```
# IMPORTANT: RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES,
# THEN FEEL FREE TO DELETE THIS CELL.
# NOTE: THIS NOTEBOOK ENVIRONMENT DIFFERS FROM KAGGLE'S PYTHON
# ENVIRONMENT SO THERE MAY BE MISSING LIBRARIES USED BY YOUR
# NOTEBOOK.
import kagglehub
shubham2703_five_crop_diseases_dataset_path = kagglehub.dataset_download('shubham2703/five-crop-diseases-dataset')

print('Data source import complete.')

```

```
!nvidia-smi
```

Installations

```
# !pip install split-folders
```

```
Collecting split-folders
  Downloading split_folders-0.5.1-py3-none-any.whl.metadata (6.2 kB)
Downloading split_folders-0.5.1-py3-none-any.whl (8.4 kB)
Installing collected packages: split-folders
Successfully installed split-folders-0.5.1
```

Imports

```
import random
# import splitfolders
import os
import json
import shutil
from pathlib import Path
from typing import Optional, List, Union, Tuple, Dict

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms, datasets, utils
import torchvision.models as models
from torch.utils.data import DataLoader, Dataset
from torch.utils.tensorboard import SummaryWriter

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm

from sklearn.metrics import confusion_matrix, classification_report, precision_recall_fscore_support
```

Set seed for deterministic results

```
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

if torch.cuda.is_available():
    torch.cuda.manual_seed(SEED)
    torch.cuda.manual_seed_all(SEED)

```

Dataset Preparation

Fetching Dataset

↳ 3 cells hidden

Extract and restructure

↳ 6 cells hidden

Train-Val-Test Split

↳ 3 cells hidden

Data Loading and Augmentation

↳ 2 cells hidden

Defining Model Architecture and Params

↳ 3 cells hidden

Training

↳ 7 cells hidden

Training

↳ 1 cell hidden

Model Evaluation

↳ 2 cells hidden

Mobilenet

↳ 4 cells hidden

Efficient Net

↳ 3 cells hidden

Contrastive Learning with Triplet Loss

```python
import os

def rename_folders(base_dir):
    """
    Renames specific folders within train, val, and test directories
    by adding a 'Sugarcane_' prefix.
    """
    if not os.path.isdir(base_dir):
        raise ValueError(f"Base directory '{base_dir}' does not exist.")

    # Define the splits (subdirectories) to process
    splits = ["train", "val", "test"]

    # Define the folder names to be changed
    folders_to_rename = ["Healthy", "Bacterial Blight", "Red Rot"]

    # Loop through each split (train, val, test)
    for split in splits:
        split_path = os.path.join(base_dir, split)

        # Check if the split directory (e.g., 'dataset/train') exists
        if not os.path.isdir(split_path):
```

```
            continue

        print(f"Processing directory: {split_path}")

        # Loop through each folder name we want to rename
        for folder_name in folders_to_rename:
            old_path = os.path.join(split_path, folder_name)
            new_name = f"Sugarcane__{folder_name}"
            new_path = os.path.join(split_path, new_name)

            # Check if the source folder (e.g., 'dataset/train/Health') exists
            if os.path.isdir(old_path):
                try:
                    os.rename(old_path, new_path)
                    print(f"  - Renamed '{folder_name}' to '{new_name}'")
                except OSError as e:
                    print(f"  - Error renaming '{folder_name}': {e}")

    print("\nRenaming complete.")

rename_folders('dataset')
```

