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

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

```
import os

base_dir = "/content/drive/MyDrive/cassava"

# Creates directories
os.makedirs(f"{base_dir}/src", exist_ok=True)
os.makedirs(f"{base_dir}/outputs/models", exist_ok=True)
os.makedirs(f"{base_dir}/outputs/logs", exist_ok=True)
os.makedirs(f"{base_dir}/training_metadata", exist_ok=True)

print(" Directories created!")
```

```
 Directories created!
```

```
%%writefile /content/drive/MyDrive/cassava/src/advanced_training.py
"""
Advanced Training Improvements for Maximum Accuracy

Key improvements:
1. Advanced data augmentation strategies
2. Progressive unfreezing and learning rate warm-up
3. Label smoothing for better generalization
4. Test-time augmentation (TTA)
5. Model ensembling capabilities
6. Advanced regularization techniques
"""

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import OneCycleLR
import random


class ImprovedTrainingConfig:
    """Enhanced training configuration for better accuracy."""

    # Advanced augmentation strategy
    AUGMENTATION_STRATEGIES = {
        'mild': {
            'rotation': 15,
            'brightness': 0.1,
            'contrast': 0.1,
            'saturation': 0.1,
            'hue': 0.05,
            'zoom': 0.1,
            'flip_horizontal': True,
            'flip_vertical': False,
            'cutout': False,
            'mixup': False
        },
        'moderate': {
            'rotation': 25,
            'brightness': 0.2,
            'contrast': 0.2,
            'saturation': 0.2,
            'hue': 0.1,
            'zoom': 0.15,
            'flip_horizontal': True,
            'flip_vertical': True,
            'cutout': True,
            'cutout_size': 16,
            'mixup': False
        },
        'aggressive': {
            'rotation': 35,
            'brightness': 0.3,
            'contrast': 0.3,
            'saturation': 0.3,
            'hue': 0.15,
            'zoom': 0.2,
            'flip_horizontal': True,
            'flip_vertical': True,
            'cutout': True,
            'cutout_size': 24,
            'mixup': True,
            'mixup_alpha': 0.2
        }
    }

    # Optimizer configurations
    OPTIMIZER_CONFIGS = {
        'adamw': {
            'lr': 3e-4,
            'weight_decay': 0.01,
            'betas': (0.9, 0.999)
        },
        'sgd': {
            'lr': 0.01,
            'momentum': 0.9,
            'weight_decay': 0.0001,
            'nesterov': True
        },
        'radam': {
            'lr': 1e-3,
            'weight_decay': 0.01,
            'betas': (0.9, 0.999)
        }
    }

    # Learning rate scheduler configurations
    SCHEDULER_CONFIGS = {
        'cosine_warm_restarts': {
            'T_0': 5,
            'T_mult': 2,
            'eta_min': 1e-6
        },
        'one_cycle': {
            'max_lr': 1e-3,
            'pct_start': 0.3,
            'anneal_strategy': 'cos',
            'div_factor': 25,
            'final_div_factor': 1e4
        },
        'reduce_on_plateau': {
            'mode': 'max',
            'factor': 0.5,
            'patience': 3,
            'min_lr': 1e-7
        }
    }


def create_advanced_optimizer(model, config_name='adamw'):
    """Create optimizer with advanced configuration."""
    config = ImprovedTrainingConfig.OPTIMIZER_CONFIGS[config_name]

    # Separate parameters for different learning rates
    backbone_params = []
    classifier_params = []

    for name, param in model.named_parameters():
        if 'classifier' in name:
            classifier_params.append(param)
        else:
            backbone_params.append(param)

    if config_name == 'adamw':
        optimizer = optim.AdamW([
            {'params': backbone_params, 'lr': config['lr'] * 0.1},  # Lower LR for backbone
            {'params': classifier_params, 'lr': config['lr']}  # Higher LR for classifier
        ], weight_decay=config['weight_decay'], betas=config['betas'])
    elif config_name == 'sgd':
        optimizer = optim.SGD([
            {'params': backbone_params, 'lr': config['lr'] * 0.1},
            {'params': classifier_params, 'lr': config['lr']}
        ], momentum=config['momentum'],
           weight_decay=config['weight_decay'],
           nesterov=config['nesterov'])

    return optimizer


def create_advanced_scheduler(optimizer, scheduler_name='one_cycle', epochs=15, steps_per_epoch=100):
    """Create learning rate scheduler."""
    config = ImprovedTrainingConfig.SCHEDULER_CONFIGS[scheduler_name]

    if scheduler_name == 'one_cycle':
        scheduler = OneCycleLR(
            optimizer,
            max_lr=[config['max_lr'] * 0.1, config['max_lr']],  # Different LR for each param group
            epochs=epochs,
            steps_per_epoch=steps_per_epoch,
            pct_start=config['pct_start'],
            anneal_strategy=config['anneal_strategy'],
            div_factor=config['div_factor'],
            final_div_factor=config['final_div_factor']
        )
    elif scheduler_name == 'cosine_warm_restarts':
        scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(
            optimizer,
            T_0=config['T_0'],
            T_mult=config['T_mult'],
            eta_min=config['eta_min']
        )
    elif scheduler_name == 'reduce_on_plateau':
        scheduler = optim.lr_scheduler.ReduceLROnPlateau(
            optimizer,
            mode=config['mode'],
            factor=config['factor'],
            patience=config['patience'],
            min_lr=config['min_lr']
        )

    return scheduler


class LabelSmoothingCrossEntropy(nn.Module):
    """Label smoothing for better generalization."""

    def __init__(self, smoothing=0.1):
        super().__init__()
        self.smoothing = smoothing

    def forward(self, pred, target):
        n_classes = pred.size(-1)
        log_preds = torch.nn.functional.log_softmax(pred, dim=-1)

        # Create smooth labels
        with torch.no_grad():
            true_dist = torch.zeros_like(log_preds)
            true_dist.fill_(self.smoothing / (n_classes - 1))
            true_dist.scatter_(1, target.unsqueeze(1), 1.0 - self.smoothing)

        return torch.mean(torch.sum(-true_dist * log_preds, dim=-1))


class MixupAugmentation:
    """Mixup augmentation for improved generalization."""

    def __init__(self, alpha=0.2):
        self.alpha = alpha

    def __call__(self, images, labels):
        if self.alpha > 0:
            lam = np.random.beta(self.alpha, self.alpha)
        else:
            lam = 1

        batch_size = images.size(0)
        index = torch.randperm(batch_size).to(images.device)

        mixed_images = lam * images + (1 - lam) * images[index]
        labels_a, labels_b = labels, labels[index]

        return mixed_images, labels_a, labels_b, lam


def test_time_augmentation(model, image, device, n_augmentations=5):
    """
    Test-time augmentation for more robust predictions.

    Args:
        model: Trained model
        image: Input image tensor (C, H, W)
        device: Device to run on
        n_augmentations: Number of augmentations to apply

    Returns:
        Average predictions across augmentations
    """
    import torchvision.transforms.functional as TF

    model.eval()
    predictions = []

    with torch.no_grad():
        # Original image
        img_tensor = image.unsqueeze(0).to(device)
        output = model(img_tensor)
        predictions.append(torch.nn.functional.softmax(output, dim=1))

        # Augmented versions
        for _ in range(n_augmentations - 1):
            # Random horizontal flip
            if random.random() > 0.5:
                img_aug = TF.hflip(image)
            else:
                img_aug = image

            # Random rotation
            angle = random.choice([-10, -5, 0, 5, 10])
            img_aug = TF.rotate(img_aug, angle)
```

```
                # Predict
                img_tensor = img_aug.unsqueeze(0).to(device)
                output = model(img_tensor)
                predictions.append(torch.nn.functional.softmax(output, dim=1))

        # Average predictions
        avg_prediction = torch.stack(predictions).mean(dim=0)
        return avg_prediction


    class EarlyStopping:
        """Early stopping with patience and model checkpoint."""

        def __init__(self, patience=7, min_delta=0.001, mode='max'):
            self.patience = patience
            self.min_delta = min_delta
            self.mode = mode
            self.counter = 0
            self.best_score = None
            self.early_stop = False

        def __call__(self, score):
            if self.best_score is None:
                self.best_score = score
            elif self.mode == 'max':
                if score < self.best_score + self.min_delta:
                    self.counter += 1
                    if self.counter >= self.patience:
                        self.early_stop = True
                else:
                    self.best_score = score
                    self.counter = 0
            elif self.mode == 'min':
                if score > self.best_score - self.min_delta:
                    self.counter += 1
                    if self.counter >= self.patience:
                        self.early_stop = True
                else:
                    self.best_score = score
                    self.counter = 0

            return self.early_stop


    class GradualUnfreezing:
        """Progressive unfreezing of model layers."""

        def __init__(self, model, unfreeze_schedule):
            """
            Args:
                model: PyTorch model
                unfreeze_schedule: Dict mapping epoch to layers to unfreeze
                    Example: {0: 0.7, 5: 0.5, 10: 0.0}  # 0.0 means unfreeze all
            """
            self.model = model
            self.unfreeze_schedule = unfreeze_schedule

        def step(self, epoch):
            if epoch in self.unfreeze_schedule:
                freeze_ratio = self.unfreeze_schedule[epoch]

                if freeze_ratio == 0.0:
                    # Unfreeze all layers
                    for param in self.model.parameters():
                        param.requires_grad = True
                    print(f"[INFO] Epoch {epoch}: All layers unfrozen")
                else:
                    # Unfreeze based on ratio
                    total_params = sum(1 for _ in self.model.backbone.parameters())
                    freeze_count = int(total_params * freeze_ratio)

                    for idx, param in enumerate(self.model.backbone.parameters()):
                        param.requires_grad = (idx >= freeze_count)

                    print(f"[INFO] Epoch {epoch}: Unfrozen {total_params - freeze_count}/{total_params} backbone 


    # Training recommendations
    TRAINING_RECOMMENDATIONS = """
    🎯 RECOMMENDATIONS FOR MAXIMUM ACCURACY:

    1. **Data Quality**
       - Ensure balanced classes (use class weights if imbalanced)
       - Remove corrupted/mislabeled images
       - Augment minority classes more aggressively

    2. **Model Architecture**
       - Use EfficientNet-B3 for better accuracy (vs B0)
       - Experiment with different dropout rates (0.3-0.5)
       - Try ensemble of 3-5 models

    3. **Training Strategy**
       - Start with frozen backbone (5 epochs)
       - Gradually unfreeze layers (epochs 5-10)
       - Fine-tune all layers (epochs 10+)
       - Use OneCycleLR scheduler for faster convergence

    4. **Regularization**
       - Label smoothing (0.1)
       - Mixup augmentation (alpha=0.2)
       - Dropout (0.4)
       - Weight decay (0.01)

    5. **Optimization**
       - AdamW optimizer with differential learning rates
       - Batch size: 32-64 (depending on GPU memory)
       - Mixed precision training (AMP)

    6. **Data Augmentation**
       - Use 'moderate' or 'aggressive' strategy
       - Test-time augmentation (TTA) for inference
       - Random erasing/cutout

    7. **Training Duration**
       - 20-30 epochs with early stopping
       - Save best model based on validation accuracy

    8. **Inference**
       - Use test-time augmentation (5-10 variations)
       - Ensemble multiple models
       - Calibrate probabilities if needed

    Expected Performance:
    - With these improvements: 85-95% validation accuracy
    - Baseline (simple training): 70-80% validation accuracy
    """


    if __name__ == "__main__":
        print(TRAINING_RECOMMENDATIONS)
```

```
Overwriting /content/drive/MyDrive/cassava/src/advanced_training.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/test_classical.py
```

```
import os
import argparse
import torch
import joblib
import numpy as np
from torchvision import transforms
from PIL import Image
import gradio as gr
from pathlib import Path
import glob


# Import feature extractor
import sys
sys.path.append(os.path.dirname(os.path.abspath(__file__)))


class FeatureExtractor:
    """Extract features using pre-trained CNN."""

    def __init__(self, device='cuda'):
        self.device = torch.device(device if torch.cuda.is_available() else 'cpu')

        # Load pre-trained EfficientNet
        from torchvision.models import efficientnet_b0, EfficientNet_B0_Weights
        weights = EfficientNet_B0_Weights.IMAGENET1K_V1
        self.model = efficientnet_b0(weights=weights)

        # Remove classifier to get features
        self.model.classifier = torch.nn.Identity()
        self.model = self.model.to(self.device)
        self.model.eval()

        # Define transforms
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
        ])

        print(f"[INFO] Feature extractor loaded on {self.device}")

    def extract_features(self, image):
        """Extract features from a single image."""
        img_tensor = self.transform(image).unsqueeze(0).to(self.device)

        with torch.no_grad():
            features = self.model(img_tensor)

        return features.cpu().numpy().flatten()


class ClassicalMLPredictor:
    """Wrapper class for classical ML model prediction."""

    def __init__(self, model_path, device='cuda'):
        self.device = torch.device(device if torch.cuda.is_available() else 'cpu')
        print(f"[INFO] Using device: {self.device}")

        # Load model with joblib (not torch.load!)
        print(f"[INFO] Loading classical ML model from: {model_path}")
        checkpoint = joblib.load(model_path)

        # Extract model components
        self.model = checkpoint['model']
        self.scaler = checkpoint['scaler']
        self.class_names = checkpoint.get('class_names', [])
```

```
            print(f"[INFO] Model type: {type(self.model).__name__}")
            print(f"[INFO] Number of estimators: {checkpoint.get('n_estimators', 'N/A')}")
            print(f"[INFO] Classes: {self.class_names}")

            # Initialize feature extractor
            print(f"[INFO] Initializing feature extractor...")
            self.feature_extractor = FeatureExtractor(device=self.device)

            print(f"[INFO] Classical ML model loaded successfully!")

        def predict(self, image):
            """
            Predict class for a single image.

            Args:
                image: PIL Image or numpy array

            Returns:
                predictions: dict with class names and probabilities
            """
            # Convert to PIL if numpy array
            if isinstance(image, np.ndarray):
                image = Image.fromarray(image).convert('RGB')
            elif not isinstance(image, Image.Image):
                raise ValueError("Image must be PIL Image or numpy array")

            # Extract features
            features = self.feature_extractor.extract_features(image)

            # Scale features
            features_scaled = self.scaler.transform(features.reshape(1, -1))

            # Predict probabilities
            probabilities = self.model.predict_proba(features_scaled)[0]

            # Create predictions dictionary
            predictions = {
                self.class_names[i]: float(probabilities[i]) * 100
                for i in range(len(self.class_names))
            }

            return predictions

        def predict_with_details(self, image):
            """
            Predict with additional details.

            Returns:
                predictions: dict with probabilities
                top_class: name of top predicted class
                confidence: confidence percentage
            """
            predictions = self.predict(image)
            top_class = max(predictions, key=predictions.get)
            confidence = predictions[top_class]

            return predictions, top_class, confidence


    def find_latest_model(work_dir, user_id=None):
        """Find the latest trained classical model."""
        models_dir = os.path.join(work_dir, "outputs", "models")

        if not os.path.exists(models_dir):
            return None

        # Search pattern for .pkl files
        if user_id:
            pattern = os.path.join(models_dir, user_id, "**", "*.pkl")
        else:
            pattern = os.path.join(models_dir, "**", "*.pkl")

        model_files = glob.glob(pattern, recursive=True)

        # Filter for random forest models
        model_files = [f for f in model_files if "random_forest" in f]

        if not model_files:
            return None

        # Get most recent
        latest_model = max(model_files, key=os.path.getmtime)
        return latest_model


    def create_gradio_interface(model_path, share=False, server_name="0.0.0.0", server_port=7860):
        """Create and launch Gradio interface."""

        # Initialize predictor
        predictor = ClassicalMLPredictor(model_path)

        # Define prediction function for Gradio
        def predict_image(image):
            if image is None:
                return "⚠️ Please upload an image!", None

            try:
                predictions, top_class, confidence = predictor.predict_with_details(image)

                # Format result text
                result_text = f"### 🎯 Prediction: **{top_class}**\n"
                result_text += f"### 📊 Confidence: **{confidence:.2f}%**\n"
                result_text += f"### 🤖 Model: **Classical ML (Random Forest)**\n\n"

                # Confidence indicator
                if confidence > 90:
                    result_text += "#### ✅ Very High Confidence\n\n"
                elif confidence > 75:
                    result_text += "#### ✔️ High Confidence\n\n"
                elif confidence > 60:
                    result_text += "#### ⚠️ Moderate Confidence\n\n"
                else:
                    result_text += "#### ⚠️ Low Confidence - Review Needed\n\n"

                result_text += "#### All Probabilities:\n"

                # Sort predictions by probability
                sorted_preds = sorted(predictions.items(), key=lambda x: x[1], reverse=True)
                for class_name, prob in sorted_preds:
                    bar = '█' * int(prob / 2)
                    result_text += f"- **{class_name}**: {prob:.2f}% {bar}\n"

                return result_text, predictions

            except Exception as e:
                return f"❌ Error: {str(e)}", None

        # Create disease info
        disease_info = """
        ## 🌿 Cassava Disease Information

        ### Common Cassava Diseases:

        1. **🦠 Cassava Mosaic Disease (CMD)**
           - Caused by cassava mosaic virus
```

```
   - Symptoms: Mosaic patterns on leaves, stunted growth
   - Management: Use resistant varieties, control whiteflies

2. **🟤 Cassava Brown Streak Disease (CBSD)**
   - Causes brown streaks on stems and roots
   - Severely affects root quality
   - Management: Plant resistant varieties, remove infected plants

3. **🦠 Cassava Bacterial Blight (CBB)**
   - Bacterial infection causing wilting and blight
   - Can lead to total crop loss
   - Management: Use disease-free cuttings, crop rotation

4. **🐛 Cassava Green Mite (CGM)**
   - Pest infestation causing leaf damage
   - Reduces photosynthesis and yield
   - Management: Biological control, resistant varieties

5. **✅ Healthy**
   - No visible disease symptoms
   - Optimal growing conditions
"""

# Create Gradio interface
with gr.Blocks(title="🌿 Cassava Disease Classifier (Classical ML)", theme=gr.themes.Soft()) as interface
    gr.Markdown("# 🌿 Cassava Disease Classification - Classical ML")
    gr.Markdown("**AI-Powered Disease Detection using Random Forest**")

    with gr.Row():
        with gr.Column(scale=1):
            gr.Markdown("### 📤 Upload Image")
            input_image = gr.Image(type="pil", label="Cassava Leaf Image")

            with gr.Row():
                clear_btn = gr.Button("🗑️ Clear", variant="secondary")
                predict_btn = gr.Button("🔍 Classify Disease", variant="primary", size="lg")

            gr.Markdown("**Supported:** JPG, JPEG, PNG")

        with gr.Column(scale=1):
            gr.Markdown("### 🎯 Results")
            output_text = gr.Markdown(label="Prediction")
            output_plot = gr.Label(label="Confidence Scores", num_top_classes=5)

    with gr.Row():
        with gr.Column():
            with gr.Accordion("📖 Disease Information", open=False):
                gr.Markdown(disease_info)

    # Model information
    with gr.Accordion("ℹ️ Model Information", open=False):
        model_info = f"""
        ### Classical ML Model Details
        - **Algorithm:** Random Forest
        - **Feature Extractor:** Pre-trained EfficientNet-B0
        - **Model Path:** `{model_path}`
        - **Classes:** {', '.join(predictor.class_names)}
        - **Device:** {predictor.device}
        """
        gr.Markdown(model_info)

    # Connect prediction function
    predict_btn.click(
        fn=predict_image,
        inputs=input_image,
        outputs=[output_text, output_plot]
    )
    clear_btn.click(fn=lambda: (None, "", None), outputs=[input_image, output_text, output_plot])
```

```
        print("\n" + "="*70)
        print("🚀 Launching Classical ML Gradio Interface")
        print("="*70)
        print(f"Model: {os.path.basename(model_path)}")
        print(f"Classes: {predictor.class_names}")
        print(f"Device: {predictor.device}")
        print(f"Port: {server_port}")
        print(f"Share: {'Yes' if share else 'No'}")
        print("="*70 + "\n")

        # Launch interface
        interface.launch(
            share=share,
            server_name=server_name,
            server_port=server_port,
            show_error=True
        )


    def main():
        parser = argparse.ArgumentParser(description="Test Classical ML Cassava Disease Classification Model")

        parser.add_argument("--model_path", type=str, default=None,
                           help="Path to trained model (.pkl file)")
        parser.add_argument("--work_dir", type=str,
                           default="/content/drive/MyDrive/cassava",
                           help="Working directory (to auto-find latest model)")
        parser.add_argument("--user_id", type=str, default=None,
                           help="User ID to find specific model")
        parser.add_argument("--share", action="store_true",
                           help="Create public sharing link")
        parser.add_argument("--server_name", type=str, default="0.0.0.0",
                           help="Server name for Gradio")
        parser.add_argument("--server_port", type=int, default=7860,
                           help="Server port for Gradio")
        parser.add_argument("--device", type=str, default="cuda",
                           choices=["cuda", "cpu"],
                           help="Device to run inference on")

        args = parser.parse_args()

        # Find model path
        if args.model_path:
            model_path = args.model_path
        else:
            print("[INFO] Searching for latest trained classical model...")
