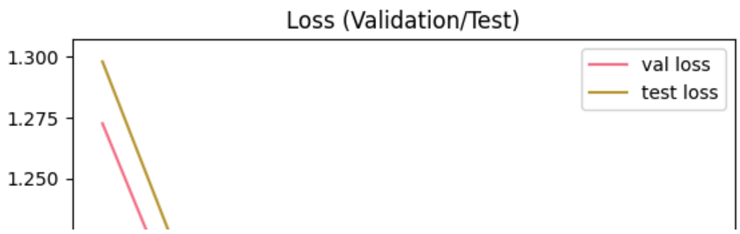
```
# ========================
# Plots & Clusters for DeiT Multi-Task Training
# - Works with the history/output of the DeiT multi-task script
# - Draws curves for plant/disease heads and averaged F1
# - Builds 2D clusters of features (PCA->t-SNE), colored by plant/disease
# ========================
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

# -------- utils to read history safely --------
def _get(hist, key_path, default=np.nan):
    """Safely dig nested metrics: key_path = ('val','f1m_p'), etc."""
    cur = hist
    for k in key_path:
        if cur is None: return default
```

```
        cur = cur.get(k, None)
    return cur if cur is not None else default

def is_multitask_history(sample_epoch_dict):
    """Detect multi-task history by presence of plant/disease metrics."""
    if not sample_epoch_dict: return False
    vt = sample_epoch_dict.get('val', {})
    # multi-task history contains f1m_p / f1m_d (or acc_p / acc_d in train)
    return ('f1m_p' in vt and 'f1m_d' in vt) or ('acc_p' in sample_epoch_dict.get('train', {}))

# -------- history adapter: supports both single- and multi-task --------
hist = output['history'] if isinstance(output, dict) and 'history' in output else output
assert isinstance(hist, list) and len(hist) > 0, "History is empty or invalid."

epochs = [d['epoch'] for d in hist]
mt = is_multitask_history(hist[0])

# ========================
# CURVES
# ========================

if mt:
    # -------- Accuracy (train) per head --------
    plt.figure()
    plt.plot(epochs, [d['train'].get('acc_p', np.nan) for d in hist], label="train acc (plant)")
    plt.plot(epochs, [d['train'].get('acc_d', np.nan) for d in hist], label="train acc (disease)")
    plt.xlabel("Epoch"); plt.ylabel("Accuracy"); plt.legend(); plt.title("Train Accuracy (Plant/Disease)")
    plt.show()

    # -------- Validation/Test F1 per head + average --------
    plt.figure()
    plt.plot(epochs, [d['val'].get('f1m_p', np.nan) for d in hist],   label="val F1-macro (plant)")
    plt.plot(epochs, [d['val'].get('f1m_d', np.nan) for d in hist],   label="val F1-macro (disease)")
    plt.plot(epochs, [d['val'].get('f1m_avg', np.nan) for d in hist], label="val F1-macro (avg)")
    plt.plot(epochs, [d['test'].get('f1m_p', np.nan) for d in hist],  '--', label="test F1-macro (plant)")
    plt.plot(epochs, [d['test'].get('f1m_d', np.nan) for d in hist],  '--', label="test F1-macro (disease)")
    plt.plot(epochs, [d['test'].get('f1m_avg', np.nan) for d in hist],'--', label="test F1-macro (avg)")
    plt.xlabel("Epoch"); plt.ylabel("F1-macro"); plt.legend(); plt.title("F1-macro (Plant/Disease/Average)")
    plt.show()

    # -------- Loss (val/test) --------
    plt.figure()
    plt.plot(epochs, [d['val'].get('loss', np.nan) for d in hist],   label="val loss")
    plt.plot(epochs, [d['test'].get('loss', np.nan) for d in hist],  label="test loss")
    plt.xlabel("Epoch"); plt.ylabel("Loss"); plt.legend(); plt.title("Loss (Validation/Test)")
    plt.show()

    # -------- Throughput & epoch time --------
    plt.figure()
    plt.plot(epochs, [d.get('train_ips', np.nan) for d in hist])
    plt.xlabel("Epoch"); plt.ylabel("Images/sec (train)"); plt.title("Training Throughput")
    plt.show()

    plt.figure()
    plt.plot(epochs, [d.get('epoch_secs', np.nan) for d in hist])
    plt.xlabel("Epoch"); plt.ylabel("Seconds"); plt.title("Epoch Wall-Clock Time")
    plt.show()

    # -------- Optional: show loss components if you logged them --------
    if 'houdini_p' in hist[0]['train']:
        plt.figure()
        plt.plot(epochs, [d['train'].get('houdini_p', np.nan) for d in hist], label="Houdini plant")
        plt.plot(epochs, [d['train'].get('ce_p', np.nan)      for d in hist], label="CE plant")
        plt.plot(epochs, [d['train'].get('houdini_d', np.nan) for d in hist], label="Houdini disease")
        plt.plot(epochs, [d['train'].get('ce_d', np.nan)      for d in hist], label="CE disease")
        plt.xlabel("Epoch"); plt.ylabel("Loss terms"); plt.legend(); plt.title("Loss Components (Train)")
        plt.show()

else:
    # ====== Backwards-compatible (old single-task history) ======
    plt.figure()
    plt.plot(epochs, [d['train']['acc'] for d in hist], label="train acc")