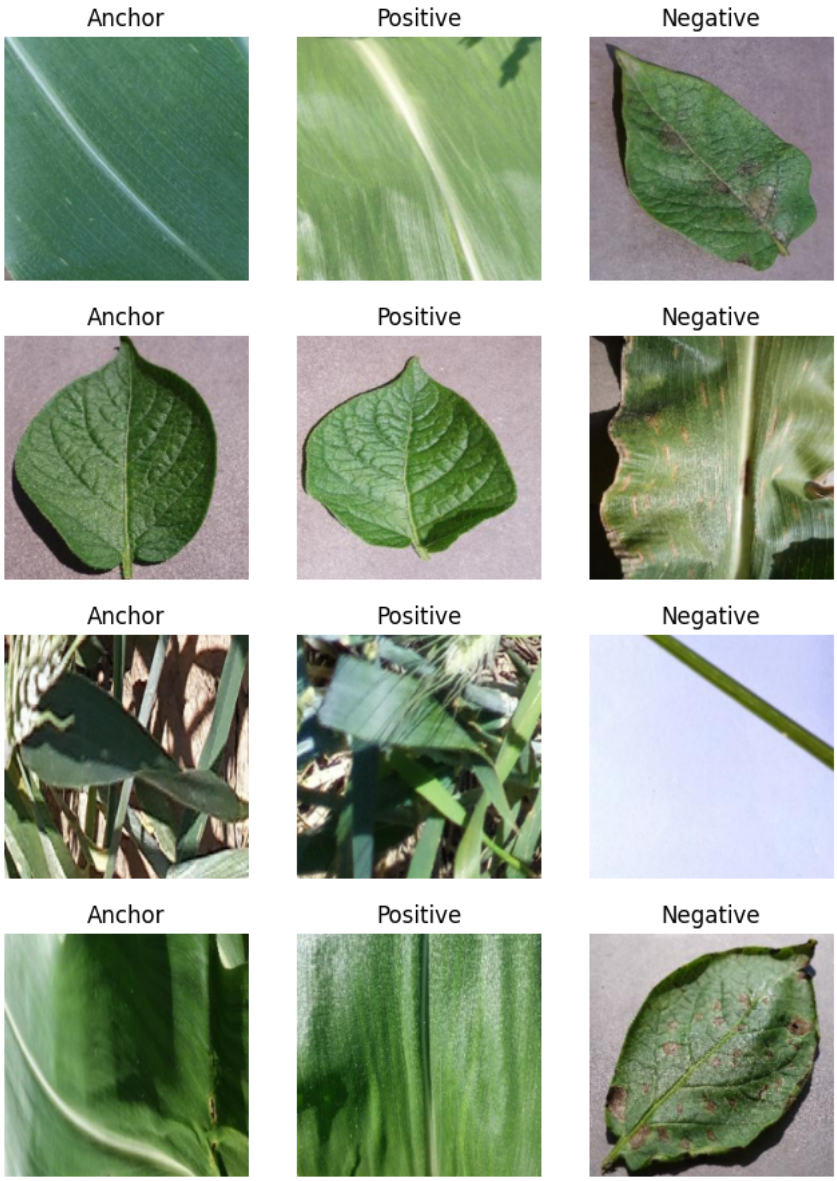
```
Processing directory: dataset/train
Processing directory: dataset/val
Processing directory: dataset/test

Renaming complete.
```

```
class Siamese(nn.Module):
    def __init__(self, backbone="resnet18", pretrained=True, embedding_dim=128, freeze_backbone=True):
        super().__init__()
        if backbone == "resnet18":
            base = models.resnet18(pretrained=pretrained)
            in_features = base.fc.in_features
            base.fc = nn.Identity()  # remove final classifier
        elif backbone == "resnet50":
            base = models.resnet50(pretrained=pretrained)
            in_features = base.fc.in_features
            base.fc = nn.Identity()  # remove final classifier
        else:
            raise NotImplementedError("Only resnet18 implemented here")

        self.backbone = base
        self.embedding = nn.Linear(in_features, embedding_dim)

        if freeze_backbone:
            for param in self.backbone.parameters():
                param.requires_grad = False

    def forward(self, x):
        x = self.backbone(x)        # features
        x = self.embedding(x)       # project to embedding_dim
        x = F.normalize(x, p=2, dim=1)  # L2 normalize
        return x
```

```
class SiameseTripletDataset(Dataset):
    def __init__(self, root, transform=None):
        """
        root: path to dataset root, structured as ImageFolder
        e.g. root/ClassName/*.jpg
        """
        self.dataset = datasets.ImageFolder(root=root, transform=transform)
        self.transform = transform

        # Map class -> indices
        self.class_to_indices = {}
        for idx, (_, label) in enumerate(self.dataset.samples):
            cls = self.dataset.classes[label]
            self.class_to_indices.setdefault(cls, []).append(idx)

        # Separate healthy and disease classes
        self.healthy_classes = [c for c in self.dataset.classes if "Healthy" in c]
        self.disease_classes = [c for c in self.dataset.classes if "Healthy" not in c]

        if not self.healthy_classes:
            raise ValueError("No healthy classes found in dataset!")

    def __len__(self):
        # length = total number of healthy samples
        return sum(len(self.class_to_indices[c]) for c in self.healthy_classes)