            model_path = find_latest_model(args.work_dir, args.user_id)

            if not model_path:
                print("[ERROR] No trained classical model found!")
                print(f"[INFO] Please train a model first or specify --model_path")
                return

        if not os.path.exists(model_path):
            print(f"[ERROR] Model not found at: {model_path}")
            return

        print(f"[INFO] Using model: {model_path}")

        # Create and launch interface
        create_gradio_interface(
            model_path=model_path,
            share=args.share,
            server_name=args.server_name,
            server_port=args.server_port
        )
```

```
if __name__ == "__main__":
    main()
```

```
Overwriting /content/drive/MyDrive/cassava/src/test_classical.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/test_deep_learning.py
import os
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import gradio as gr
import numpy as np


# Import model architecture
import sys
sys.path.append(os.path.dirname(os.path.abspath(__file__)))


class SimpleClassifierWrapper(nn.Module):
    """
    Wrapper that directly loads the classifier as a Sequential from state_dict.
    No reconstruction needed - just load what exists!
    """
    def __init__(self, state_dict_classifier):
        super().__init__()
        # Store the layers directly from state dict
        self.layers = nn.ModuleDict()

        # Extract classifier layers
        classifier_keys = sorted([k for k in state_dict_classifier.keys() if k.startswith('classifier.')])
        layer_indices = sorted(set(int(k.split('.')[1]) for k in classifier_keys))

        print(f"[INFO] Loading {len(layer_indices)} classifier layers from checkpoint")

        for idx in layer_indices:
            weight_key = f'classifier.{idx}.weight'

            if weight_key in state_dict_classifier:
                weight_shape = state_dict_classifier[weight_key].shape

                if len(weight_shape) == 2:  # Linear layer
                    out_f, in_f = weight_shape
                    layer = nn.Linear(in_f, out_f)
                    self.layers[str(idx)] = layer
                    print(f"[INFO]   Layer {idx}: Linear({in_f} → {out_f})")

                elif len(weight_shape) == 1:  # BatchNorm
                    n_f = weight_shape[0]
                    layer = nn.BatchNorm1d(n_f)
                    self.layers[str(idx)] = layer
                    print(f"[INFO]   Layer {idx}: BatchNorm1d({n_f})")

            elif f'classifier.{idx}.running_mean' in state_dict_classifier:
                # BatchNorm without weight in key list
                n_f = state_dict_classifier[f'classifier.{idx}.running_mean'].shape[0]
                layer = nn.BatchNorm1d(n_f)
                self.layers[str(idx)] = layer
                print(f"[INFO]   Layer {idx}: BatchNorm1d({n_f})")

    def forward(self, x):
        # Apply layers in order with ReLU activations between
        layer_names = sorted(self.layers.keys(), key=int)

        for i, name in enumerate(layer_names):
            x = self.layers[name](x)

            # Add ReLU after Linear layers (except last)
            if isinstance(self.layers[name], nn.Linear) and i < len(layer_names) - 1:
                x = torch.relu(x)

        return x


class FlexibleCassavaModel(nn.Module):
    """Model that loads any saved architecture."""

    def __init__(self, num_classes=5, model_type="efficientnet_b0", state_dict=None):
        super().__init__()

        from torchvision.models import (
            efficientnet_b0, efficientnet_b3, efficientnet_b4,
            efficientnet_v2_s, efficientnet_v2_m,
            EfficientNet_B0_Weights, EfficientNet_B3_Weights,
            EfficientNet_B4_Weights, EfficientNet_V2_S_Weights,
            EfficientNet_V2_M_Weights
        )

        # Load backbone
        if model_type == "efficientnet_b4":
            self.backbone = efficientnet_b4(weights=EfficientNet_B4_Weights.IMAGENET1K_V1)
        elif model_type == "efficientnet_v2_s":
            self.backbone = efficientnet_v2_s(weights=EfficientNet_V2_S_Weights.IMAGENET1K_V1)
        elif model_type == "efficientnet_v2_m":
            self.backbone = efficientnet_v2_m(weights=EfficientNet_V2_M_Weights.IMAGENET1K_V1)
        elif model_type == "efficientnet_b3":
            self.backbone = efficientnet_b3(weights=EfficientNet_B3_Weights.IMAGENET1K_V1)
        else:  # efficientnet_b0
            self.backbone = efficientnet_b0(weights=EfficientNet_B0_Weights.IMAGENET1K_V1)

        # Remove original classifier
        self.backbone.classifier = nn.Identity()

        # Build classifier from state dict
        if state_dict:
            self.classifier = SimpleClassifierWrapper(state_dict)
        else:
            raise ValueError("state_dict required for loading")

    def forward(self, x):
        features = self.backbone(x)
        output = self.classifier(features)
        return output


class DeepLearningPredictor:
    """Wrapper class for deep learning model prediction."""

    def __init__(self, model_path, device='cuda'):
        self.device = torch.device(device if torch.cuda.is_available() else 'cpu')
        print(f"[INFO] Using device: {self.device}")

        # Load checkpoint
        print(f"[INFO] Loading deep learning model from: {model_path}")
        checkpoint = torch.load(model_path, map_location=self.device, weights_only=False)

        # Extract model info
        self.class_names = checkpoint.get('class_names', [])
        num_classes = checkpoint.get('num_classes', len(self.class_names))
        model_type = checkpoint.get('model_type', 'efficientnet_b0')

        print(f"[INFO] Model architecture: {model_type}")
        print(f"[INFO] Number of classes: {num_classes}")
        # Build model
        state_dict = checkpoint['model_state_dict']
        self.model = FlexibleCassavaModel(
            num_classes=num_classes,
            model_type=model_type,
            state_dict=state_dict
        )

        # Load weights
        print("[INFO] Loading model weights...")

        # Load backbone weights (these should match perfectly)
        backbone_state = {k.replace('backbone.', ''): v for k, v in state_dict.items()
                         if k.startswith('backbone.')}
        self.model.backbone.load_state_dict(backbone_state, strict=False)

        # Load classifier weights
        classifier_state = {k.replace('classifier.', 'layers.'): v for k, v in state_dict.items()
                           if k.startswith('classifier.')}

        # Map to correct structure
        mapped_state = {}
        for key, value in classifier_state.items():
            # Extract layer index
            parts = key.split('.')
            if len(parts) >= 2:
                layer_idx = parts[1]
                param_name = '.'.join(parts[2:])
                new_key = f'layers.{layer_idx}.{param_name}'
                mapped_state[new_key] = value

        self.model.classifier.load_state_dict(mapped_state, strict=False)

        print(f"[INFO] ✓ Model loaded successfully!")

        self.model = self.model.to(self.device)
        self.model.eval()

        # Define transforms
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                               std=[0.229, 0.224, 0.225])
        ])

        print(f"[INFO] Deep learning model ready!")
        print(f"[INFO] Classes: {self.class_names}")

    def predict(self, image):
        """Predict class for a single image."""
        # Convert to PIL if numpy array
        if isinstance(image, np.ndarray):
            image = Image.fromarray(image).convert('RGB')

        # Preprocess
        img_tensor = self.transform(image).unsqueeze(0).to(self.device)

        # Predict
        with torch.no_grad():
            outputs = self.model(img_tensor)
            probabilities = torch.nn.functional.softmax(outputs, dim=1)[0]

        # Create predictions dictionary
        predictions = {
            self.class_names[i]: float(probabilities[i]) * 100
            for i in range(len(self.class_names))
            }

        return predictions


def create_gradio_interface(model_path, share=False, server_name="0.0.0.0", server_port=7860):
    """Create and launch Gradio interface for deep learning model."""

    # Initialize predictor
    predictor = DeepLearningPredictor(model_path)

    # Disease information
    disease_info = get_disease_info()

    # Define prediction function
    def predict_image(image):
        if image is None:
            return "⚠️ Please upload an image!", None

        try:
            predictions = predictor.predict(image)

            # Get top prediction
            top_class = max(predictions, key=predictions.get)
            confidence = predictions[top_class]

            # Format result text
            result_text = f"### 🎯 Prediction: **{top_class}**\n"
            result_text += f"### 📊 Confidence: **{confidence:.2f}%**\n"
            result_text += f"### 🤖 Model: **Deep Learning (PyTorch EfficientNet)**\n\n"

            # Confidence indicator
            if confidence > 90:
                result_text += "#### ✅ Very High Confidence\n\n"
            elif confidence > 75:
                result_text += "#### ✔️ High Confidence\n\n"
            elif confidence > 60:
                result_text += "#### ⚠️ Moderate Confidence\n\n"
            else:
                result_text += "#### ⚠️ Low Confidence - Review Needed\n\n"

            result_text += "#### All Probabilities:\n"

            # Sort predictions
            sorted_preds = sorted(predictions.items(), key=lambda x: x[1], reverse=True)
            for class_name, prob in sorted_preds:
                bar = '█' * int(prob / 2)
                result_text += f"- **{class_name}**: {prob:.2f}% {bar}\n"

            return result_text, predictions

        except Exception as e:
            return f"❌ Error: {str(e)}", None

    # Create interface
    with gr.Blocks(title="🌿 Cassava Disease Classifier (Deep Learning)", theme=gr.themes.Soft()) as interfac
        gr.Markdown("# 🌿 Cassava Disease Classification - Deep Learning")
        gr.Markdown("**AI-Powered Disease Detection using PyTorch EfficientNet**")

        with gr.Row():
            with gr.Column(scale=1):
                gr.Markdown("### 📤 Upload Image")
                input_image = gr.Image(type="pil", label="Cassava Leaf Image")

                with gr.Row():
                    clear_btn = gr.Button("🗑️ Clear", variant="secondary")
                    predict_btn = gr.Button("🔍 Classify Disease", variant="primary", size="lg")
                gr.Markdown("**Supported:** JPG, JPEG, PNG")

            with gr.Column(scale=1):
                gr.Markdown("### 🎯 Results")
                output_text = gr.Markdown(label="Prediction")
                output_plot = gr.Label(label="Confidence Scores", num_top_classes=5)

        with gr.Row():
            with gr.Column():
                with gr.Accordion("📖 Disease Information", open=False):
                    gr.Markdown(disease_info)

        with gr.Accordion("ℹ️ Model Information", open=False):
            model_info = f"""
            ### Deep Learning Model Details
            - **Framework:** PyTorch
            - **Architecture:** EfficientNet with Flexible Classifier
            - **Model Path:** `{model_path}`
            - **Classes:** {', '.join(predictor.class_names)}
            - **Device:** {predictor.device}
            - **Input Size:** 224x224 pixels
            """
            gr.Markdown(model_info)

        # Connect buttons
        predict_btn.click(fn=predict_image, inputs=input_image, outputs=[output_text, output_plot])
        clear_btn.click(fn=lambda: (None, "", None), outputs=[input_image, output_text, output_plot])

    print(f"\n{'='*70}")
    print("🚀 Launching Deep Learning Gradio Interface")
    print(f"{'='*70}")
    print(f"Model: {os.path.basename(model_path)}")
    print(f"Device: {predictor.device}")
    print(f"Port: {server_port}")
    print(f"Share: {'Yes' if share else 'No'}")
    print(f"{'='*70}\n")

    interface.launch(share=share, server_name=server_name, server_port=server_port, show_error=True)


def get_disease_info():
    """Return disease information markdown."""
    return """
    ## 🌿 Cassava Disease Information

    ### Disease Classes:

    1. **🦠 Cassava Mosaic Disease (CMD)**
       - **Symptoms**: Mosaic patterns, stunted growth
       - **Cause**: Cassava mosaic virus (whitefly transmission)
       - **Management**: Resistant varieties, whitefly control

    2. **🟤 Cassava Brown Streak Disease (CBSD)**
       - **Symptoms**: Brown streaks on stems, root necrosis
       - **Cause**: Cassava brown streak virus
       - **Management**: Clean planting material, vector control

    3. **🦠 Cassava Bacterial Blight (CBB)**
       - **Symptoms**: Wilting, angular leaf spots, stem cankers
       - **Cause**: Xanthomonas axonopodis bacteria
       - **Management**: Disease-free cuttings, crop rotation

    4. **🐛 Cassava Green Mite (CGM)**
       - **Symptoms**: Yellowing leaves, stunted growth
       - **Cause**: Mononychellus tanajoa mite
       - **Management**: Biological control, resistant varieties

    5. **✅ Healthy**
```

```
        - **Characteristics**: Green vigorous leaves, normal growth
        - **Maintenance**: Regular monitoring, proper nutrition
    """
```

```
Overwriting /content/drive/MyDrive/cassava/src/test_deep_learning.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/dataset.py
import torch
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
from PIL import Image
import os
import pandas as pd
from datetime import datetime
import random


class CassavaDataset(Dataset):
    """Custom Dataset for Cassava images."""

    def __init__(self, data_dir, transform=None, specific_classes=None):
        self.data_dir = data_dir
        self.transform = transform
        self.image_paths = []
        self.labels = []
        self.class_to_idx = {}

        # Get classes (folders)
        classes = sorted([d for d in os.listdir(data_dir)
                         if os.path.isdir(os.path.join(data_dir, d)) and not d.startswith('.')])

        # Filter specific classes if requested
        if specific_classes:
            classes = [c for c in classes if c in specific_classes]

        # Create class to index mapping
        self.class_to_idx = {cls_name: idx for idx, cls_name in enumerate(classes)}
        self.idx_to_class = {idx: cls_name for cls_name, idx in self.class_to_idx.items()}

        # Load all image paths and labels
        for class_name in classes:
            class_dir = os.path.join(data_dir, class_name)
            class_idx = self.class_to_idx[class_name]

            for img_name in os.listdir(class_dir):
                if img_name.lower().endswith(('.jpg', '.jpeg', '.png')):
                    img_path = os.path.join(class_dir, img_name)
                    self.image_paths.append(img_path)
                    self.labels.append(class_idx)

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

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        label = self.labels[idx]

        # Load image
        image = Image.open(img_path).convert('RGB')

        # Apply transforms
        if self.transform:
            image = self.transform(image)

        return image, label
```

```
def get_transforms(augment=None, is_training=True):
    """
    Get image transformations.

    Args:
        augment: Dictionary with augmentation parameters
        is_training: Whether this is for training (vs validation)

    Returns:
        transforms: torchvision transforms
    """
    if is_training and augment:
        transform_list = [
            transforms.Resize((256, 256)),
            transforms.RandomCrop(224),
        ]

        # Add augmentations
        if augment.get('rotation', 0) > 0:
            transform_list.append(transforms.RandomRotation(augment['rotation']))

        if augment.get('flip_horizontal', False):
            transform_list.append(transforms.RandomHorizontalFlip(p=0.5))

        if augment.get('flip_vertical', False):
            transform_list.append(transforms.RandomVerticalFlip(p=0.5))

        if augment.get('brightness', 0) > 0:
            brightness_factor = augment['brightness']
            transform_list.append(
                transforms.ColorJitter(brightness=brightness_factor)
            )

        # Add final transforms
        transform_list.extend([
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
        ])

    else:
        # Validation transforms (no augmentation)
        transform_list = [
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
        ]

    return transforms.Compose(transform_list)


def load_data(data_config, batch_size=32, augment=None, user_id=None, num_workers=4):
    """
    Load and prepare data loaders.