    plt.plot(epochs, [d['val']['acc']   for d in hist], label="val acc")
    plt.plot(epochs, [d['test']['acc']  for d in hist], label="test acc")
    plt.xlabel("Epoch"); plt.ylabel("Accuracy"); plt.legend(); plt.title("Accuracy")
    plt.show()

    plt.figure()
    plt.plot(epochs, [d['train']['loss'] for d in hist], label="train loss")
    plt.plot(epochs, [d['val']['loss']   for d in hist], label="val loss")
    plt.plot(epochs, [d['test']['loss']  for d in hist], label="test loss")
    plt.xlabel("Epoch"); plt.ylabel("Loss"); plt.legend(); plt.title("Loss")
    plt.show()
```

```
    plt.show()

    plt.figure()
    plt.plot(epochs, [d['val']['f1_macro']  for d in hist], label="val f1_macro")
    plt.plot(epochs, [d['test']['f1_macro'] for d in hist], label="test f1_macro")
    plt.xlabel("Epoch"); plt.ylabel("F1-macro"); plt.legend(); plt.title("F1-macro")
    plt.show()

    plt.figure()
    plt.plot(epochs, [d['val']['f1_weighted']  for d in hist], label="val f1_weighted")
    plt.plot(epochs, [d['test']['f1_weighted'] for d in hist], label="test f1_weighted")
    plt.xlabel("Epoch"); plt.ylabel("F1-weighted"); plt.legend(); plt.title("F1-weighted")
    plt.show()

    plt.figure()
    plt.plot(epochs, [d['train_ips'] for d in hist])
    plt.xlabel("Epoch"); plt.ylabel("Images/sec (train)"); plt.title("Training Throughput")
    plt.show()

    plt.figure()
    plt.plot(epochs, [d['epoch_secs'] for d in hist])
    plt.xlabel("Epoch"); plt.ylabel("Seconds"); plt.title("Epoch Wall-Clock Time")
    plt.show()

# =========================
# CLUSTER PLOTS (PCA -> t-SNE) for features
# - Requires: backbone, test_dl, device, and the class lists in `base`
# - Colors by plant classes and by disease classes
# =========================
@torch.no_grad()
def collect_features_and_labels(backbone, dl, max_samples=4000):
    """Collect backbone features and both labels from a dataloader."""
    backbone.eval()
    feats = []; y_pl = []; y_di = []
    n_seen = 0
    for x, yp, yd in dl:
        x = x.to(device)
        f = backbone(x)  # [B, D]
        feats.append(f.detach().cpu().numpy())
        y_pl.append(yp.numpy()); y_di.append(yd.numpy())
        n_seen += x.size(0)
        if n_seen >= max_samples:
            break
    F = np.concatenate(feats, axis=0)
    YP = np.concatenate(y_pl, axis=0)
    YD = np.concatenate(y_di, axis=0)
    return F, YP, YD

def tsne2d_from_feats(F, pca_dim=50, tsne_perplexity=35, tsne_iter=1000, random_state=42):
    """Light PCA compression then t-SNE to 2D."""
    if F.shape[1] > pca_dim:
        Fp = PCA(n_components=pca_dim, random_state=random_state).fit_transform(F)
    else:
        Fp = F
    T = TSNE(n_components=2, perplexity=tsne_perplexity, n_iter=tsne_iter, init='pca',
             learning_rate='auto', random_state=random_state).fit_transform(Fp)
    return T

def scatter_clusters(T2, labels, title, class_names, alpha=0.7, s=10):
    """2D scatter with class colors (matplotlib default cycle)."""
    plt.figure(figsize=(7,6))
    # Draw per class to get a legend
    uniq = np.unique(labels)
    for c in uniq:
        m = labels == c
        plt.scatter(T2[m,0], T2[m,1], s=s, alpha=alpha, label=class_names[c])
    plt.legend(markerscale=2, frameon=True, fontsize=8)
    plt.title(title); plt.xlabel("t-SNE 1"); plt.ylabel("t-SNE 2")
    plt.tight_layout()
    plt.show()

try:
    # 1) collect features from TEST split
    F, YP, YD = collect_features_and_labels(backbone, test_dl, max_samples=4000)
    # 2) tsne
    T2 = tsne2d_from_feats(F, pca_dim=50, tsne_perplexity=35, tsne_iter=1200, random_state=RANDOM_SEED)
    # 3) plot clusters by PLANT
    scatter_clusters(T2, YP, title="Feature clusters — PLANT", class_names=base['plant_classes'])
    # 4) plot clusters by DISEASE
    scatter_clusters(T2, YD, title="Feature clusters — DISEASE", class_names=base['disease_classes'])
except Exception as e:
    print(f"[cluster plots] skipped due to: {e}")
```


Train Accuracy (Plant/Disease)
Accuracy
Epoch
train acc (plant)
train acc (disease)


F1-macro (Plant/Disease/Average)
F1-macro
Epoch
val F1-macro (plant)
val F1-macro (disease)
val F1-macro (avg)
test F1-macro (plant)
test F1-macro (disease)
test F1-macro (avg)


Loss (Validation/Test)
val loss
test loss