```

```
    def __getitem__(self, idx):
        # Pick a healthy class at random
        healthy_class = random.choice(self.healthy_classes)
        anchor_idx = random.choice(self.class_to_indices[healthy_class])
        anchor_img, _ = self.dataset[anchor_idx]

        # Positive (same healthy class)
        pos_idx = random.choice(self.class_to_indices[healthy_class])
        pos_img, _ = self.dataset[pos_idx]

        # Negative (any disease class)
        neg_class = random.choice(self.disease_classes)
        neg_idx = random.choice(self.class_to_indices[neg_class])
        neg_img, _ = self.dataset[neg_idx]

        return anchor_img, pos_img, neg_img

```

```
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Define transforms
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
])

# Initialize dataset
data_dir = "dataset/train"  # path to your dataset
triplet_ds = SiameseTripletDataset(root=data_dir, transform=transform)

# Wrap in DataLoader
loader = DataLoader(triplet_ds, batch_size=1, shuffle=True)

# Visualize a few triplets
for i, (a, p, n) in enumerate(loader):
    fig, axes = plt.subplots(1, 3, figsize=(8, 3))
    axes[0].imshow(a[0].permute(1,2,0))
    axes[0].set_title("Anchor")
    axes[1].imshow(p[0].permute(1,2,0))
    axes[1].set_title("Positive")
    axes[2].imshow(n[0].permute(1,2,0))
    axes[2].set_title("Negative")
    for ax in axes: ax.axis("off")
    plt.show()
    if i == 3: break  # show 3 triplets only

```




```
from torch import amp
```

```
class SiameseTrainer:
    def __init__(self, model, device, lr=1e-3, log_interval=20, use_amp=False, checkpoint_dir="siamese_crops_checkpoints"
        model = model.to(device)
        if torch.cuda.device_count() > 1:
            print(f"Using {torch.cuda.device_count()} GPUs with DataParallel")
            model = nn.DataParallel(model)
        self.model = model # torch.compile(model)
        if isinstance(self.model, nn.DataParallel):
            self.model_to_save = self.model.module
        else:
            self.model_to_save = self.model

        self.device = device
        self.optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, self.model.parameters()), lr=lr)
        self.criterion = nn.TripletMarginLoss(margin=1.0, p=2)
        self.scaler = amp.GradScaler("cuda", enabled=use_amp)
        self.use_amp = use_amp
        self.log_interval = log_interval

        self.checkpoint_dir = checkpoint_dir
        os.makedirs(self.checkpoint_dir, exist_ok=True)
        self.best_loss = float('inf')
        self.count_trainable_parameters()

    def count_trainable_parameters(self):
        """Counts and prints the number of trainable parameters in the model."""
        num_params = sum(p.numel() for p in self.model.parameters() if p.requires_grad)
        print(f"Number of trainable parameters: {num_params:,}")
        return num_params