    Args:
        data_config: Dictionary with dataset configuration
        batch_size: Batch size for training
        augment: Augmentation configuration dict
        user_id: User identifier for metadata export
        num_workers: Number of workers for data loading

    Returns:
        train_loader, val_loader, class_names
    """
    data_dir = data_config['data_dir']
    specific_classes = data_config.get('specific_classes', None)
    print(f"[INFO] Loading dataset from: {data_dir}")

    # Create full dataset
    full_dataset = CassavaDataset(
        data_dir=data_dir,
        transform=None,  # Will be applied separately for train/val
        specific_classes=specific_classes
    )

    # Split into train and validation (80/20)
    train_size = int(0.8 * len(full_dataset))
    val_size = len(full_dataset) - train_size

    # Set seed for reproducibility
    torch.manual_seed(42)
    train_dataset, val_dataset = random_split(full_dataset, [train_size, val_size])

    # Apply transforms
    train_transform = get_transforms(augment=augment, is_training=True)
    val_transform = get_transforms(augment=None, is_training=False)

    # Update transforms for split datasets
    train_dataset.dataset.transform = train_transform
    val_dataset.dataset.transform = val_transform

    # Create data loaders
    train_loader = DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
        persistent_workers=True if num_workers > 0 else False
    )

    val_loader = DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        pin_memory=True,
        persistent_workers=True if num_workers > 0 else False
    )

    class_names = list(full_dataset.class_to_idx.keys())

    # Export metadata
    if user_id:
        export_training_metadata(train_dataset, val_dataset, full_dataset,
                                data_dir, user_id, augment)

    print(f"[INFO] Dataset loaded successfully")
    print(f"[INFO] Classes: {class_names}")
    print(f"[INFO] Training samples: {len(train_dataset):,}")
    print(f"[INFO] Validation samples: {len(val_dataset):,}")

    return train_loader, val_loader, class_names


def export_training_metadata(train_dataset, val_dataset, full_dataset,
                             data_dir, user_id, augment):
    """Export training metadata to CSV."""
    csv_export_dir = os.path.join(os.path.dirname(data_dir), "training_metadata")
    os.makedirs(csv_export_dir, exist_ok=True)

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    csv_filename = f"training_metadata_{user_id}_{timestamp}.csv"
```

```
        csv_path = os.path.join(csv_export_dir, csv_filename)

        metadata_records = []

        # Training set
        for idx in train_dataset.indices:
            img_path = full_dataset.image_paths[idx]
            label_idx = full_dataset.labels[idx]
            label_name = full_dataset.idx_to_class[label_idx]

            metadata_records.append({
                'filepath': img_path,
                'filename': os.path.basename(img_path),
                'label_index': label_idx,
                'label_name': label_name,
                'split': 'train',
                'user_id': user_id,
                'timestamp': timestamp
            })

        # Validation set
        for idx in val_dataset.indices:
            img_path = full_dataset.image_paths[idx]
            label_idx = full_dataset.labels[idx]
            label_name = full_dataset.idx_to_class[label_idx]

            metadata_records.append({
                'filepath': img_path,
                'filename': os.path.basename(img_path),
                'label_index': label_idx,
                'label_name': label_name,
                'split': 'validation',
                'user_id': user_id,
                'timestamp': timestamp
            })

        df = pd.DataFrame(metadata_records)
        df.to_csv(csv_path, index=False)

        # Save augmentation config
        if augment:
            aug_config_path = os.path.join(csv_export_dir,
                                           f"augmentation_config_{user_id}_{timestamp}.csv")
            aug_df = pd.DataFrame([augment])
            aug_df['user_id'] = user_id
            aug_df['timestamp'] = timestamp
            aug_df.to_csv(aug_config_path, index=False)

        print(f"[INFO] Training metadata exported to: {csv_path}")
```

```
Overwriting /content/drive/MyDrive/cassava/src/dataset.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/dashboard_plots.py
#!/usr/bin/env python3
"""
Dashboard Plotting Functions
Modular visualization functions for model comparison dashboard
"""

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Set style
sns.set_style("whitegrid")
plt.rcParams['figure.figsize'] = (16, 10)
plt.rcParams['font.size'] = 10


def plot_accuracy_comparison(dl_accs, cl_accs, save_path=None):
    """Plot accuracy comparison box plot."""
    fig, ax = plt.subplots(figsize=(10, 6))

    ax.boxplot([dl_accs, cl_accs], labels=['Deep Learning', 'Classical ML'])
    ax.set_title('Validation Accuracy Distribution', fontweight='bold', fontsize=14)
    ax.set_ylabel('Accuracy (%)', fontsize=12)
    ax.grid(True, alpha=0.3)

    # Add mean markers
    if dl_accs:
        ax.plot(1, np.mean(dl_accs), 'rx', markersize=12,
               label=f'DL Mean: {np.mean(dl_accs):.2f}%')
    if cl_accs:
        ax.plot(2, np.mean(cl_accs), 'bx', markersize=12,
               label=f'CL Mean: {np.mean(cl_accs):.2f}%')
    ax.legend(fontsize=11)

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()


def plot_bar_comparison(dl_accs, cl_accs, save_path=None):
    """Plot bar chart comparison."""
    fig, ax = plt.subplots(figsize=(10, 6))

    model_names = ['Deep Learning', 'Classical ML']
    mean_accs = [np.mean(dl_accs) if dl_accs else 0,
                 np.mean(cl_accs) if cl_accs else 0]
    std_accs = [np.std(dl_accs) if dl_accs else 0,
                np.std(cl_accs) if cl_accs else 0]

    x_pos = np.arange(len(model_names))
    ax.bar(x_pos, mean_accs, yerr=std_accs, capsize=5,
          color=['#3498db', '#e74c3c'], alpha=0.8, edgecolor='black', linewidth=1.5)
    ax.set_title('Average Validation Accuracy', fontweight='bold', fontsize=14)
    ax.set_ylabel('Accuracy (%)', fontsize=12)
    ax.set_xticks(x_pos)
    ax.set_xticklabels(model_names, fontsize=12)
    ax.grid(True, alpha=0.3, axis='y')

    # Add value labels
    for i, (acc, std) in enumerate(zip(mean_accs, std_accs)):
        ax.text(i, acc + std + 1, f'{acc:.2f}%\n±{std:.2f}',
               ha='center', va='bottom', fontweight='bold', fontsize=11)

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()


def plot_training_runs_count(dl_logs, cl_logs, save_path=None):
    """Plot number of training runs."""
    fig, ax = plt.subplots(figsize=(8, 6))

    model_names = ['Deep Learning', 'Classical ML']
    counts = [len(dl_logs), len(cl_logs)]
    x_pos = np.arange(len(model_names))

    ax.bar(x_pos, counts, color=['#3498db', '#e74c3c'], alpha=0.8,
          edgecolor='black', linewidth=1.5)
    ax.set_title('Number of Training Runs', fontweight='bold', fontsize=14)
    ax.set_ylabel('Count', fontsize=12)
    ax.set_xticks(x_pos)
    ax.set_xticklabels(model_names, fontsize=12)
    ax.grid(True, alpha=0.3, axis='y')

    # Add value labels
    for i, count in enumerate(counts):
        ax.text(i, count + 0.5, str(count), ha='center', va='bottom',
               fontsize=16, fontweight='bold')

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()


def plot_confusion_matrix(cm, class_names, title, cmap='Blues', save_path=None):
    """Plot a single confusion matrix."""
    fig, ax = plt.subplots(figsize=(10, 8))

    cm_norm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    sns.heatmap(cm_norm, annot=True, fmt='.2f', cmap=cmap,
               xticklabels=class_names, yticklabels=class_names, ax=ax,
               cbar_kws={'label': 'Accuracy'}, vmin=0, vmax=1,
               linewidths=0.5, linecolor='gray')

    ax.set_title(title, fontweight='bold', fontsize=14, pad=20)
    ax.set_ylabel('True Label', fontsize=12, fontweight='bold')
    ax.set_xlabel('Predicted Label', fontsize=12, fontweight='bold')

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()


def plot_training_curves(dl_logs, save_path=None):
    """Plot training curves for deep learning models."""
    if not dl_logs:
        print("[INFO] No deep learning logs with training history")
        return

    fig, axes = plt.subplots(1, 2, figsize=(16, 6))
    fig.suptitle('Deep Learning Training Curves', fontsize=16, fontweight='bold')

    colors = plt.cm.tab20(np.linspace(0, 1, len(dl_logs)))

    for idx, log in enumerate(dl_logs):
        if 'history' not in log:
            continue

        history = log['history']
        label = f"{log.get('user_id', 'unknown')[:10]}_{log.get('timestamp', '')[:8]}"

        # Validation accuracy
        if 'val_accuracy' in history:
            epochs = range(1, len(history['val_accuracy']) + 1)
            axes[0].plot(epochs, history['val_accuracy'],
                        label=label, color=colors[idx], alpha=0.7, linewidth=2)

        # Validation loss
        if 'val_loss' in history:
            epochs = range(1, len(history['val_loss']) + 1)
            axes[1].plot(epochs, history['val_loss'],
                        label=label, color=colors[idx], alpha=0.7, linewidth=2)

    axes[0].set_title('Validation Accuracy', fontweight='bold', fontsize=12)
    axes[0].set_xlabel('Epoch', fontsize=11)
    axes[0].set_ylabel('Accuracy (%)', fontsize=11)
    axes[0].legend(bbox_to_anchor=(1.05, 1), loc='upper left', fontsize=8)
    axes[0].grid(True, alpha=0.3)

    axes[1].set_title('Validation Loss', fontweight='bold', fontsize=12)
    axes[1].set_xlabel('Epoch', fontsize=11)
    axes[1].set_ylabel('Loss', fontsize=11)
    axes[1].legend(bbox_to_anchor=(1.05, 1), loc='upper left', fontsize=8)
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()


def plot_per_class_accuracy(dl_cm, cl_cm, class_names, save_path=None):
    """Plot per-class accuracy comparison."""
    if len(dl_cm) == 0 or len(cl_cm) == 0:
        return

    fig, ax = plt.subplots(figsize=(14, 6))

    dl_class_acc = [dl_cm[i, i] / dl_cm[i].sum() * 100 if dl_cm[i].sum() > 0 else 0
                   for i in range(len(class_names))]
    cl_class_acc = [cl_cm[i, i] / cl_cm[i].sum() * 100 if cl_cm[i].sum() > 0 else 0
                   for i in range(len(class_names))]

    x = np.arange(len(class_names))
    width = 0.35

    bars1 = ax.bar(x - width/2, dl_class_acc, width, label='Deep Learning',
                   color='#3498db', alpha=0.8, edgecolor='black')
    bars2 = ax.bar(x + width/2, cl_class_acc, width, label='Classical ML',
                   color='#e74c3c', alpha=0.8, edgecolor='black')

    ax.set_ylabel('Accuracy (%)', fontsize=12, fontweight='bold')
    ax.set_title('Per-Class Accuracy Comparison', fontsize=14, fontweight='bold')
    ax.set_xticks(x)
    ax.set_xticklabels(class_names, rotation=45, ha='right', fontsize=11)
    ax.legend(fontsize=12, loc='lower left')
    ax.grid(True, alpha=0.3, axis='y')
    ax.set_ylim([0, 105])

    # Add value labels
    for bars in [bars1, bars2]:
        for bar in bars:
            height = bar.get_height()
            ax.text(bar.get_x() + bar.get_width()/2., height + 1,
                   f'{height:.1f}',
                   ha='center', va='bottom', fontsize=9, fontweight='bold')

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()


def plot_class_distribution(cm, class_names, save_path=None):
    """Plot class distribution analysis."""
    if len(cm) == 0:
        return

    actual_counts = cm.sum(axis=1)
    predicted_counts = cm.sum(axis=0)
    correct_counts = np.diag(cm)
    fig, axes = plt.subplots(2, 2, figsize=(16, 12))
    fig.suptitle('Class Distribution Analysis', fontsize=18, fontweight='bold')

    # 1. Actual distribution bar chart
    ax1 = axes[0, 0]
    bars = ax1.bar(range(len(class_names)), actual_counts,
                   color='steelblue', alpha=0.8, edgecolor='black')
    ax1.set_title('Actual Class Distribution', fontweight='bold', fontsize=14)
    ax1.set_xlabel('Class', fontsize=12)
    ax1.set_ylabel('Samples', fontsize=12)
    ax1.set_xticks(range(len(class_names)))
    ax1.set_xticklabels(class_names, rotation=45, ha='right')
    ax1.grid(True, alpha=0.3, axis='y')

    for bar, count in zip(bars, actual_counts):
        height = bar.get_height()
        ax1.text(bar.get_x() + bar.get_width()/2., height,
                f'{int(count)}', ha='center', va='bottom',
                fontweight='bold', fontsize=11)

    # 2. Pie chart
    ax2 = axes[0, 1]
    colors = plt.cm.Set3(np.linspace(0, 1, len(class_names)))
    wedges, texts, autotexts = ax2.pie(actual_counts, labels=class_names,
                                        autopct='%1.1f%%', colors=colors,
                                        startangle=90, textprops={'fontsize': 11})
    ax2.set_title('Percentage Distribution', fontweight='bold', fontsize=14)
    for autotext in autotexts:
        autotext.set_color('white')
        autotext.set_fontweight('bold')

    # 3. Actual vs Predicted
    ax3 = axes[1, 0]
    x = np.arange(len(class_names))
    width = 0.35

    ax3.bar(x - width/2, actual_counts, width, label='Actual',
            color='steelblue', alpha=0.8, edgecolor='black')
    ax3.bar(x + width/2, predicted_counts, width, label='Predicted',
            color='coral', alpha=0.8, edgecolor='black')

    ax3.set_title('Actual vs Predicted', fontweight='bold', fontsize=14)
    ax3.set_xlabel('Class', fontsize=12)
    ax3.set_ylabel('Samples', fontsize=12)
    ax3.set_xticks(x)
    ax3.set_xticklabels(class_names, rotation=45, ha='right')
    ax3.legend(fontsize=11)
    ax3.grid(True, alpha=0.3, axis='y')

    # 4. Correct vs Incorrect
    ax4 = axes[1, 1]
    incorrect_counts = actual_counts - correct_counts

    ax4.bar(x - width/2, correct_counts, width, label='Correct',
            color='green', alpha=0.8, edgecolor='black')
    ax4.bar(x + width/2, incorrect_counts, width, label='Incorrect',
            color='red', alpha=0.8, edgecolor='black')

    ax4.set_title('Correct vs Incorrect', fontweight='bold', fontsize=14)
    ax4.set_xlabel('Class', fontsize=12)
    ax4.set_ylabel('Samples', fontsize=12)
    ax4.set_xticks(x)
    ax4.set_xticklabels(class_names, rotation=45, ha='right')
    ax4.legend(fontsize=11)
    ax4.grid(True, alpha=0.3, axis='y')

    # Add accuracy labels
    for i, (correct, total) in enumerate(zip(correct_counts, actual_counts)):
        acc = (correct / total * 100) if total > 0 else 0
        ax4.text(i, max(correct_counts[i], incorrect_counts[i]) + 2,
                f'{acc:.1f}%', ha='center', va='bottom',
                fontweight='bold', fontsize=10, color='darkblue')

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()
```

```
Overwriting /content/drive/MyDrive/cassava/src/dashboard_plots.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/dashboard_reports.py
#!/usr/bin/env python3
"""
Dashboard Reports
Generate text-based reports and summaries
"""

import numpy as np
from tabulate import tabulate
from colorama import Fore, Style, init

init(autoreset=True)


def generate_comparison_report(dl_logs, cl_logs):
    """Generate detailed comparison report."""
    print(f"\n{Fore.CYAN}{'='*70}")
    print(f"📊 MODEL COMPARISON REPORT")
    print(f"{'='*70}{Style.RESET_ALL}\n")

    # Extract accuracies
    dl_accs = [log.get('best_val_acc', log.get('val_acc', 0)) for log in dl_logs]
    cl_accs = [log.get('best_val_acc', log.get('val_acc', 0)) for log in cl_logs]

    # Summary table
    comparison_data = []

    if dl_accs:
        comparison_data.append([
            "Deep Learning",
            len(dl_logs),
            f"{np.mean(dl_accs):.2f}%",
            f"{np.std(dl_accs):.2f}%",
            f"{max(dl_accs):.2f}%",
            f"{min(dl_accs):.2f}%"
        ])

    if cl_accs:
        comparison_data.append([
            "Classical ML",
            len(cl_logs),
            f"{np.mean(cl_accs):.2f}%",
            f"{np.std(cl_accs):.2f}%",
            f"{max(cl_accs):.2f}%",
            f"{min(cl_accs):.2f}%"
        ])

    print(f"{Fore.GREEN}Model Performance Summary:{Style.RESET_ALL}\n")
    print(tabulate(comparison_data, headers=[
        "Model Type", "Runs", "Mean Acc", "Std Dev", "Best", "Worst"
    ], tablefmt="fancy_grid"))

    # Winner declaration
    if dl_accs and cl_accs:
        print(f"\n{Fore.YELLOW}🏆 Performance Winner:{Style.RESET_ALL}")
```

```
            diff = np.mean(dl_accs) - np.mean(cl_accs)
            if abs(diff) < 1:
                print(f"  TIE! (Difference: {abs(diff):.2f}%)")
            elif diff > 0:
                print(f"  Deep Learning wins by {diff:.2f}%!")
            else:
                print(f"  Classical ML wins by {abs(diff):.2f}%!")

        print()


    def display_best_models(best_dl, best_cl):
        """Display best model information."""
        if best_dl:
            print(f"\n{Fore.GREEN}{'='*70}")
            print(f"🏆 BEST DEEP LEARNING MODEL")
            print(f"{'='*70}{Style.RESET_ALL}")
            print(f"User ID: {best_dl.get('user_id', 'unknown')}")
            print(f"Timestamp: {best_dl.get('timestamp', 'unknown')}")
            print(f"Validation Accuracy: {best_dl.get('best_val_acc', 0):.2f}%")
            print(f"Epochs Trained: {best_dl.get('epochs_trained', 0)}")

        if best_cl:
            print(f"\n{Fore.GREEN}{'='*70}")
            print(f"🏆 BEST CLASSICAL ML MODEL")
            print(f"{'='*70}{Style.RESET_ALL}")
            print(f"User ID: {best_cl.get('user_id', 'unknown')}")
            print(f"Timestamp: {best_cl.get('timestamp', 'unknown')}")
            print(f"Validation Accuracy: {best_cl.get('best_val_acc', 0):.2f}%")
            print(f"N Estimators: {best_cl.get('n_estimators', 0)}")


    def display_overall_winner(best_dl, best_cl):
        """Display overall winner."""
        if best_dl and best_cl:
            print(f"\n{Fore.YELLOW}{'='*70}")
            print(f"🎯 OVERALL WINNER")
            print(f"{'='*70}{Style.RESET_ALL}")
            dl_acc = best_dl.get('best_val_acc', 0)
            cl_acc = best_cl.get('best_val_acc', 0)
            if dl_acc > cl_acc:
                print(f"🥇 Deep Learning: {dl_acc:.2f}% (beats Classical ML by {dl_acc - cl_acc:.2f}%)")
            elif cl_acc > dl_acc:
                print(f"🥇 Classical ML: {cl_acc:.2f}% (beats Deep Learning by {cl_acc - dl_acc:.2f}%)")
            else:
                print(f"🤝 TIE: Both models achieve {dl_acc:.2f}%")
            print()


    def print_visualization_summary(viz_dir, timestamp):
        """Print summary of generated visualizations."""
        print(f"\n{Fore.GREEN}✅ All visualizations saved to: {viz_dir}{Style.RESET_ALL}")
        print(f"\n{Fore.CYAN}Generated files:{Style.RESET_ALL}")
        print(f"  1️⃣  accuracy_comparison_{timestamp}.png")
        print(f"  2️⃣  bar_comparison_{timestamp}.png")
        print(f"  3️⃣  training_runs_{timestamp}.png")
        print(f"  4️⃣  class_distribution_{timestamp}.png")
        print(f"  5️⃣  per_class_accuracy_{timestamp}.png")
        print(f"  6️⃣  training_curves_{timestamp}.png")
        print(f"  7️⃣  dl_confusion_matrix_{timestamp}.png")
        print(f"  8️⃣  cl_confusion_matrix_{timestamp}.png")


    def generate_detailed_summary(best_dl, best_cl):
        """Generate detailed text summary."""
        summary = []
        summary.append("\n" + "="*70)
        summary.append("📝 DETAILED ANALYSIS")
        summary.append("="*70)

        if best_dl:
            summary.append("\n🔵 Deep Learning Model:")
            summary.append(f"  • Accuracy: {best_dl.get('best_val_acc', 0):.2f}%")
            summary.append(f"  • Framework: PyTorch")
            summary.append(f"  • Architecture: EfficientNet")
            summary.append(f"  • Epochs: {best_dl.get('epochs_trained', 0)}")
            summary.append(f"  • Training completed: {best_dl.get('timestamp', 'N/A')}")

            if 'class_names' in best_dl:
                summary.append(f"  • Classes: {', '.join(best_dl['class_names'])}")

        if best_cl:
            summary.append("\n🔴 Classical ML Model:")
            summary.append(f"  • Accuracy: {best_cl.get('best_val_acc', 0):.2f}%")
            summary.append(f"  • Framework: Scikit-learn")
            summary.append(f"  • Algorithm: Random Forest")
            summary.append(f"  • Estimators: {best_cl.get('n_estimators', 0)}")
            summary.append(f"  • Training completed: {best_cl.get('timestamp', 'N/A')}")

            if 'class_names' in best_cl:
                summary.append(f"  • Classes: {', '.join(best_cl['class_names'])}")

        if best_dl and best_cl:
            summary.append("\n📊 Comparison Insights:")
            dl_acc = best_dl.get('best_val_acc', 0)
            cl_acc = best_cl.get('best_val_acc', 0)
            diff = abs(dl_acc - cl_acc)

            if diff < 2:
                summary.append("  • Models perform very similarly (difference < 2%)")
            elif diff < 5:
                summary.append("  • Moderate performance difference (2-5%)")
            else:
                summary.append("  • Significant performance difference (> 5%)")

            summary.append(f"  • Accuracy gap: {diff:.2f}%")

        summary.append("\n" + "="*70)

        return "\n".join(summary)


    def print_training_statistics(dl_logs, cl_logs):
        """Print training statistics."""
        print(f"\n{Fore.CYAN}{'='*70}")
        print(f"📈 TRAINING STATISTICS")
        print(f"{'='*70}{Style.RESET_ALL}\n")

        if dl_logs:
            dl_accs = [log.get('best_val_acc', 0) for log in dl_logs]
            dl_epochs = [log.get('epochs_trained', 0) for log in dl_logs]

            print(f"{Fore.BLUE}Deep Learning:{Style.RESET_ALL}")
            print(f"  Total runs: {len(dl_logs)}")
            print(f"  Avg accuracy: {np.mean(dl_accs):.2f}% (±{np.std(dl_accs):.2f}%)")
            print(f"  Best accuracy: {max(dl_accs):.2f}%")
            print(f"  Worst accuracy: {min(dl_accs):.2f}%")
            print(f"  Avg epochs: {np.mean(dl_epochs):.1f}")
            print()

        if cl_logs:
            cl_accs = [log.get('best_val_acc', 0) for log in cl_logs]
            cl_estimators = [log.get('n_estimators', 0) for log in cl_logs]

            print(f"{Fore.RED}Classical ML:{Style.RESET_ALL}")
            print(f"  Total runs: {len(cl_logs)}")
            print(f"  Avg accuracy: {np.mean(cl_accs):.2f}% (±{np.std(cl_accs):.2f}%)")
            print(f"  Best accuracy: {max(cl_accs):.2f}%")
            print(f"  Worst accuracy: {min(cl_accs):.2f}%")
            print(f"  Avg estimators: {np.mean(cl_estimators):.0f}")
            print()
```

```
Overwriting /content/drive/MyDrive/cassava/src/dashboard_reports.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/dashboard_main.py
#!/usr/bin/env python3
"""
Dashboard Main - Simplified and Modular
Orchestrates the comparison dashboard using modular components
"""

import os
import json
import glob
import numpy as np
from datetime import datetime
from colorama import Fore, Style

# Import modular components
from dashboard_plots import (
    plot_accuracy_comparison, plot_bar_comparison, plot_training_runs_count,
    plot_confusion_matrix, plot_training_curves, plot_per_class_accuracy,
    plot_class_distribution
)
from dashboard_reports import (
    generate_comparison_report, display_best_models, display_overall_winner,
    print_visualization_summary, generate_detailed_summary, print_training_statistics
)
from best_models_tracker import BestModelsTracker


def load_all_logs(log_dir="outputs/logs"):
    """Load all training logs from JSON files."""
    if not os.path.exists(log_dir):
        return []

    logs = []
    for root, dirs, files in os.walk(log_dir):
        for f in files:
            if f.endswith(".json"):
                path = os.path.join(root, f)
                try:
                    with open(path, "r") as jf:
                        data = json.load(jf)
                        data['file'] = f
                        data['path'] = path
                        # Determine model type
                        if 'deep_learning' in path or data.get('framework') == 'pytorch':
                            data['model_category'] = 'deep_learning'
                        elif 'classical' in path or data.get('framework') == 'classical_ml':
                            data['model_category'] = 'classical'
                        logs.append(data)
                except Exception as e:
                    print(f"[WARNING] Error loading {f}: {e}")

    return logs


def separate_logs_by_type(logs):
    """Separate logs into deep learning and classical ML."""
    dl_logs = [log for log in logs if log.get('model_category') == 'deep_learning']
    cl_logs = [log for log in logs if log.get('model_category') == 'classical']
    return dl_logs, cl_logs
```

```
def show_comparison_dashboard(log_dir="outputs/logs", save_plots=True, work_dir=None):
    """Display comprehensive comparison dashboard."""

    print(f"\n{Fore.CYAN}{'='*70}")
    print(f"🌿 CASSAVA DISEASE CLASSIFICATION - COMPARISON DASHBOARD")
    print(f"{'='*70}{Style.RESET_ALL}\n")

    # Load logs
    print("[INFO] Loading training logs...")
    logs = load_all_logs(log_dir)

    if not logs:
        print(Fore.RED + "❌ No logs found. Train models first!" + Style.RESET_ALL)
        return

    # Separate by type
    dl_logs, cl_logs = separate_logs_by_type(logs)

    print(f"[INFO] Found {len(dl_logs)} deep learning run(s)")
    print(f"[INFO] Found {len(cl_logs)} classical ML run(s)\n")

    # Get best models
    best_dl = max(dl_logs, key=lambda x: x.get('best_val_acc', 0)) if dl_logs else None
    best_cl = max(cl_logs, key=lambda x: x.get('best_val_acc', 0)) if cl_logs else None

    # Generate text reports
    generate_comparison_report(dl_logs, cl_logs)
    display_best_models(best_dl, best_cl)
    display_overall_winner(best_dl, best_cl)
    print_training_statistics(dl_logs, cl_logs)

    # Print detailed summary
    print(generate_detailed_summary(best_dl, best_cl))

    # Update best models tracker
    if work_dir:
        print(f"\n{Fore.CYAN}[INFO] Updating best models tracker...{Style.RESET_ALL}")
        tracker = BestModelsTracker(work_dir)
        tracker.update_best_models()

    # Extract accuracies for plotting
    dl_accs = [log.get('best_val_acc', log.get('val_acc', 0)) for log in dl_logs]
    cl_accs = [log.get('best_val_acc', log.get('val_acc', 0)) for log in cl_logs]

    # Generate visualizations
    if save_plots and work_dir:
        viz_dir = os.path.join(work_dir, "outputs", "visualizations")
        os.makedirs(viz_dir, exist_ok=True)
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        print(f"\n{Fore.CYAN}[INFO] Generating visualizations...{Style.RESET_ALL}\n")

        # 1. Accuracy comparison
        if dl_accs or cl_accs:
            print("[1/8] Accuracy comparison...")
            plot_accuracy_comparison(
                dl_accs, cl_accs,
                save_path=os.path.join(viz_dir, f"accuracy_comparison_{timestamp}.png")
            )

        # 2. Bar comparison
        if dl_accs or cl_accs:
            print("[2/8] Bar comparison...")
            plot_bar_comparison(
                dl_accs, cl_accs,
                save_path=os.path.join(viz_dir, f"bar_comparison_{timestamp}.png")
            )

        # 3. Training runs count
        print("[3/8] Training runs count...")
        plot_training_runs_count(
            dl_logs, cl_logs,
            save_path=os.path.join(viz_dir, f"training_runs_{timestamp}.png")
        )

        # 4. Class distribution
        if best_dl and 'confusion_matrix' in best_dl:
            print("[4/8] Class distribution...")
            cm = np.array(best_dl['confusion_matrix'])
            class_names = best_dl.get('class_names', [])
            plot_class_distribution(
                cm, class_names,
                save_path=os.path.join(viz_dir, f"class_distribution_{timestamp}.png")
            )

        # 5. Per-class accuracy comparison
        if (best_dl and 'confusion_matrix' in best_dl and
            best_cl and 'confusion_matrix' in best_cl):
            print("[5/8] Per-class accuracy...")
            dl_cm = np.array(best_dl['confusion_matrix'])
            cl_cm = np.array(best_cl['confusion_matrix'])
            class_names = best_dl.get('class_names', [])
            plot_per_class_accuracy(
                dl_cm, cl_cm, class_names,
                save_path=os.path.join(viz_dir, f"per_class_accuracy_{timestamp}.png")
            )

        # 6. Training curves
        if dl_logs:
            print("[6/8] Training curves...")
            plot_training_curves(
                dl_logs,
                save_path=os.path.join(viz_dir, f"training_curves_{timestamp}.png")
            )

        # 7. Deep Learning confusion matrix
        if best_dl and 'confusion_matrix' in best_dl:
            print("[7/8] DL confusion matrix...")
            cm = np.array(best_dl['confusion_matrix'])
            class_names = best_dl.get('class_names', [])
            acc = best_dl.get('best_val_acc', 0)
            title = f"Deep Learning Confusion Matrix\nAccuracy: {acc:.2f}%"
            plot_confusion_matrix(
                cm, class_names, title, cmap='Blues',
                save_path=os.path.join(viz_dir, f"dl_confusion_matrix_{timestamp}.png")
            )

        # 8. Classical ML confusion matrix
        if best_cl and 'confusion_matrix' in best_cl:
            print("[8/8] Classical confusion matrix...")
            cm = np.array(best_cl['confusion_matrix'])
            class_names = best_cl.get('class_names', [])
            acc = best_cl.get('best_val_acc', 0)
            title = f"Classical ML Confusion Matrix\nAccuracy: {acc:.2f}%"
            plot_confusion_matrix(
                cm, class_names, title, cmap='Reds',
                save_path=os.path.join(viz_dir, f"cl_confusion_matrix_{timestamp}.png")
            )

        print_visualization_summary(viz_dir, timestamp)
    else:
        # Show plots without saving
        print(f"\n{Fore.CYAN}[INFO] Displaying visualizations...{Style.RESET_ALL}\n")
        if dl_accs or cl_accs:
            plot_accuracy_comparison(dl_accs, cl_accs)
            plot_bar_comparison(dl_accs, cl_accs)

        plot_training_runs_count(dl_logs, cl_logs)

        if best_dl and 'confusion_matrix' in best_dl:
            cm = np.array(best_dl['confusion_matrix'])
            class_names = best_dl.get('class_names', [])
            plot_class_distribution(cm, class_names)

        if (best_dl and 'confusion_matrix' in best_dl and
            best_cl and 'confusion_matrix' in best_cl):
            dl_cm = np.array(best_dl['confusion_matrix'])
            cl_cm = np.array(best_cl['confusion_matrix'])
            class_names = best_dl.get('class_names', [])
            plot_per_class_accuracy(dl_cm, cl_cm, class_names)

        if dl_logs:
            plot_training_curves(dl_logs)

        if best_dl and 'confusion_matrix' in best_dl:
            cm = np.array(best_dl['confusion_matrix'])
            class_names = best_dl.get('class_names', [])
            acc = best_dl.get('best_val_acc', 0)
            title = f"Deep Learning Confusion Matrix\nAccuracy: {acc:.2f}%"
            plot_confusion_matrix(cm, class_names, title, cmap='Blues')

        if best_cl and 'confusion_matrix' in best_cl:
            cm = np.array(best_cl['confusion_matrix'])
            class_names = best_cl.get('class_names', [])
            acc = best_cl.get('best_val_acc', 0)
            title = f"Classical ML Confusion Matrix\nAccuracy: {acc:.2f}%"
            plot_confusion_matrix(cm, class_names, title, cmap='Reds')

    print(f"\n{Fore.CYAN}{'='*70}")
    print(f"✅ Comparison Dashboard Complete!")
    print(f"{'='*70}{Style.RESET_ALL}\n")


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Model Comparison Dashboard")
    parser.add_argument("--log_dir", type=str, default="outputs/logs")
    parser.add_argument("--work_dir", type=str, default="/content/drive/MyDrive/cassava")
    parser.add_argument("--no_save", action="store_true")

    args = parser.parse_args()

    show_comparison_dashboard(
        log_dir=args.log_dir,
        save_plots=not args.no_save,
        work_dir=args.work_dir
    )
```

```
Overwriting /content/drive/MyDrive/cassava/src/dashboard_main.py
```

```
%%writefile /content/drive/MyDrive/cassava/best_models_tracker.py
#!/usr/bin/env python3
"""
Best Models Tracker
Manages the tracking and storage of best model paths for easy testing
"""

import os
import json
import glob
from datetime import datetime
from typing import Dict, Optional, List


class BestModelsTracker:
    """Track and manage best model paths."""

    def __init__(self, work_dir: str):
        self.work_dir = work_dir
        self.tracker_file = os.path.join(work_dir, "outputs", "best_models.json")
        self.models_dir = os.path.join(work_dir, "outputs", "models")
        self.logs_dir = os.path.join(work_dir, "outputs", "logs")

    def load_tracker(self) -> Dict:
        """Load existing tracker data."""
        if os.path.exists(self.tracker_file):
            try:
                with open(self.tracker_file, 'r') as f:
                    return json.load(f)
            except Exception as e:
                print(f"[WARNING] Failed to load tracker: {e}")

        return {
            "last_updated": None,
            "models": {
                "deep_learning": {
                    "best_model_path": None,
                    "best_accuracy": 0.0,
                    "user_id": None,
                    "timestamp": None,
                    "epochs": 0
                },
                "classical": {
                    "best_model_path": None,
                    "best_accuracy": 0.0,
                    "user_id": None,
                    "timestamp": None,
                    "n_estimators": 0
                }
            },
            "history": []
        }

    def save_tracker(self, data: Dict):
        """Save tracker data to JSON."""
        data["last_updated"] = datetime.now().isoformat()
        os.makedirs(os.path.dirname(self.tracker_file), exist_ok=True)

        with open(self.tracker_file, 'w') as f:
            json.dump(data, f, indent=2)

        print(f"[INFO] Best models tracker updated: {self.tracker_file}")

    def find_best_model_from_logs(self, model_type: str) -> Optional[Dict]:
        """Find best model by analyzing all log files."""
        if not os.path.exists(self.logs_dir):
            return None

        # Determine framework identifier
        framework = "pytorch" if model_type == "deep_learning" else "classical_ml"
        keyword = "cassava_model" if model_type == "deep_learning" else "random_forest"
        ext = ".pth" if model_type == "deep_learning" else ".pkl"

        best_model_info = None
        best_acc = -1

        # Search all log files
        log_pattern = os.path.join(self.logs_dir, "**", "*.json")
        log_files = glob.glob(log_pattern, recursive=True)

        for log_file in log_files:
            try:
                with open(log_file, 'r') as f:
                    log_data = json.load(f)

                # Check if correct framework
                if log_data.get('framework') != framework:
                    continue

                # Get accuracy
                val_acc = log_data.get('best_val_acc', log_data.get('val_acc', 0))

                if val_acc > best_acc:
                    # Try to find corresponding model file
                    log_timestamp = log_data.get('timestamp', '')
                    log_user_id = log_data.get('user_id', '')

                    # Search for model file
                    model_pattern = os.path.join(self.models_dir, "**", f"*{keyword}*{ext}")
                    model_files = glob.glob(model_pattern, recursive=True)

                    # Filter by timestamp or user_id
                    matching_models = [
                        mf for mf in model_files
                        if (log_timestamp in mf or log_user_id in mf)
                        and "checkpoint" not in mf
                    ]

                    if matching_models:
                        best_acc = val_acc
                        best_model_info = {
                            'model_path': matching_models[0],
                            'accuracy': val_acc,
                            'user_id': log_user_id,
                            'timestamp': log_timestamp,
                            'log_path': log_file,
                            'metadata': {
                                'epochs': log_data.get('epochs_trained', 0),
                                'n_estimators': log_data.get('n_estimators', 0),
                                'class_names': log_data.get('class_names', []),
                                'framework': framework
                            }
                        }
            except Exception as e:
                continue

        return best_model_info

    def update_best_models(self):
        """Update tracker with current best models."""
        tracker_data = self.load_tracker()
        updated = False

        print("\n[INFO] Scanning for best models...")