```

```
    def train(self, train_loader, epochs=5):
        self.model.train()

        epoch_bar = tqdm(range(epochs), desc="Epochs")
        for epoch in epoch_bar:
            total_loss = 0

            batch_bar = tqdm(train_loader, desc=f"Epoch {epoch+1}", leave=False)
            for batch_idx, (a, p, n) in enumerate(batch_bar):
                a, p, n = a.to(self.device), p.to(self.device), n.to(self.device)

                with amp.autocast("cuda", enabled=self.use_amp):
                    emb_a = self.model(a)
                    emb_p = self.model(p)
                    emb_n = self.model(n)
                    loss = self.criterion(emb_a, emb_p, emb_n)

                self.optimizer.zero_grad(set_to_none=True)
                self.scaler.scale(loss).backward()
                self.scaler.step(self.optimizer)
                self.scaler.update()

                total_loss += loss.item()
                batch_bar.set_postfix(loss=f"{loss.item():.4f}")

            avg_loss = total_loss / len(train_loader)
            epoch_bar.set_postfix(avg_loss=f"{avg_loss:.4f}")

            last_checkpoint_path = os.path.join(self.checkpoint_dir, "siamese_crops_last_model.pth")
            self.save_checkpoint(last_checkpoint_path, epoch, avg_loss)

            if avg_loss < self.best_loss:
                self.best_loss = avg_loss
                best_checkpoint_path = os.path.join(self.checkpoint_dir, "siamese_crops_best_model.pth")
                self.save_checkpoint(best_checkpoint_path, epoch, self.best_loss)
                tqdm.write(f"best model updated: {self.best_loss:.4f} at epoch {epoch+1}")

    def save_checkpoint(self, file_path, epoch, loss):
        """Saves a checkpoint dictionary to a file (works for single & multi-GPU)."""
        # unwrap model if DataParallel was used
        model_to_save = self.model.module if isinstance(self.model, torch.nn.DataParallel) else self.model
        checkpoint = {
            'epoch': epoch + 1,
            'model_state_dict': model_to_save.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict(),
            'scaler_state_dict': self.scaler.state_dict() if hasattr(self, "scaler") else None,
            'loss': loss,
        }
        # save checkpoint (portable)
        torch.save(checkpoint, file_path)

        print(f"Checkpoint saved at {file_path}")

    def save_embeddings(self, dataloader, save_path="embeddings.pt"):
        """Extract embeddings for whole dataset"""
        self.model.eval()
        embeddings, labels = [], []
        with torch.no_grad():
            for imgs, lbls in dataloader:
                imgs = imgs.to(self.device)
                emb = self.model(imgs)
                embeddings.append(emb.cpu())
                labels.append(lbls)
        torch.save({"embeddings": torch.cat(embeddings), "labels": torch.cat(labels)}, save_path)
        print(f"Embeddings saved to {save_path}")
```

data_dir: " dataset/train "