        # Find best deep learning model
        dl_info = self.find_best_model_from_logs("deep_learning")
        if dl_info:
            current_best = tracker_data["models"]["deep_learning"]["best_accuracy"]
            if dl_info['accuracy'] > current_best:
                print(f"[INFO] New best Deep Learning model found: {dl_info['accuracy']:.2f}%")
                tracker_data["models"]["deep_learning"] = {
                    "best_model_path": dl_info['model_path'],
                    "best_accuracy": dl_info['accuracy'],
                    "user_id": dl_info['user_id'],
                    "timestamp": dl_info['timestamp'],
                    "epochs": dl_info['metadata']['epochs'],
                    "class_names": dl_info['metadata']['class_names']
                }
                updated = True

        # Find best classical model
        cl_info = self.find_best_model_from_logs("classical")
        if cl_info:
            current_best = tracker_data["models"]["classical"]["best_accuracy"]
            if cl_info['accuracy'] > current_best:
                print(f"[INFO] New best Classical ML model found: {cl_info['accuracy']:.2f}%")
                tracker_data["models"]["classical"] = {
                    "best_model_path": cl_info['model_path'],
                    "best_accuracy": cl_info['accuracy'],
                    "user_id": cl_info['user_id'],
                    "timestamp": cl_info['timestamp'],
                    "n_estimators": cl_info['metadata']['n_estimators'],
                    "class_names": cl_info['metadata']['class_names']
                }
                updated = True

        if updated:
            # Add to history
            history_entry = {
                "timestamp": datetime.now().isoformat(),
                "deep_learning_acc": tracker_data["models"]["deep_learning"]["best_accuracy"],
                "classical_acc": tracker_data["models"]["classical"]["best_accuracy"]
            }
            tracker_data["history"].append(history_entry)

            # Keep only last 50 history entries
            tracker_data["history"] = tracker_data["history"][-50:]

            self.save_tracker(tracker_data)
        else:
            print("[INFO] No new best models found.")

        return tracker_data

    def get_best_model_path(self, model_type: str) -> Optional[str]:
        """Get path to best model of specified type."""
        tracker_data = self.load_tracker()
        model_path = tracker_data["models"][model_type]["best_model_path"]

        if model_path and os.path.exists(model_path):
            return model_path

        return None

    def display_summary(self):
        """Display summary of best models."""
        tracker_data = self.load_tracker()

        print("\n" + "="*70)
        print("🏆 BEST MODELS SUMMARY")
        print("="*70)

        # Deep Learning
        dl_data = tracker_data["models"]["deep_learning"]
        if dl_data["best_model_path"]:
            print("\n📘 Deep Learning:")
            print(f"  Path:      {dl_data['best_model_path']}")
            print(f"  Accuracy:  {dl_data['best_accuracy']:.2f}%")
            print(f"  User:      {dl_data['user_id']}")
            print(f"  Timestamp: {dl_data['timestamp']}")
            print(f"  Epochs:    {dl_data['epochs']}")
        else:
            print("\n📘 Deep Learning: No model found")

        # Classical ML
        cl_data = tracker_data["models"]["classical"]
        if cl_data["best_model_path"]:
            print("\n📕 Classical ML:")
            print(f"  Path:         {cl_data['best_model_path']}")
            print(f"  Accuracy:     {cl_data['best_accuracy']:.2f}%")
            print(f"  User:         {cl_data['user_id']}")
            print(f"  Timestamp:    {cl_data['timestamp']}")
            print(f"  Estimators:   {cl_data['n_estimators']}")
        else:
            print("\n📕 Classical ML: No model found")

        # Winner
        if dl_data["best_model_path"] and cl_data["best_model_path"]:
            print("\n🎯 Overall Winner:")
            if dl_data["best_accuracy"] > cl_data["best_accuracy"]:
                diff = dl_data["best_accuracy"] - cl_data["best_accuracy"]
                print(f"  🥇 Deep Learning ({dl_data['best_accuracy']:.2f}%)")
                print(f"     Beats Classical ML by {diff:.2f}%")
            elif cl_data["best_accuracy"] > dl_data["best_accuracy"]:
                diff = cl_data["best_accuracy"] - dl_data["best_accuracy"]
                print(f"  🥇 Classical ML ({cl_data['best_accuracy']:.2f}%)")
                print(f"     Beats Deep Learning by {diff:.2f}%")
            else:
                print(f"  🤝 TIE: Both achieve {dl_data['best_accuracy']:.2f}%")

        print("\n" + "="*70 + "\n")


def update_best_models_tracker(work_dir: str):
    """Convenience function to update best models tracker."""
    tracker = BestModelsTracker(work_dir)
    tracker_data = tracker.update_best_models()
    tracker.display_summary()
    return tracker_data


def get_best_model_path(work_dir: str, model_type: str) -> Optional[str]:
    """Convenience function to get best model path."""
    tracker = BestModelsTracker(work_dir)
    return tracker.get_best_model_path(model_type)


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Best Models Tracker")
    parser.add_argument("--work_dir", type=str,
                       default="/content/drive/MyDrive/cassava",
                       help="Working directory")
    parser.add_argument("--update", action="store_true",
                       help="Update best models tracker")
    parser.add_argument("--show", action="store_true",
                       help="Show best models summary")

    args = parser.parse_args()

    tracker = BestModelsTracker(args.work_dir)

    if args.update:
        tracker.update_best_models()

    if args.show or not args.update:
        tracker.display_summary()
```

```
Overwriting /content/drive/MyDrive/cassava/best_models_tracker.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/model.py
import torch
import torch.nn as nn
import torchvision.models as models
from torchvision.models import (
    EfficientNet_B0_Weights,
    EfficientNet_B3_Weights,
    EfficientNet_B4_Weights,
    EfficientNet_V2_S_Weights,
    EfficientNet_V2_M_Weights
)


class CassavaModel(nn.Module):
    """
    Advanced EfficientNet-based classifier for Cassava disease detection.

    Supports multiple architectures:
    - efficientnet_b0: Fast, 5.3M params (4GB GPU)
    - efficientnet_b3: Balanced, 12M params (6GB GPU)
    - efficientnet_b4: High accuracy, 19M params (8GB GPU)
    - efficientnet_v2_s: Modern, faster, 21M params (8GB GPU)
    - efficientnet_v2_m: Best accuracy, 54M params (12GB GPU)
    """
    def __init__(self, num_classes=5, model_type="efficientnet_b3", dropout=0.4):
        super(CassavaModel, self).__init__()

        self.model_type = model_type

        # Load pretrained EfficientNet with proper architecture
        if model_type == "efficientnet_b4":
            self.backbone = models.efficientnet_b4(weights=EfficientNet_B4_Weights.IMAGENET1K_V1)
            feature_dim = 1792
        elif model_type == "efficientnet_v2_s":
            self.backbone = models.efficientnet_v2_s(weights=EfficientNet_V2_S_Weights.IMAGENET1K_V1)
            feature_dim = 1280
        elif model_type == "efficientnet_v2_m":
            self.backbone = models.efficientnet_v2_m(weights=EfficientNet_V2_M_Weights.IMAGENET1K_V1)
            feature_dim = 1280
        elif model_type == "efficientnet_b3":
            self.backbone = models.efficientnet_b3(weights=EfficientNet_B3_Weights.IMAGENET1K_V1)
            feature_dim = 1536
        else:  # efficientnet_b0
            self.backbone = models.efficientnet_b0(weights=EfficientNet_B0_Weights.IMAGENET1K_V1)
            feature_dim = 1280

        # Remove the original classifier
        self.backbone.classifier = nn.Identity()

        # Freeze early layers initially (will be unfrozen progressively)
        self._freeze_layers(freeze_ratio=0.7)

        # Advanced classifier head with deeper architecture
        self.classifier = nn.Sequential(
            # First block
            nn.BatchNorm1d(feature_dim),
            nn.Dropout(dropout),
            nn.Linear(feature_dim, 1024),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(1024),

            # Second block
            nn.Dropout(dropout * 0.75),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(512),

            # Third block
            nn.Dropout(dropout * 0.5),
            nn.Linear(512, 256),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(256),

            # Fourth block (optional, can be removed for faster training)
            nn.Dropout(dropout * 0.25),
            nn.Linear(256, 128),
            nn.ReLU(inplace=True),

            # Output layer
            nn.Linear(128, num_classes)
        )

        # Initialize classifier weights
        self._initialize_weights()

    def _freeze_layers(self, freeze_ratio=0.7):
        """Freeze early layers for transfer learning."""
        total_params = sum(1 for _ in self.backbone.parameters())
        freeze_count = int(total_params * freeze_ratio)

        for idx, param in enumerate(self.backbone.parameters()):
            if idx < freeze_count:
                param.requires_grad = False

    def _initialize_weights(self):
        """Initialize classifier weights with Xavier initialization."""
        for m in self.classifier.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm1d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        features = self.backbone(x)
        output = self.classifier(features)
        return output

    def unfreeze_all(self):
        """Unfreeze all layers for fine-tuning."""
        for param in self.parameters():
            param.requires_grad = True

    def get_trainable_params(self):
        """Get number of trainable parameters."""
        return sum(p.numel() for p in self.parameters() if p.requires_grad)


class AttentionClassifier(nn.Module):
    """
    Advanced classifier with attention mechanism.
    Use this for maximum accuracy (experimental).
    """
    def __init__(self, feature_dim, num_classes, dropout=0.4):
        super().__init__()

        # Self-attention layer
        self.attention = nn.Sequential(
            nn.Linear(feature_dim, feature_dim // 4),
            nn.Tanh(),
            nn.Linear(feature_dim // 4, 1)
        )

        # Classifier
        self.classifier = nn.Sequential(
            nn.BatchNorm1d(feature_dim),
            nn.Dropout(dropout),
            nn.Linear(feature_dim, 512),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(512),
            nn.Dropout(dropout * 0.5),
            nn.Linear(512, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        # Apply attention
        attention_weights = torch.softmax(self.attention(x), dim=1)
        x = x * attention_weights

        # Classify
        return self.classifier(x)


def build_model(num_classes=5, model_type="efficientnet_b3", device='cuda', use_attention=False):
    """
    Build and return the model optimized for 15GB GPU.

    Args:
        num_classes: Number of output classes
        model_type: Model architecture choice
        device: 'cuda' or 'cpu'
        use_attention: Use attention-based classifier (experimental)

    Returns:
        model: PyTorch model
    """
    # Determine best model for available memory
    if device == 'cuda':
        gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1024**3

        if gpu_memory >= 14:
            # 15GB GPU - can use best models
            recommended = "efficientnet_v2_m"
            print(f"[INFO] 🚀 Detected {gpu_memory:.1f}GB GPU - Using best model!")
            if model_type == "efficientnet_b3":  # Override default
                model_type = recommended
                print(f"[INFO] Auto-upgraded to {model_type} for maximum accuracy")
        elif gpu_memory >= 10:
            # 10-14GB GPU
            recommended = "efficientnet_b4"
            if model_type == "efficientnet_b0":
                model_type = recommended
                print(f"[INFO] Auto-upgraded to {model_type}")
        elif gpu_memory >= 6:
            # 6-10GB GPU
            recommended = "efficientnet_b3"
        else:
            # <6GB GPU
            recommended = "efficientnet_b0"
            if model_type not in ["efficientnet_b0", "efficientnet_b3"]:
                model_type = recommended
                print(f"[INFO] Limited GPU memory, using {model_type}")

    model = CassavaModel(num_classes=num_classes, model_type=model_type, dropout=0.4)
    model = model.to(device)

    # Count parameters
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

    print(f"[INFO] Model Architecture: {model_type}")
    print(f"[INFO] Total parameters: {total_params:,}")
    print(f"[INFO] Trainable parameters: {trainable_params:,}")
    print(f"[INFO] Frozen parameters: {total_params - trainable_params:,}")
    print(f"[INFO] Model size: ~{total_params * 4 / 1024**2:.1f} MB")

    # Memory estimation
    if device == 'cuda':
        estimated_memory = (total_params * 4 + trainable_params * 8) / 1024**3
        print(f"[INFO] Estimated GPU memory: ~{estimated_memory:.2f} GB")

    return model


def load_pretrained_model(model_path, device='cuda'):
    """
    Load a pretrained model from checkpoint.

    Args:
        model_path: Path to .pth file
        device: 'cuda' or 'cpu'

    Returns:
        model: Loaded model
        metadata: Model metadata (class_names, etc.)
    """
    checkpoint = torch.load(model_path, map_location=device)

    # Extract metadata
    class_names = checkpoint.get('class_names', [])
    num_classes = checkpoint.get('num_classes', len(class_names))
    model_type = checkpoint.get('model_type', 'efficientnet_b0')

    # Build model
    model = CassavaModel(num_classes=num_classes, model_type=model_type)
    model.load_state_dict(checkpoint['model_state_dict'])
    model = model.to(device)
    model.eval()

    metadata = {
        'class_names': class_names,
        'num_classes': num_classes,
        'model_type': model_type,
        'best_val_acc': checkpoint.get('best_val_acc', 0),
        'history': checkpoint.get('history', {})
    }

    print(f"[INFO] Loaded model: {model_type}")
    print(f"[INFO] Classes: {class_names}")
    print(f"[INFO] Best validation accuracy: {metadata['best_val_acc']:.2f}%")

    return model, metadata


# Model comparison for reference
MODEL_SPECS = {
    'efficientnet_b0': {
        'params': '5.3M',
        'gpu_memory': '4GB',
        'accuracy': '★★★☆☆',
        'speed': '★★★★★',
        'use_case': 'Fast prototyping, CPU training'
    },
    'efficientnet_b3': {
        'params': '12M',
        'gpu_memory': '6GB',
        'accuracy': '★★★★☆',
        'speed': '★★★★☆',
        'use_case': 'Balanced performance'
    },
    'efficientnet_b4': {
        'params': '19M',
        'gpu_memory': '8GB',
        'accuracy': '★★★★☆',
        'speed': '★★★☆☆',
        'use_case': 'High accuracy'
    },
    'efficientnet_v2_s': {
        'params': '21M',
        'gpu_memory': '8GB',
        'accuracy': '★★★★☆',
        'speed': '★★★★☆',
        'use_case': 'Modern, faster training'
    },
    'efficientnet_v2_m': {
        'params': '54M',
        'gpu_memory': '12GB',
        'accuracy': '★★★★★',
        'speed': '★★★☆☆',
        'use_case': 'Maximum accuracy (15GB GPU)'
    }
}


def print_model_specs():
    """Print model specifications table."""
    print("\n" + "="*80)
    print("MODEL SPECIFICATIONS")
    print("="*80)
    print(f"{'Model':<20} {'Params':<10} {'GPU':<8} {'Accuracy':<12} {'Speed':<12} {'Use Case':<25}")
    print("-"*80)
    for model, specs in MODEL_SPECS.items():
        print(f"{model:<20} {specs['params']:<10} {specs['gpu_memory']:<8} {specs['accuracy']:<12} {specs['spe
    print("="*80 + "\n")


if __name__ == "__main__":
    print_model_specs()

    # Test model creation
    print("Testing model creation...")
    if torch.cuda.is_available():
        device = 'cuda'
        print(f"GPU: {torch.cuda.get_device_name(0)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(0).total_memory / 1024**3:.1f} GB")
    else:
        device = 'cpu'

    model = build_model(num_classes=5, model_type="efficientnet_v2_m", device=device)
    print("\n✓ Model created successfully!")
```

```
Overwriting /content/drive/MyDrive/cassava/src/model.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/evaluate.py
import os
import json
import csv
import torch
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
from datetime import datetime
from tqdm import tqdm


def plot_history(history):
    """Plot training history."""
    plt.figure(figsize=(14, 5))

    epochs = range(1, len(history['train_acc']) + 1)

    # Accuracy plot
    plt.subplot(1, 3, 1)
    plt.plot(epochs, history['train_acc'], 'b-', label='Train Acc', linewidth=2)
    plt.plot(epochs, history['val_acc'], 'r-', label='Val Acc', linewidth=2)
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Accuracy (%)', fontsize=12)
    plt.title('Model Accuracy', fontsize=14, fontweight='bold')
    plt.legend(fontsize=11)
    plt.grid(True, alpha=0.3)

    # Loss plot
    plt.subplot(1, 3, 2)
    plt.plot(epochs, history['train_loss'], 'b-', label='Train Loss', linewidth=2)
    plt.plot(epochs, history['val_loss'], 'r-', label='Val Loss', linewidth=2)
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Loss', fontsize=12)
    plt.title('Model Loss', fontsize=14, fontweight='bold')
    plt.legend(fontsize=11)
    plt.grid(True, alpha=0.3)

    # Learning rate plot
    plt.subplot(1, 3, 3)
    plt.plot(epochs, history['lr'], 'g-', linewidth=2)
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Learning Rate', fontsize=12)
    plt.title('Learning Rate Schedule', fontsize=14, fontweight='bold')
    plt.yscale('log')
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()


def evaluate_model(model, val_loader, class_names, history=None,
                   augmentations=None, log_dir=None, user_id=None,
                   device='cuda', model_type="deep_learning"):
    """
    Evaluate PyTorch model and save logs.

    Args:
        model: Trained PyTorch model
        val_loader: Validation data loader
        class_names: List of class names
        history: Training history dictionary
        augmentations: List of augmentation configurations
        log_dir: Directory to save logs
        user_id: User identifier
        device: 'cuda' or 'cpu'
        model_type: Type of model (for logging)
    """
    device = torch.device(device if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    model.eval()

    print("[INFO] Evaluating model on validation set...")