BATCH_SIZE: 512

IMAGE_SIZE: 224

EPOCHS: 20

LR: 1e-5

Show code

```
model = Siamese(backbone="resnet18", embedding_dim=128, freeze_backbone=True)
model.to(device)
trainer = SiameseTrainer(model, device, lr=1e-5, use_amp=True)
```

```
Using 2 GPUs with DataParallel
Number of trainable parameters: 65,664
```

```
# trainer.train(triplet_loader, epochs=10)
```

```
Epochs:   0%|          | 0/10 [00:00<?, ?it/s]
Epoch 1:   0%|          | 0/7 [00:00<?, ?it/s]
Epoch 1:   0%|          | 0/7 [03:03<?, ?it/s, loss=0.8442]
Epoch 1:  14%|█▍        | 1/7 [03:03<18:23, 183.87s/it, loss=0.8442]
Epoch 1:  14%|█▍        | 1/7 [03:19<18:23, 183.87s/it, loss=0.8356]
Epoch 1:  29%|██▊       | 2/7 [03:19<07:03, 84.63s/it, loss=0.8356]
Epoch 1:  29%|██▊       | 2/7 [03:25<07:03, 84.63s/it, loss=0.8359]
Epoch 1:  43%|████▎     | 3/7 [03:25<03:15, 48.84s/it, loss=0.8359]
Epoch 1:  43%|████▎     | 3/7 [03:25<03:15, 48.84s/it, loss=0.8369]
Epoch 1:  57%|█████▋    | 4/7 [03:25<01:29, 29.76s/it, loss=0.8369]
Epoch 1:  57%|█████▋    | 4/7 [05:18<01:29, 29.76s/it, loss=0.8301]
Epoch 1:  71%|███████▏  | 5/7 [05:18<01:59, 59.78s/it, loss=0.8301]
Epoch 1:  71%|███████▏  | 5/7 [05:30<01:59, 59.78s/it, loss=0.8183]
Epoch 1:  86%|████████▌ | 6/7 [05:30<00:43, 43.42s/it, loss=0.8183]
Epoch 1:  86%|████████▌ | 6/7 [05:30<00:43, 43.42s/it, loss=0.8274]
Epoch 1: 100%|██████████| 7/7 [05:30<00:00, 29.31s/it, loss=0.8274]
Epochs:  10%|▉         | 1/10 [05:30<49:38, 331.00s/it, avg_loss=0.8326]Checkpoint saved at siamese_crops_checkpoints/siamese
Checkpoint saved at siamese_crops_checkpoints/siamese_crops_best_model.pth
best model updated: 0.8326 at epoch 1

Epoch 2:   0%|          | 0/7 [00:00<?, ?it/s]
Epoch 2:   0%|          | 0/7 [02:58<?, ?it/s, loss=0.8211]
Epoch 2:  14%|█▍        | 1/7 [02:58<17:49, 178.30s/it, loss=0.8211]
Epoch 2:  14%|█▍        | 1/7 [03:00<17:49, 178.30s/it, loss=0.8278]
Epoch 2:  29%|██▊       | 2/7 [03:00<06:14, 74.96s/it, loss=0.8278]
Epoch 2:  29%|██▊       | 2/7 [03:09<06:14, 74.96s/it, loss=0.8301]
Epoch 2:  43%|████▎     | 3/7 [03:09<02:57, 44.46s/it, loss=0.8301]
Epoch 2:  43%|████▎     | 3/7 [03:09<02:57, 44.46s/it, loss=0.8358]
Epoch 2:  57%|█████▋    | 4/7 [03:09<01:21, 27.12s/it, loss=0.8358]
Epoch 2:  57%|█████▋    | 4/7 [04:56<01:21, 27.12s/it, loss=0.8274]
Epoch 2:  71%|███████▏  | 5/7 [04:56<01:51, 55.83s/it, loss=0.8274]
Epoch 2:  71%|███████▏  | 5/7 [05:04<01:51, 55.83s/it, loss=0.8221]
Epoch 2:  86%|████████▌ | 6/7 [05:04<00:39, 39.52s/it, loss=0.8221]
Epoch 2:  86%|████████▌ | 6/7 [05:04<00:39, 39.52s/it, loss=0.8484]
Epoch 2: 100%|██████████| 7/7 [05:04<00:00, 26.67s/it, loss=0.8484]
Epochs:  20%|██        | 2/10 [10:35<42:05, 315.66s/it, avg_loss=0.8304]Checkpoint saved at siamese_crops_checkpoints/siamese
Checkpoint saved at siamese_crops_checkpoints/siamese_crops_best_model.pth
best model updated: 0.8304 at epoch 2

Epoch 3:   0%|          | 0/7 [00:00<?, ?it/s]
Epoch 3:   0%|          | 0/7 [03:07<?, ?it/s, loss=0.8184]
Epoch 3:  14%|█▍        | 1/7 [03:07<18:42, 187.12s/it, loss=0.8184]
Epoch 3:  14%|█▍        | 1/7 [03:07<18:42, 187.12s/it, loss=0.8314]
Epoch 3:  29%|██▊       | 2/7 [03:07<06:26, 77.34s/it, loss=0.8314]
Epoch 3:  29%|██▊       | 2/7 [03:11<06:26, 77.34s/it, loss=0.8089]
Epoch 3:  43%|████▎     | 3/7 [03:11<02:55, 43.78s/it, loss=0.8089]
Epoch 3:  43%|████▎     | 3/7 [03:11<02:55, 43.78s/it, loss=0.8268]
Epoch 3:  57%|█████▋    | 4/7 [03:11<01:20, 26.69s/it, loss=0.8268]
Epoch 3:  57%|█████▋    | 4/7 [05:16<01:20, 26.69s/it, loss=0.8215]
Epoch 3:  71%|███████▏  | 5/7 [05:16<02:03, 61.98s/it, loss=0.8215]
Epoch 3:  71%|███████▏  | 5/7 [05:16<02:03, 61.98s/it, loss=0.8244]
Epoch 3:  86%|████████▌ | 6/7 [05:16<00:41, 41.07s/it, loss=0.8244]
Epoch 3:  86%|████████▌ | 6/7 [05:17<00:41, 41.07s/it, loss=0.8459]
Epoch 3: 100%|██████████| 7/7 [05:17<00:00, 27.71s/it, loss=0.8459]
Epochs:  30%|███       | 3/10 [15:53<36:56, 316.59s/it, avg_loss=0.8254]Checkpoint saved at siamese_crops_checkpoints/siamese
Checkpoint saved at siamese_crops_checkpoints/siamese_crops_best_model.pth
best model updated: 0.8254 at epoch 3
```

```
model_to_save = trainer.model.module if isinstance(trainer.model, torch.nn.DataParallel) else trainer.model
checkpoint = {
    'epoch': 1,
    'model_state_dict': model_to_save.state_dict(),
    'optimizer_state_dict': trainer.optimizer.state_dict(),
    'scaler_state_dict': trainer.scaler.state_dict() if hasattr(trainer, "scaler") else None,
}
# save checkpoint (portable)
torch.save(checkpoint, "siamese_crops_first_model.pth")
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
siamese_model = Siamese()

siamese_model.load_state_dict(
    torch.load("siamese_crops_first_model.pth", map_location='cpu').get("model_state_dict")
```

```
)

```

```
<All keys matched successfully>
```

```
class EmbeddingDataset(Dataset):
    def __init__(self, image_folder_root, siamese_model, device, transform=None, class_to_idx=None):
        self.image_folder_dataset = datasets.ImageFolder(root=image_folder_root, transform=transform)

        if class_to_idx:
            self.image_folder_dataset.class_to_idx = class_to_idx
            self.image_folder_dataset.samples = [
                (s, class_to_idx[c]) for s, c in self.image_folder_dataset.samples if c in class_to_idx
            ]

        self.siamese_model = siamese_model.eval().to(device)
        self.device = device
        self.classes = self.image_folder_dataset.classes
        self.class_to_idx = self.image_folder_dataset.class_to_idx

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

    def __getitem__(self, idx):
        img, label = self.image_folder_dataset[idx]
        with torch.no_grad():
            embedding = self.siamese_model(img.unsqueeze(0).to(self.device))
        return embedding.squeeze(0), label

```

```
class EmbeddingClassifier(nn.Module):
    """A simple linear classifier on top of embeddings."""
    def __init__(self, embedding_dim, num_classes):
        super().__init__()
        self.fc = nn.Linear(embedding_dim, num_classes)

    def forward(self, x):
        return self.fc(x)

```

```
import os

def train_classifier(train_loader, val_loader, model, device, epochs=10, lr=1e-3, checkpoint_dir="/kaggle/working"):
    os.makedirs(checkpoint_dir, exist_ok=True)

    if torch.cuda.device_count() > 1:
        print(f"Using {torch.cuda.device_count()} GPUs for training")
        model = nn.DataParallel(model)

    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    best_val_acc = 0.0

    for epoch in range(epochs):
        model.train()
        total_loss, correct, total = 0, 0, 0
        for emb, labels in train_loader:
            emb, labels = emb.to(device), labels.to(device)
            outputs = model(emb)
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item() * emb.size(0)
            _, preds = torch.max(outputs, 1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

        train_loss = total_loss / total
        train_acc = correct / total

        # Validation
        model.eval()
        val_correct, val_total = 0, 0
        val_loss_total = 0
        with torch.no_grad():
            for emb, labels in val_loader:
                emb, labels = emb.to(device), labels.to(device)
```

```
                outputs = model(emb)
                loss = criterion(outputs, labels)
                val_loss_total += loss.item() * emb.size(0)
                _, preds = torch.max(outputs, 1)
                val_correct += (preds == labels).sum().item()
                val_total += labels.size(0)

        val_loss = val_loss_total / val_total
        val_acc = val_correct / val_total

        print(f"Epoch [{epoch+1}/{epochs}] - "
              f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
              f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

        latest_path = os.path.join(checkpoint_dir, "siamese_classifier_last.pth")
        torch.save({
            'epoch': epoch + 1,
            'model_state_dict': model.module.state_dict() if isinstance(model, nn.DataParallel) else model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'val_loss': val_loss,
            'val_acc': val_acc
        }, latest_path)

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_path = os.path.join(checkpoint_dir, "siamese_classifier_best.pth")
            torch.save({
                'epoch': epoch + 1,
                'model_state_dict': model.module.state_dict() if isinstance(model, nn.DataParallel) else model.state_dic
                'optimizer_state_dict': optimizer.state_dict(),
                'val_loss': val_loss,
                'val_acc': val_acc
            }, best_path)
            print(f"--> Best model updated (Val Acc: {best_val_acc:.4f})")

    # Unwrap model for final return
    model_to_save = model.module if isinstance(model, nn.DataParallel) else model
```

```
import gc
gc.collect()
os.environ['CUDA_LAUNCH_BLOCKING']='1'
os.environ['TORCH_USE_CUDA_DSA'] = '1'
!echo $CUDA_LAUNCH_BLOCKING
```

```
1
```

```
IMAGE_SIZE = 224
BATCH_SIZE = 64

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

train_dataset = EmbeddingDataset("dataset/train", siamese_model, device, transform)

class_to_idx_map = train_dataset.class_to_idx
num_classes = len(train_dataset.classes)

val_dataset = EmbeddingDataset("dataset/val", siamese_model, device, transform)

```