    all_preds = []
    all_labels = []
    # Generate predictions
    with torch.no_grad():
        for images, labels in tqdm(val_loader, desc="Evaluating"):
            images = images.to(device)
            outputs = model(images)
            _, predicted = outputs.max(1)

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.numpy())

    y_pred = np.array(all_preds)
    y_true = np.array(all_labels)

    # Classification report
    print("\n" + "="*60)
    print("CLASSIFICATION REPORT")
    print("="*60)
    print(classification_report(y_true, y_pred, target_names=class_names))

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    print("\n" + "="*60)
    print("CONFUSION MATRIX")
    print("="*60)
    print(cm)
    print()

    # Per-class accuracy
    print("="*60)
    print("PER-CLASS ACCURACY")
    print("="*60)
    for i, class_name in enumerate(class_names):
        class_acc = cm[i, i] / cm[i].sum() if cm[i].sum() > 0 else 0
        print(f"  {class_name}: {class_acc*100:.2f}%")
    print()

    # Overall accuracy
    overall_acc = np.mean(y_pred == y_true) * 100

    # Plot history if available
    if history:
        plot_history(history)

    # Save logs
    if history and log_dir:
        os.makedirs(log_dir, exist_ok=True)

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        user_prefix = f"{user_id}_" if user_id else ""
        json_path = os.path.join(log_dir, f"training_log_{user_prefix}{timestamp}.json")
        csv_path = os.path.join(log_dir, f"training_log_{user_prefix}{timestamp}.csv")

        # JSON log
        log_data = {
            "timestamp": timestamp,
            "user_id": user_id,
            "framework": "pytorch",
            "model_type": model_type,
            "augmentations_count": len(augmentations) if augmentations else 0,
            "augmentations_sample": augmentations[0] if augmentations else None,
            "epochs_trained": len(history['train_acc']),
            "final_train_acc": float(history['train_acc'][-1]),
            "final_train_loss": float(history['train_loss'][-1]),
            "final_val_acc": float(history['val_acc'][-1]),
            "final_val_loss": float(history['val_loss'][-1]),
            "best_val_acc": float(max(history['val_acc'])),
            "history": {
                "accuracy": [float(x) for x in history['train_acc']],
                "val_accuracy": [float(x) for x in history['val_acc']],
                "loss": [float(x) for x in history['train_loss']],
                "val_loss": [float(x) for x in history['val_loss']],
                "learning_rate": [float(x) for x in history['lr']]
            },
            "confusion_matrix": cm.tolist(),
            "class_names": class_names
        }

        with open(json_path, "w") as f:
            json.dump(log_data, f, indent=4)

        # CSV log
        with open(csv_path, "w", newline="") as f:
            writer = csv.writer(f)
            writer.writerow(["epoch", "train_acc", "val_acc", "train_loss", "val_loss", "lr"])
            for i in range(len(history['train_acc'])):
                writer.writerow([
                    i + 1,
                    history['train_acc'][i],
                    history['val_acc'][i],
                    history['train_loss'][i],
                    history['val_loss'][i],
                    history['lr'][i]
                ])

        print(f"[INFO] Logs saved:")
        print(f"  - JSON: {json_path}")
        print(f"  - CSV: {csv_path}")

        # Print summary
        print(f"\n" + "="*60)
        print("TRAINING SUMMARY (PyTorch Deep Learning)")
        print("="*60)
        if user_id:
            print(f"User ID: {user_id}")
        print(f"Framework: PyTorch")
        print(f"Model Type: {model_type}")
        print(f"Augmentation Variations: {len(augmentations) if augmentations else 0}")
        print(f"Final Validation Accuracy: {history['val_acc'][-1]:.2f}%")
        print(f"Final Validation Loss: {history['val_loss'][-1]:.4f}")
        print(f"Best Validation Accuracy: {max(history['val_acc']):.2f}%")
        print(f"Epochs Trained: {len(history['train_acc'])}")
        print("="*60 + "\n")
```

```
Overwriting /content/drive/MyDrive/cassava/src/evaluate.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/evaluate_classical.py
import os
import json
import csv
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
from datetime import datetime
from tqdm import tqdm


def evaluate_classical_model(model, feature_extractor, scaler, data_config,
                             class_names, history=None, augmentations=None,
                             log_dir=None, user_id=None):
    """
    Evaluate classical ML model and save logs.

    Args:
        model: Trained Random Forest model
        feature_extractor: CNN feature extractor
        scaler: StandardScaler for features
        data_config: Dataset configuration
        class_names: List of class names
        history: Training history dictionary
        augmentations: List of augmentation configurations
        log_dir: Directory to save logs
        user_id: User identifier
    """
    print("[INFO] Evaluating classical ML model on validation set...")

    # Load validation data
    from train_classical import load_dataset

    data_dir = data_config['data_dir']
    specific_classes = data_config.get('specific_classes', None)

    image_paths, labels, _ = load_dataset(data_dir, specific_classes)

    # Split train/val (80/20) - same split as training
    from sklearn.model_selection import train_test_split
    _, val_paths, _, val_labels = train_test_split(
        image_paths, labels, test_size=0.2, random_state=42, stratify=labels
    )

    # Extract features
    print("[INFO] Extracting validation features...")
    X_val = feature_extractor.extract_batch_features(val_paths, batch_size=32)
    X_val_scaled = scaler.transform(X_val)
    y_val = np.array(val_labels)

    # Predict
    print("[INFO] Making predictions...")
    y_pred = model.predict(X_val_scaled)
    y_pred_proba = model.predict_proba(X_val_scaled)

    # Classification report
    print("\n" + "="*60)
    print("CLASSIFICATION REPORT")
    print("="*60)
    print(classification_report(y_val, y_pred, target_names=class_names))

    # Confusion matrix
    cm = confusion_matrix(y_val, y_pred)
    print("\n" + "="*60)
    print("CONFUSION MATRIX")
    print("="*60)
    print(cm)
    print()

    # Per-class accuracy
    print("="*60)
    print("PER-CLASS ACCURACY")
    print("="*60)
    for i, class_name in enumerate(class_names):
        class_acc = cm[i, i] / cm[i].sum() if cm[i].sum() > 0 else 0
        print(f"  {class_name}: {class_acc*100:.2f}%")
    print()

    # Overall accuracy
    overall_acc = np.mean(y_pred == y_val) * 100

    # Plot confusion matrix
    plot_confusion_matrix(cm, class_names)

    # Save logs
    if log_dir:
        os.makedirs(log_dir, exist_ok=True)
```

```
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            user_prefix = f"{user_id}_" if user_id else ""
            json_path = os.path.join(log_dir, f"training_log_{user_prefix}{timestamp}.json")
            csv_path = os.path.join(log_dir, f"training_log_{user_prefix}{timestamp}.csv")

            # JSON log
            log_data = {
                "timestamp": timestamp,
                "user_id": user_id,
                "framework": "classical_ml",
                "model_type": "random_forest",
                "augmentations_count": len(augmentations) if augmentations else 0,
                "n_estimators": model.n_estimators,
                "max_depth": model.max_depth,
                "train_acc": float(history['train_acc'][0]) if history else 0,
                "val_acc": float(history['val_acc'][0]) if history else overall_acc,
                "best_val_acc": float(history['val_acc'][0]) if history else overall_acc,
                "final_val_acc": overall_acc,
                "confusion_matrix": cm.tolist(),
                "class_names": class_names,
                "feature_dimension": X_val.shape[1]
            }

            with open(json_path, "w") as f:
                json.dump(log_data, f, indent=4)

            # CSV log
            with open(csv_path, "w", newline="") as f:
                writer = csv.writer(f)
                writer.writerow(["metric", "value"])
                writer.writerow(["train_acc", history['train_acc'][0] if history else 0])
                writer.writerow(["val_acc", overall_acc])
                writer.writerow(["n_estimators", model.n_estimators])
                writer.writerow(["max_depth", model.max_depth if model.max_depth else "unlimited"])

            print(f"[INFO] Logs saved:")
            print(f"  - JSON: {json_path}")
            print(f"  - CSV: {csv_path}")

            # Print summary
            print(f"\n" + "="*60)
            print("TRAINING SUMMARY (Classical ML)")
            print("="*60)
            if user_id:
                print(f"User ID: {user_id}")
            print(f"Framework: Classical ML (Random Forest)")
            print(f"Feature Extractor: EfficientNet-B0")
            print(f"Feature Dimension: {X_val.shape[1]}")
            print(f"N Estimators: {model.n_estimators}")
            print(f"Max Depth: {model.max_depth if model.max_depth else 'unlimited'}")
            print(f"Validation Accuracy: {overall_acc:.2f}%")
            print("="*60 + "\n")


    def plot_confusion_matrix(cm, class_names):
        """Plot confusion matrix."""
        plt.figure(figsize=(10, 8))

        # Normalize confusion matrix
        cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

        plt.imshow(cm_normalized, interpolation='nearest', cmap='Blues')
        plt.title('Confusion Matrix (Normalized)', fontsize=16, fontweight='bold')
        plt.colorbar(label='Accuracy')

        tick_marks = np.arange(len(class_names))
        plt.xticks(tick_marks, class_names, rotation=45, ha='right')
        plt.yticks(tick_marks, class_names)
        # Add text annotations
        thresh = cm_normalized.max() / 2.
        for i, j in np.ndindex(cm_normalized.shape):
            plt.text(j, i, f'{cm_normalized[i, j]:.2f}\n({cm[i, j]})',
                    ha="center", va="center",
                    color="white" if cm_normalized[i, j] > thresh else "black")

        plt.ylabel('True Label', fontsize=12)
        plt.xlabel('Predicted Label', fontsize=12)
        plt.tight_layout()
        plt.show()
```

```
Overwriting /content/drive/MyDrive/cassava/src/evaluate_classical.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/train.py
import os
import sys
import torch
import torch.nn as nn
import torch.optim as optim
from datetime import datetime
from tqdm import tqdm
import random
import numpy as np
from torch.cuda.amp import autocast, GradScaler
from torch.optim.lr_scheduler import OneCycleLR, CosineAnnealingWarmRestarts
import copy


# ============================================
# ADVANCED TRAINING COMPONENTS
# ============================================

class LabelSmoothingCrossEntropy(nn.Module):
    """Label smoothing for better generalization."""
    def __init__(self, smoothing=0.1):
        super().__init__()
        self.smoothing = smoothing
        self.confidence = 1.0 - smoothing

    def forward(self, pred, target):
        pred = pred.log_softmax(dim=-1)
        with torch.no_grad():
            true_dist = torch.zeros_like(pred)
            true_dist.fill_(self.smoothing / (pred.size(-1) - 1))
            true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        return torch.mean(torch.sum(-true_dist * pred, dim=-1))


class MixupAugmentation:
    """Mixup data augmentation."""
    def __init__(self, alpha=0.2):
        self.alpha = alpha

    def __call__(self, x, y):
        if self.alpha > 0:
            lam = np.random.beta(self.alpha, self.alpha)
        else:
            lam = 1

        batch_size = x.size(0)
        index = torch.randperm(batch_size).to(x.device)

        mixed_x = lam * x + (1 - lam) * x[index]
        y_a, y_b = y, y[index]
        return mixed_x, y_a, y_b, lam
```

```
class EarlyStopping:
    """Early stopping with patience."""
    def __init__(self, patience=7, min_delta=0.001, mode='max'):
        self.patience = patience
        self.min_delta = min_delta
        self.mode = mode
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.best_model_state = None

    def __call__(self, score, model):
        if self.best_score is None:
            self.best_score = score
            self.best_model_state = copy.deepcopy(model.state_dict())
        elif self.mode == 'max':
            if score < self.best_score + self.min_delta:
                self.counter += 1
                if self.counter >= self.patience:
                    self.early_stop = True
            else:
                self.best_score = score
                self.best_model_state = copy.deepcopy(model.state_dict())
                self.counter = 0

        return self.early_stop


class CutMix:
    """CutMix augmentation."""
    def __init__(self, alpha=1.0):
        self.alpha = alpha

    def __call__(self, x, y):
        lam = np.random.beta(self.alpha, self.alpha)
        batch_size = x.size(0)
        index = torch.randperm(batch_size).to(x.device)

        # Generate random box
        W = x.size(2)
        H = x.size(3)
        cut_rat = np.sqrt(1. - lam)
        cut_w = int(W * cut_rat)
        cut_h = int(H * cut_rat)

        cx = np.random.randint(W)
        cy = np.random.randint(H)

        bbx1 = np.clip(cx - cut_w // 2, 0, W)
        bby1 = np.clip(cy - cut_h // 2, 0, H)
        bbx2 = np.clip(cx + cut_w // 2, 0, W)
        bby2 = np.clip(cy + cut_h // 2, 0, H)

        x[:, :, bbx1:bbx2, bby1:bby2] = x[index, :, bbx1:bbx2, bby1:bby2]

        # Adjust lambda to exactly match pixel ratio
        lam = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (W * H))

        y_a, y_b = y, y[index]
        return x, y_a, y_b, lam


def set_seed(seed=42):
    """Set random seeds for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
```

```
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False


    # ============================================
    # MAIN TRAINING FUNCTION
    # ============================================

    def train_model(data_config, save_dir, epochs=10, batch_size=32,
                    augmentations=None, user_id=None,
                    device='cuda', use_amp=True, num_workers=4):
        """
        Train a PyTorch model with maximum accuracy optimizations.

        Optimizations for 15GB GPU:
        - Larger batch size (128)
        - Mixed precision training
        - Label smoothing
        - Mixup + CutMix augmentation
        - OneCycleLR scheduler
        - Gradient accumulation
        - Model checkpointing
        - Early stopping
        - Progressive unfreezing
        """
        from dataset import load_data
        from model import build_model

        # Set seeds for reproducibility
        set_seed(42)

        os.makedirs(save_dir, exist_ok=True)

        # Set device
        device = torch.device(device if torch.cuda.is_available() else 'cpu')
        print(f"[INFO] Using device: {device}")

        if device.type == 'cuda':
            print(f"[INFO] GPU: {torch.cuda.get_device_name(0)}")
            gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1024**3
            print(f"[INFO] GPU Memory: {gpu_memory:.1f} GB")
            print(f"[INFO] Optimizing for maximum performance...")

            # Optimize batch size for 15GB GPU
            if gpu_memory >= 14:
                recommended_batch = 128
                print(f"[INFO] Recommended batch size: {recommended_batch} (can handle large batches!)")
            else:
                recommended_batch = batch_size

        # Select random augmentation
        augment = None
        if augmentations and len(augmentations) > 0:
            augment = random.choice(augmentations)
            print(f"[INFO] Selected augmentation: {augment}")

        # Load data with optimized settings
        print("[INFO] Loading dataset...")
        train_loader, val_loader, class_names = load_data(
            data_config=data_config,
            batch_size=batch_size,
            augment=augment,
            user_id=user_id,
            num_workers=num_workers
        )
        num_classes = len(class_names)

        print(f"\n{'='*70}")
        print("Dataset Summary")
        print(f"{'='*70}")
        print(f"Number of Classes: {num_classes}")
        print(f"Classes: {', '.join(class_names)}")
        print(f"Training Batches: {len(train_loader)}")
        print(f"Validation Batches: {len(val_loader)}")
        print(f"Batch Size: {batch_size}")
        print(f"{'='*70}\n")

        # Build model (use EfficientNet-B3 for better accuracy with 15GB GPU)
        print("[INFO] Building model...")
        model_type = "efficientnet_b3" if device.type == 'cuda' else "efficientnet_b0"
        print(f"[INFO] Using {model_type} (optimized for available GPU memory)")
        model = build_model(num_classes=num_classes, model_type=model_type, device=device)

        # Advanced loss function with label smoothing
        criterion = LabelSmoothingCrossEntropy(smoothing=0.1)
        print(f"[INFO] Using Label Smoothing CrossEntropy (smoothing=0.1)")

        # Differential learning rates (backbone vs classifier)
        backbone_params = []
        classifier_params = []

        for name, param in model.named_parameters():
            if 'classifier' in name:
                classifier_params.append(param)
            else:
                backbone_params.append(param)

        # AdamW optimizer with differential learning rates
        optimizer = optim.AdamW([
            {'params': backbone_params, 'lr': 1e-4},      # Lower LR for pretrained backbone
            {'params': classifier_params, 'lr': 1e-3}     # Higher LR for classifier
        ], weight_decay=0.01)

        print(f"[INFO] Optimizer: AdamW with differential learning rates")
        print(f"[INFO]   - Backbone LR: 1e-4")
        print(f"[INFO]   - Classifier LR: 1e-3")

        # OneCycleLR scheduler for better convergence
        steps_per_epoch = len(train_loader)
        scheduler = OneCycleLR(
            optimizer,
            max_lr=[1e-3, 3e-3],  # Different max LR for each param group
            epochs=epochs,
            steps_per_epoch=steps_per_epoch,
            pct_start=0.3,
            anneal_strategy='cos',
            div_factor=25,
            final_div_factor=1e4
        )

        print(f"[INFO] Scheduler: OneCycleLR (cosine annealing)")

        # Mixed precision scaler
        scaler = GradScaler() if use_amp and device.type == 'cuda' else None
        if scaler:
            print(f"[INFO] Mixed Precision Training: Enabled")

        # Data augmentation techniques
        mixup = MixupAugmentation(alpha=0.2)
        cutmix = CutMix(alpha=1.0)
        print(f"[INFO] Advanced Augmentations: Mixup + CutMix")

        # Early stopping
        early_stopping = EarlyStopping(patience=10, min_delta=0.001, mode='max')
        print(f"[INFO] Early Stopping: patience=10")

        # Training history
        history = {
            'train_loss': [],
            'train_acc': [],
            'val_loss': [],
            'val_acc': [],
            'lr': []
        }

        best_val_acc = 0.0

        # Progressive unfreezing schedule
        unfreeze_schedule = {
            0: 0.7,    # Start with 70% frozen
            5: 0.5,    # Unfreeze more at epoch 5
            10: 0.3,   # Unfreeze even more at epoch 10
            15: 0.0    # Unfreeze all at epoch 15
        }

        print(f"\n{'='*70}")
        print(f"🚀 Starting Advanced Training (User: {user_id})")
        print(f"{'='*70}")
        print(f"Total Epochs: {epochs}")
        print(f"Gradient Accumulation: {'Adaptive' if batch_size < 64 else 'None'}")
        print(f"Progressive Unfreezing: Enabled")
        print(f"{'='*70}\n")

        # Training loop
        for epoch in range(epochs):
            print(f"{'='*70}")
            print(f"Epoch {epoch + 1}/{epochs}")
            print(f"{'='*70}")

            # Progressive unfreezing
            if epoch in unfreeze_schedule:
                freeze_ratio = unfreeze_schedule[epoch]
                if freeze_ratio == 0.0:
                    for param in model.parameters():
                        param.requires_grad = True
                    print(f"[INFO] 🔓 All layers unfrozen")
                else:
                    total_params = sum(1 for _ in model.backbone.parameters())
                    freeze_count = int(total_params * freeze_ratio)
                    for idx, param in enumerate(model.backbone.parameters()):
                        param.requires_grad = (idx >= freeze_count)
                    print(f"[INFO] 🔓 Unfrozen {total_params - freeze_count}/{total_params} backbone layers")

            # Training phase
            model.train()
            train_loss = 0.0
            train_correct = 0
            train_total = 0

            train_pbar = tqdm(train_loader, desc=f"Training",
                              bar_format='{l_bar}{bar:30}{r_bar}{bar:-10b}')

            for batch_idx, (images, labels) in enumerate(train_pbar):
                images, labels = images.to(device), labels.to(device)

                # Random augmentation selection: 50% normal, 25% Mixup, 25% CutMix
                aug_choice = random.random()

                if aug_choice < 0.5:
                    # Normal training
                    optimizer.zero_grad()
                    if scaler:
                        with autocast():
                            outputs = model(images)
                            loss = criterion(outputs, labels)

                        scaler.scale(loss).backward()

                        # Gradient clipping for stability
                        scaler.unscale_(optimizer)
                        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                        scaler.step(optimizer)
                        scaler.update()
                    else:
                        outputs = model(images)
                        loss = criterion(outputs, labels)
                        loss.backward()
                        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                        optimizer.step()

                    _, predicted = outputs.max(1)
                    train_correct += predicted.eq(labels).sum().item()

                elif aug_choice < 0.75:
                    # Mixup augmentation
                    mixed_images, labels_a, labels_b, lam = mixup(images, labels)

                    optimizer.zero_grad()

                    if scaler:
                        with autocast():
                            outputs = model(mixed_images)
                            loss = lam * criterion(outputs, labels_a) + (1 - lam) * criterion(outputs, labels_b)

                        scaler.scale(loss).backward()
                        scaler.unscale_(optimizer)
                        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                        scaler.step(optimizer)
                        scaler.update()
                    else:
                        outputs = model(mixed_images)
                        loss = lam * criterion(outputs, labels_a) + (1 - lam) * criterion(outputs, labels_b)
                        loss.backward()
                        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                        optimizer.step()

                    # Approximate accuracy for mixup
                    _, predicted = outputs.max(1)
                    train_correct += (lam * predicted.eq(labels_a).sum().item() +
                                    (1 - lam) * predicted.eq(labels_b).sum().item())

                else:
                    # CutMix augmentation
                    mixed_images, labels_a, labels_b, lam = cutmix(images, labels)

                    optimizer.zero_grad()

                    if scaler:
                        with autocast():
                            outputs = model(mixed_images)
                            loss = lam * criterion(outputs, labels_a) + (1 - lam) * criterion(outputs, labels_b)

                        scaler.scale(loss).backward()
                        scaler.unscale_(optimizer)
                        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                        scaler.step(optimizer)
                        scaler.update()
                    else:
                        outputs = model(mixed_images)
                        loss = lam * criterion(outputs, labels_a) + (1 - lam) * criterion(outputs, labels_b)
                        loss.backward()
                        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                        optimizer.step()

                    # Approximate accuracy for cutmix
                    _, predicted = outputs.max(1)
                    train_correct += (lam * predicted.eq(labels_a).sum().item() +
                                    (1 - lam) * predicted.eq(labels_b).sum().item())

                # Update scheduler after each batch (OneCycleLR requirement)
                scheduler.step()

                # Statistics
                train_loss += loss.item()
                train_total += labels.size(0)

                # Update progress bar
                current_acc = 100. * train_correct / train_total
                current_lr = optimizer.param_groups[0]['lr']
                train_pbar.set_postfix({
                    'Loss': f'{loss.item():.4f}',
                    'Acc': f'{current_acc:.2f}%',
                    'LR': f'{current_lr:.6f}'
                })

            train_loss /= len(train_loader)
            train_acc = 100. * train_correct / train_total

            # Validation phase
            model.eval()
            val_loss = 0.0
            val_correct = 0
            val_total = 0

            val_pbar = tqdm(val_loader, desc=f"Validation",
                           bar_format='{l_bar}{bar:30}{r_bar}{bar:-10b}')

            with torch.no_grad():
                for images, labels in val_pbar:
                    images, labels = images.to(device), labels.to(device)

                    if scaler:
                        with autocast():
                            outputs = model(images)
                            loss = criterion(outputs, labels)
                    else:
                        outputs = model(images)
                        loss = criterion(outputs, labels)

                    val_loss += loss.item()
                    _, predicted = outputs.max(1)
                    val_total += labels.size(0)
                    val_correct += predicted.eq(labels).sum().item()

                    # Update progress bar
                    current_acc = 100. * val_correct / val_total
                    val_pbar.set_postfix({
                        'Loss': f'{loss.item():.4f}',
                        'Acc': f'{current_acc:.2f}%'
                    })

            val_loss /= len(val_loader)
            val_acc = 100. * val_correct / val_total

            # Current learning rate
```

```
        current_lr = optimizer.param_groups[0]['lr']

        # Save history
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)
        history['lr'].append(current_lr)

        # Print epoch summary
        print(f"\n{'='*70}")
        print(f"✓ Epoch {epoch + 1} Complete!")
        print(f"  Train: Loss={train_loss:.4f}, Acc={train_acc:.2f}%")
        print(f"  Val:   Loss={val_loss:.4f}, Acc={val_acc:.2f}%")
        print(f"  LR:    {current_lr:.6f}")

        # Calculate overfitting indicator
        overfit_gap = train_acc - val_acc
        if overfit_gap > 15:
            print(f"  ⚠️  Overfitting detected (gap: {overfit_gap:.2f}%)")
        elif overfit_gap > 10:
            print(f"  ⚠️  Slight overfitting (gap: {overfit_gap:.2f}%)")
        else:
            print(f"  ✓ Healthy training (gap: {overfit_gap:.2f}%)")

        print(f"{'='*70}\n")

        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            checkpoint_path = os.path.join(save_dir, "best_model_checkpoint.pth")
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'val_acc': val_acc,
                'val_loss': val_loss,
                'class_names': class_names,
                'model_type': model_type
            }, checkpoint_path)
            print(f"[INFO] 💾 Best model saved (Val Acc: {val_acc:.2f}%)\n")

        # Early stopping check
        if early_stopping(val_acc, model):
            print(f"[INFO] 🛑 Early stopping triggered at epoch {epoch + 1}")
            print(f"[INFO] No improvement for {early_stopping.patience} epochs")
            print(f"[INFO] Restoring best model (Val Acc: {early_stopping.best_score:.2f}%)")
            model.load_state_dict(early_stopping.best_model_state)
            break

        # Memory cleanup every 5 epochs
        if (epoch + 1) % 5 == 0 and device.type == 'cuda':
            torch.cuda.empty_cache()

    # Save final model
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    model_filename = f"cassava_model_{user_id}_{timestamp}.pth"
    model_path = os.path.join(save_dir, model_filename)

    torch.save({
        'model_state_dict': model.state_dict(),
        'class_names': class_names,
        'num_classes': num_classes,
        'history': history,
        'model_type': model_type,
        'best_val_acc': best_val_acc,
        'final_val_acc': val_acc
    }, model_path)

    print(f"\n{'='*70}")
    print(f"🎉 Training Complete! (User: {user_id})")
    print(f"{'='*70}")
    print(f"Final Model:        {model_path}")
    print(f"Best Checkpoint:    {checkpoint_path}")
    print(f"Best Val Acc:       {best_val_acc:.2f}%")
    print(f"Final Val Acc:      {val_acc:.2f}%")
    print(f"Final Train Acc:    {train_acc:.2f}%")
    print(f"Total Epochs:       {len(history['train_acc'])}")
    print(f"Model Architecture: {model_type}")
    print(f"{'='*70}\n")

    return model, history, val_loader, class_names
```

```
Overwriting /content/drive/MyDrive/cassava/src/train.py
```

```
%%writefile /content/drive/MyDrive/cassava/src/train_classical.py
import os
import sys
import torch
import numpy as np
import joblib
from datetime import datetime
from tqdm import tqdm
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import torchvision.models as models
from torchvision import transforms
from PIL import Image
import random


def set_seed(seed=42):
    """Set random seeds for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)


class FeatureExtractor:
    """Extract features using pre-trained CNN."""

    def __init__(self, device='cuda'):
        self.device = torch.device(device if torch.cuda.is_available() else 'cpu')

        # Load pre-trained EfficientNet
        from torchvision.models import efficientnet_b0, EfficientNet_B0_Weights
        weights = EfficientNet_B0_Weights.IMAGENET1K_V1
        self.model = efficientnet_b0(weights=weights)

        # Remove classifier to get features
        self.model.classifier = torch.nn.Identity()
        self.model = self.model.to(self.device)
        self.model.eval()

        # Define transforms
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
        ])

        print(f"[INFO] Feature extractor loaded on {self.device}")

    def extract_features(self, image_path):
        """Extract features from a single image."""
        image = Image.open(image_path).convert('RGB')
        img_tensor = self.transform(image).unsqueeze(0).to(self.device)

        with torch.no_grad():
            features = self.model(img_tensor)

        return features.cpu().numpy().flatten()

    def extract_batch_features(self, image_paths, batch_size=32):
        """Extract features from multiple images."""
        all_features = []

        for i in tqdm(range(0, len(image_paths), batch_size), desc="Extracting features"):
            batch_paths = image_paths[i:i + batch_size]
            batch_tensors = []

            for img_path in batch_paths:
                try:
                    image = Image.open(img_path).convert('RGB')
                    img_tensor = self.transform(image)
                    batch_tensors.append(img_tensor)
                except Exception as e:
                    print(f"[WARNING] Failed to load {img_path}: {e}")
                    continue

            if batch_tensors:
                batch_tensor = torch.stack(batch_tensors).to(self.device)

                with torch.no_grad():
                    features = self.model(batch_tensor)

                all_features.append(features.cpu().numpy())

        if all_features:
            return np.vstack(all_features)
        else:
            return np.array([])


def load_dataset(data_dir, specific_classes=None):
    """Load dataset and return image paths with labels."""
    image_paths = []
    labels = []

    # Get classes
    classes = sorted([d for d in os.listdir(data_dir)
                     if os.path.isdir(os.path.join(data_dir, d))
                     and not d.startswith('.')])

    if specific_classes:
        classes = [c for c in classes if c in specific_classes]

    class_to_idx = {cls_name: idx for idx, cls_name in enumerate(classes)}

    # Load all image paths
    for class_name in classes:
        class_dir = os.path.join(data_dir, class_name)
        class_idx = class_to_idx[class_name]

        for img_name in os.listdir(class_dir):
            if img_name.lower().endswith(('.jpg', '.jpeg', '.png')):
                img_path = os.path.join(class_dir, img_name)
                image_paths.append(img_path)
                labels.append(class_idx)

    return image_paths, labels, list(class_to_idx.keys())


def train_classical_model(data_config, save_dir, n_estimators=100, max_depth=None,
                          augmentations=None, user_id=None):
    """
    Train a classical ML model (Random Forest) with CNN features.

    Args:
        data_config: Dictionary with dataset configuration
        save_dir: Directory to save model
        n_estimators: Number of trees in Random Forest
        max_depth: Maximum depth of trees (None for unlimited)
        augmentations: Augmentation configurations (not used for classical)
        user_id: User identifier

    Returns:
        model, feature_extractor, scaler, history, class_names
    """
    set_seed(42)

    os.makedirs(save_dir, exist_ok=True)

    data_dir = data_config['data_dir']
    specific_classes = data_config.get('specific_classes', None)

    print(f"\n{'='*70}")
    print("🚀 Starting Classical ML Training")
    print(f"{'='*70}\n")

    # Load dataset
    print("[INFO] Loading dataset...")
    image_paths, labels, class_names = load_dataset(data_dir, specific_classes)

    print(f"[INFO] Total images: {len(image_paths):,}")
    print(f"[INFO] Classes: {class_names}")

    # Split train/val (80/20)
    from sklearn.model_selection import train_test_split

    train_paths, val_paths, train_labels, val_labels = train_test_split(
        image_paths, labels, test_size=0.2, random_state=42, stratify=labels
    )

    print(f"[INFO] Training samples: {len(train_paths):,}")
    print(f"[INFO] Validation samples: {len(val_paths):,}")

    # Initialize feature extractor
    print("\n[INFO] Initializing CNN feature extractor...")
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    feature_extractor = FeatureExtractor(device=device)

    # Extract features
    print("\n[INFO] Extracting training features...")
    X_train = feature_extractor.extract_batch_features(train_paths, batch_size=32)
    y_train = np.array(train_labels)

    print("[INFO] Extracting validation features...")
    X_val = feature_extractor.extract_batch_features(val_paths, batch_size=32)
    y_val = np.array(val_labels)

    print(f"\n[INFO] Feature shape: {X_train.shape}")
    print(f"[INFO] Feature dimension: {X_train.shape[1]}")

    # Scale features
```

```
        print("\n[INFO] Scaling features...")
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        # Train Random Forest
        print(f"\n[INFO] Training Random Forest...")
        print(f"[INFO] n_estimators: {n_estimators}")
        print(f"[INFO] max_depth: {max_depth if max_depth else 'unlimited'}")

        model = RandomForestClassifier(
            n_estimators=n_estimators,
            max_depth=max_depth,
            random_state=42,
            n_jobs=-1,
            verbose=1
        )

        model.fit(X_train_scaled, y_train)

        # Evaluate
        print("\n[INFO] Evaluating model...")
        train_pred = model.predict(X_train_scaled)
        val_pred = model.predict(X_val_scaled)

        train_acc = accuracy_score(y_train, train_pred) * 100
        val_acc = accuracy_score(y_val, val_pred) * 100

        print(f"\n{'='*70}")
        print("✓ Training Complete!")
        print(f"{'='*70}")
        print(f"Training Accuracy:   {train_acc:.2f}%")
        print(f"Validation Accuracy: {val_acc:.2f}%")
        print(f"{'='*70}\n")

        # Create history for compatibility
        history = {
            'train_acc': [train_acc],
            'val_acc': [val_acc],
            'train_loss': [0],  # Not applicable for RF
            'val_loss': [0]
        }

        # Feature importance
        feature_importance = model.feature_importances_
        top_features = np.argsort(feature_importance)[-10:][::-1]

        print("[INFO] Top 10 most important features:")
        for i, feat_idx in enumerate(top_features, 1):
            print(f"  {i}. Feature {feat_idx}: {feature_importance[feat_idx]:.4f}")

        # Save model
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        model_filename = f"random_forest_{user_id}_{timestamp}.pkl"
        model_path = os.path.join(save_dir, model_filename)

        # Save everything
        save_data = {
            'model': model,
            'feature_extractor_type': 'efficientnet_b0',
            'scaler': scaler,
            'class_names': class_names,
            'num_classes': len(class_names),
            'history': history,
            'train_acc': train_acc,
            'val_acc': val_acc,
            'n_estimators': n_estimators,
            'max_depth': max_depth,
            'feature_importance': feature_importance,
            'timestamp': timestamp,
            'user_id': user_id
        }

        joblib.dump(save_data, model_path)

        print(f"\n[INFO] 💾 Model saved: {model_path}")
        print(f"[INFO] Model size: {os.path.getsize(model_path) / 1024**2:.2f} MB")

        return model, feature_extractor, scaler, history, class_names
```

```
Overwriting /content/drive/MyDrive/cassava/src/train_classical.py
```

```
%%writefile /content/drive/MyDrive/cassava/main.py
#!/usr/bin/env python3
"""
Cassava Disease Classification - Unified Entry Point
Supports both Classical ML (Random Forest) and Deep Learning (PyTorch)
"""

import argparse
import os
import socket
import sys
import torch
import glob
import numpy as np
import random
from itertools import product
from datetime import datetime

# Add src to path
sys.path.insert(0, os.path.join(os.path.dirname(__file__), 'src'))


# Import modular components
from best_models_tracker import BestModelsTracker, get_best_model_path


def optimize_pytorch():
    """Optimize PyTorch for maximum performance."""
    if torch.cuda.is_available():
        torch.backends.cudnn.benchmark = True
        torch.backends.cuda.matmul.allow_tf32 = True
        torch.backends.cudnn.allow_tf32 = True
        print("[INFO] PyTorch optimizations enabled")


def detect_device():
    """Detect and configure compute device."""
    if torch.cuda.is_available():
        device_name = torch.cuda.get_device_name(0)
        device_count = torch.cuda.device_count()
        total_memory = torch.cuda.get_device_properties(0).total_memory / 1024**3

        print(f"[INFO] GPU Available: {device_name} ({device_count} device(s))")
        print(f"[INFO] GPU Memory: {total_memory:.1f} GB total")
        return "GPU"
    else:
        print("[INFO] Using CPU (GPU not available)")
        return "CPU"


def get_user_identifier():
    """Generate unique user identifier."""
    hostname = socket.gethostname()
    import getpass
    try:
        username = getpass.getuser()
    except:
        username = "unknown"
    return f"{username}_{hostname}"


def detect_dataset_structure(working_dir, specific_classes=None):
    """Detect and validate dataset structure."""
    possible_dirs = [
        os.path.join(working_dir, "datasets"),
        os.path.join(working_dir, "data"),
        os.path.join(working_dir, "train_images"),
        working_dir
    ]

    for base_dir in possible_dirs:
        if not os.path.exists(base_dir):
            continue

        subdirs = sorted([d for d in os.listdir(base_dir)
                          if os.path.isdir(os.path.join(base_dir, d))
                          and not d.startswith('.')])

        if specific_classes:
            subdirs = [d for d in subdirs if d in specific_classes]
            if not subdirs:
                continue

        if subdirs:
            first_subdir = os.path.join(base_dir, subdirs[0])
            if not os.path.exists(first_subdir):
                continue

            images = [f for f in os.listdir(first_subdir)
                      if f.lower().endswith(('.jpg', '.jpeg', '.png'))]

            if images:
                total_images = 0
                class_counts = {}
                for subdir in subdirs:
                    subdir_path = os.path.join(base_dir, subdir)
                    img_count = len([f for f in os.listdir(subdir_path)
                                     if f.lower().endswith(('.jpg', '.jpeg', '.png'))])
                    class_counts[subdir] = img_count
                    total_images += img_count

                return {
                    'type': 'directory',
                    'path': base_dir,
                    'classes': subdirs,
                    'class_counts': class_counts,
                    'total_images': total_images,
                    'specific_classes': specific_classes
                }

    return None


# ============================
# MODE: TRAIN
# ============================
def mode_train(args):
    """Execute training mode."""
    print(f"\n{'='*70}")
    print(f"🌿 TRAINING MODE - {args.model_type.upper()}")
    print(f"{'='*70}\n")
```

```
        # Dataset detection
        print("[INFO] Detecting dataset structure...")
        dataset_info = detect_dataset_structure(args.work_dir, specific_classes=args.classes)

        if not dataset_info:
            print("❌ ERROR: Dataset not found or invalid structure.")
            sys.exit(1)

        print(f"\n{'='*70}")
        print("✅ Dataset Detected Successfully!")
        print(f"{'='*70}")
        print(f"Dataset Path: {dataset_info['path']}")
        print(f"Classes: {len(dataset_info['classes'])}")
        print(f"Total Images: {dataset_info['total_images']:,}")
        print(f"{'='*70}\n")

        data_config = {
            'type': 'directory',
            'data_dir': dataset_info['path'],
            'specific_classes': args.classes
        }

        # Route to appropriate trainer
        if args.model_type == "deep_learning":
            train_deep_learning(args, data_config)
        else:
            train_classical(args, data_config)

        # Update best models tracker after training
        print(f"\n[INFO] Updating best models tracker...")
        tracker = BestModelsTracker(args.work_dir)
        tracker.update_best_models()


    def train_deep_learning(args, data_config):
        """Train deep learning model."""
        from train import train_model
        from evaluate import evaluate_model

        augmentations = generate_augmentations(args)
        print(f"[INFO] Generated {len(augmentations)} augmentation combinations.\n")

        device = 'cuda' if torch.cuda.is_available() else 'cpu'

        model, history, val_loader, class_names = train_model(
            data_config=data_config,
            save_dir=args.save_dir,
            epochs=args.epochs,
            batch_size=args.batch_size,
            augmentations=augmentations,
            user_id=args.user_id,
            device=device,
            use_amp=args.use_amp,
            num_workers=args.num_workers
        )

        print(f"\n[INFO] Training complete! Evaluating model...\n")
        evaluate_model(
            model=model,
            val_loader=val_loader,
            class_names=class_names,
            history=history,
            augmentations=augmentations,
            log_dir=args.log_dir,
            user_id=args.user_id,
            device=device,
            model_type="deep_learning"
        )

        print_completion_message(args, "Deep Learning")


    def train_classical(args, data_config):
        """Train classical ML model."""
        from train_classical import train_classical_model
        from evaluate_classical import evaluate_classical_model

        augmentations = generate_augmentations(args)
        print(f"[INFO] Generated {len(augmentations)} augmentation combinations.\n")

        model, feature_extractor, scaler, history, class_names = train_classical_model(
            data_config=data_config,
            save_dir=args.save_dir,
            n_estimators=args.n_estimators,
            max_depth=args.max_depth,
            augmentations=augmentations,
            user_id=args.user_id
        )

        print(f"\n[INFO] Training complete! Evaluating model...\n")
        evaluate_classical_model(
            model=model,
            feature_extractor=feature_extractor,
            scaler=scaler,
            data_config=data_config,
            class_names=class_names,
            history=history,
            augmentations=augmentations,
            log_dir=args.log_dir,
            user_id=args.user_id
        )

        print_completion_message(args, "Classical ML")


    def generate_augmentations(args):
        """Generate augmentation combinations."""
        brightness_levels = [round(x, 2) for x in np.arange(0.0, args.brightness + 0.1, 0.1)] if args.brightness :
        rotation_angles = list(range(0, 360, args.rotation)) if args.rotation > 0 else [0]
        zoom_levels = [round(z, 2) for z in np.arange(0.0, args.zoom + 0.1, 0.1)] if args.zoom > 0 else [0.0]

        flip_modes = [
            {"flip_horizontal": False, "flip_vertical": False},
            {"flip_horizontal": True,  "flip_vertical": False},
            {"flip_horizontal": False, "flip_vertical": True},
            {"flip_horizontal": True,  "flip_vertical": True}
        ] if args.flip else [{"flip_horizontal": False, "flip_vertical": False}]

        augmentations = [
            {
                "rotation": rot,
                "brightness": bright,
                "zoom": zoom,
                "flip_horizontal": flip["flip_horizontal"],
                "flip_vertical": flip["flip_vertical"]
            }
            for rot, bright, zoom, flip in product(rotation_angles, brightness_levels, zoom_levels, flip_modes)
        ]

        random.shuffle(augmentations)
        return augmentations[:200]


    def print_completion_message(args, model_type):
        """Print training completion message."""
        print(f"\n{'='*70}")
        print(f"✅ {model_type.upper()} TRAINING COMPLETE")
        print(f"{'='*70}")
        print(f"📁 Model saved:   {args.save_dir}")
        print(f"📊 Logs saved:    {args.log_dir}")
        print(f"\n📈 Next Steps:")
        print(f"  View dashboard:  python main.py --mode dashboard --work_dir {args.work_dir}")
        print(f"  Test model:      python main.py --mode test --model_type {args.model_type}")
        print(f"{'='*70}\n")


    # =============================
    # MODE: DASHBOARD
    # =============================
    def mode_dashboard(args):
        """Execute dashboard mode."""
        from dashboard_main import show_comparison_dashboard

        print(f"\n{'='*70}")
        print(f"📊 COMPARISON DASHBOARD MODE")
        print(f"{'='*70}\n")

        show_comparison_dashboard(
            log_dir=os.path.join(args.work_dir, "outputs", "logs"),
            save_plots=args.save_plots,
            work_dir=args.work_dir
        )


    # =============================
    # MODE: TEST
    # =============================
    def mode_test(args):
        """Execute testing mode with Gradio interface."""
        print(f"\n{'='*70}")
        print(f"🧪 TESTING MODE - {args.model_type.upper()}")
        print(f"{'='*70}\n")

        # Get best model path
        if args.model_path:
            model_path = args.model_path
            print(f"[INFO] Using specified model: {os.path.basename(model_path)}")
        else:
            print(f"[INFO] Finding BEST {args.model_type} model from tracker...")
            model_path = get_best_model_path(args.work_dir, args.model_type)

            if not model_path:
                print(f"❌ ERROR: No trained {args.model_type} model found!")
                print(f"\n💡 Train a model first:")
                print(f"   python main.py --mode train --model_type {args.model_type}")
                sys.exit(1)

        if not os.path.exists(model_path):
            print(f"❌ ERROR: Model not found at: {model_path}")
            sys.exit(1)

        print(f"\n{'='*70}")
        print(f"🎯 SELECTED MODEL")
        print(f"{'='*70}")
        print(f"Path: {model_path}")
        print(f"Type: {args.model_type}")
        print(f"Size: {os.path.getsize(model_path) / 1024**2:.2f} MB")
        print(f"{'='*70}\n")

        # Launch interface
        if args.model_type == "deep_learning":
            from test_deep_learning import create_gradio_interface as create_dl_interface
            create_dl_interface(
                model_path=model_path,
                share=args.share,
                server_name=args.server_name,
                server_port=args.server_port
            )
        else:
            from test_classical import create_gradio_interface as create_cl_interface
            create_cl_interface(
                model_path=model_path,
                share=args.share,
                server_name=args.server_name,
                server_port=args.server_port
            )


    def mode_test_both(args):
        """Execute testing mode for BOTH models simultaneously."""
        import threading
        import time

        print(f"\n{'='*70}")
        print(f"🧪 TESTING MODE - BOTH MODELS")
        print(f"{'='*70}\n")

        # Get best models from tracker
        print(f"[INFO] Finding best models from tracker...")
        dl_model_path = get_best_model_path(args.work_dir, "deep_learning")
        cl_model_path = get_best_model_path(args.work_dir, "classical")

        if not dl_model_path and not cl_model_path:
            print(f"❌ ERROR: No trained models found!")
            print(f"\n💡 Train models first:")
            print(f"   python main.py --mode train --model_type deep_learning")
            print(f"   python main.py --mode train --model_type classical")
            sys.exit(1)

        print(f"\n{'='*70}")
        print(f"🎯 SELECTED MODELS")
        print(f"{'='*70}")

        if dl_model_path:
            print(f"\n📘 Deep Learning Model:")
            print(f"   Path: {dl_model_path}")
            print(f"   Size: {os.path.getsize(dl_model_path) / 1024**2:.2f} MB")
            print(f"   Port: {args.server_port}")

        if cl_model_path:
            print(f"\n📕 Classical ML Model:")
            print(f"   Path: {cl_model_path}")
            print(f"   Size: {os.path.getsize(cl_model_path) / 1024**2:.2f} MB")
            print(f"   Port: {args.server_port + 1}")

        print(f"{'='*70}\n")

        # Launch interfaces
        def launch_dl():
            if dl_model_path:
                from test_deep_learning import create_gradio_interface as create_dl_interface
                create_dl_interface(
                    model_path=dl_model_path,
                    share=args.share,
                    server_name=args.server_name,
                    server_port=args.server_port
                )

        def launch_cl():
            if cl_model_path:
                from test_classical import create_gradio_interface as create_cl_interface
                create_cl_interface(
                    model_path=cl_model_path,
                    share=args.share,
                    server_name=args.server_name,
                    server_port=args.server_port + 1
                )

        threads = []
        if dl_model_path:
            t1 = threading.Thread(target=launch_dl)
            t1.start()
            threads.append(t1)

        if cl_model_path:
            time.sleep(2)  # Avoid port conflicts
            t2 = threading.Thread(target=launch_cl)
            t2.start()
            threads.append(t2)

        for t in threads:
            t.join()


    # =============================
    # MAIN ENTRY POINT
    # =============================
    def main():
        """Main entry point for all operations."""

        banner = """
        ╔════════════════════════════════════════════════════════════════╗
        ║                                                                ║
        ║           🌿 CASSAVA DISEASE CLASSIFICATION SYSTEM 🌿           ║
        ║                                                                ║
        ║          Deep Learning vs Classical ML Comparison            ║
        ║                                                                ║
        ╚════════════════════════════════════════════════════════════════╝
        """
        print(banner)

        optimize_pytorch()
        DEVICE = detect_device()

        parser = argparse.ArgumentParser(
            description="🌿 Cassava Disease Classification System",
            formatter_class=argparse.RawDescriptionHelpFormatter
        )

        parser.add_argument("--mode", type=str, required=True,
                           choices=["train", "test", "test_both", "dashboard"],
                           help="Operation mode")
        parser.add_argument("--model_type", type=str,
                           choices=["deep_learning", "classical"],
                           help="Model type (not needed for test_both)")
        parser.add_argument("--work_dir", type=str,
                           default="/content/drive/MyDrive/cassava",
                           help="Working directory")
        parser.add_argument("--user_id", type=str, default=None,
                           help="User identifier")

        # Training
        train_group = parser.add_argument_group('Training Options')
        train_group.add_argument("--epochs", type=int, default=15)
        train_group.add_argument("--batch_size", type=int, default=64)
        train_group.add_argument("--num_workers", type=int, default=4)
        train_group.add_argument("--classes", type=str, nargs='+', default=None)
        train_group.add_argument("--use_amp", action="store_true", default=True)
        train_group.add_argument("--no_amp", action="store_false", dest="use_amp")
        # Classical ML
        classical_group = parser.add_argument_group('Classical ML')
        classical_group.add_argument("--n_estimators", type=int, default=100)
        classical_group.add_argument("--max_depth", type=int, default=None)

        # Augmentation
        aug_group = parser.add_argument_group('Augmentation')
        aug_group.add_argument("--rotation", type=int, default=15)
        aug_group.add_argument("--zoom", type=float, default=0.2)
        aug_group.add_argument("--flip", action="store_true", default=True)
        aug_group.add_argument("--brightness", type=float, default=0.2)

        # Dashboard
        dash_group = parser.add_argument_group('Dashboard')
        dash_group.add_argument("--save_plots", action="store_true", default=True)
        dash_group.add_argument("--no_save_plots", action="store_false", dest="save_plots")

        # Testing
        test_group = parser.add_argument_group('Testing')
        test_group.add_argument("--model_path", type=str, default=None)
        test_group.add_argument("--share", action="store_true")
        test_group.add_argument("--server_port", type=int, default=7860)
        test_group.add_argument("--server_name", type=str, default="0.0.0.0")

        args = parser.parse_args()

        # Validate
        if args.mode in ["train", "test"] and not args.model_type:
            parser.error(f"--model_type is required for {args.mode} mode")

        # Setup user ID
        if not args.user_id:
            args.user_id = get_user_identifier()
        args.user_id = args.user_id.replace(" ", "_")

        # Setup directories
        model_subdir = args.model_type if args.mode in ["train", "test"] else "all"
        args.save_dir = os.path.join(args.work_dir, "outputs", "models", args.user_id, model_subdir)
        args.log_dir = os.path.join(args.work_dir, "outputs", "logs", args.user_id, model_subdir)

        if args.mode == "train":
            os.makedirs(args.save_dir, exist_ok=True)
            os.makedirs(args.log_dir, exist_ok=True)

        # Execute
        try:
            if args.mode == "train":
                mode_train(args)
            elif args.mode == "dashboard":
                mode_dashboard(args)
            elif args.mode == "test":
                mode_test(args)
            elif args.mode == "test_both":
                mode_test_both(args)
        except KeyboardInterrupt:
            print("\n\n⚠️ Operation interrupted by user")
            sys.exit(0)
        except Exception as e:
            print(f"\n❌ ERROR: {e}")
            import traceback
            traceback.print_exc()
            sys.exit(1)


    if __name__ == "__main__":
        main()
```

```
Overwriting /content/drive/MyDrive/cassava/main.py
```

```
# 1. Install PyTorch
!pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
!pip install tqdm scikit-learn pandas matplotlib colorama tabulate
```

Show hidden output

```
#  Run Deep Learning training!
!python /content/drive/MyDrive/cassava/main.py  --model_type deep_learning \
  --mode train \
  --user_id wicky \
  --epochs 40 \
  --batch_size 128 \
  --rotation 35 \
  --brightness 0.3 \
  --zoom 0.2 \
  --flip \
  --num_workers 12 \
  --use_amp
```

Show hidden output

```
#  Run Classical training!
!python /content/drive/MyDrive/cassava/main.py --model_type classical \
  --mode train \
  --user_id wicky \
  --n_estimators 150 \
  --rotation 25 \
  --brightness 0.2
```

```
╔════════════════════════════════════════════════════════════════╗
║                                                                ║
║           🌿 CASSAVA DISEASE CLASSIFICATION SYSTEM 🌿            ║
║                                                                ║
║          Deep Learning vs Classical ML Comparison             ║
║                                                                ║
╚════════════════════════════════════════════════════════════════╝

/usr/local/lib/python3.12/dist-packages/torch/backends/__init__.py:46: UserWarning: Please use the new API se
  self.setter(val)
[INFO] PyTorch optimizations enabled
[INFO] GPU Available: Tesla T4 (1 device(s))
[INFO] GPU Memory: 14.7 GB total

======================================================================
⚙️ CONFIGURATION
======================================================================
Mode:           TRAIN
Model Type:     CLASSICAL
Work Directory: /content/drive/MyDrive/cassava
User ID:        wicky
Device:         GPU
PyTorch:        2.9.0+cu126
======================================================================

======================================================================
🌿 TRAINING MODE - CLASSICAL
======================================================================

[INFO] Detecting dataset structure...

======================================================================
✓ Dataset Detected Successfully!
```

```
======================================================================
Dataset Path: /content/drive/MyDrive/cassava/datasets
Classes: 5
Total Images: 11,397

📊 Class Distribution:
  Cassava___bacterial_blight │ ████                                              │ 1,087 images (  9.5%)
  Cassava___brown_streak_disease │ █████████                                         │ 2,189 images ( 19.2%)
  Cassava___green_mottle │ ██████████                                        │ 2,386 images ( 20.9%)
  Cassava___healthy │ ███████████                                       │ 2,577 images ( 22.6%)
  Cassava___mosaic_disease │ █████████████                                     │ 3,158 images ( 27.7%)
======================================================================

[INFO] Generated 200 augmentation combinations.

======================================================================
⚙️ Classical ML Training Configuration
======================================================================
  Model Type:         Classical (Random Forest)
  Feature Extractor:  Pre-trained CNN (EfficientNet)
  Estimators:         150
  Max Depth:          None (unlimited)
  Augmentations:      200 variations
```

```
#  Run training!
!python /content/drive/MyDrive/cassava/main.py --model_type deep_learning \
  --mode train \
  --user_id riho \
  --epochs 35 \
  --batch_size 128 \
  --use_amp \
  --rotation 30 \
  --brightness 0.3 \
  --zoom 0.2 \
  --flip \
  --num_workers 12
```

Show hidden output

```
#  Run Classical training!
!python /content/drive/MyDrive/cassava/main.py --model_type classical \
  --mode train \
  --user_id riho \
  --n_estimators 150 \
  --rotation 25 \
  --brightness 0.2
```

Show hidden output

```
#  Run Classical training!
!python /content/drive/MyDrive/cassava/main.py --model_type classical \
  --mode train \
  --user_id edward \
  --n_estimators 150 \
  --rotation 25 \
  --brightness 0.2
```

Show hidden output

```
#  Run training!
```

```
!python /content/drive/MyDrive/cassava/main.py --model_type deep_learning \
  --mode train \
  --user_id edward \
  --epochs 60 \
  --batch_size 128 \
  --rotation 15 \
  --brightness 0.3 \
  --zoom 0.2 \
  --flip \
  --num_workers 15 \
  --use_amp
```

Show hidden output

```
!pip install gradio
```

Show hidden output

```
#  Run Dashboard!
!python /content/drive/MyDrive/cassava/main.py \
  --mode dashboard
```

Show hidden output

```
#  Run Test!
!python /content/drive/MyDrive/cassava/main.py --model_type classical \
  --mode test --share
```

```
╔══════════════════════════════════════════════════════════════╗
║                                                              ║
║            🌿 CASSAVA DISEASE CLASSIFICATION SYSTEM 🌿          ║
║                                                              ║
║          Deep Learning vs Classical ML Comparison           ║
║                                                              ║
╚══════════════════════════════════════════════════════════════╝

[INFO] Using CPU (GPU not available)

======================================================================
🧪 TESTING MODE - CLASSICAL
======================================================================

[INFO] Finding BEST classical model from tracker...

======================================================================
🎯 SELECTED MODEL
======================================================================
Path: /content/drive/MyDrive/cassava/outputs/models/wicky/classical/random_forest_wicky_20251208_151333.pkl
Type: classical
Size: 47.90 MB
======================================================================

[INFO] Using device: cpu
[INFO] Loading classical ML model from: /content/drive/MyDrive/cassava/outputs/models/wicky/classical/random_fo
[INFO] Model type: RandomForestClassifier
[INFO] Number of estimators: 150
[INFO] Classes: ['Cassava___bacterial_blight', 'Cassava___brown_streak_disease', 'Cassava___green_mottle', 'Cas
[INFO] Initializing feature extractor...
[INFO] Feature extractor loaded on cpu
[INFO] Classical ML model loaded successfully!

======================================================================
🚀 Launching Classical ML Gradio Interface
======================================================================
Model: random_forest_wicky_20251208_151333.pkl
Classes: ['Cassava___bacterial_blight', 'Cassava___brown_streak_disease', 'Cassava___green_mottle', 'Cassava___
```

```
Device: cpu
Port: 7860
Share: Yes
======================================================================

* Running on local URL:  http://0.0.0.0:7860
* Running on public URL: https://cb6408b6dc0fe87f9b.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the te
Keyboard interruption in main thread... closing server.
Killing tunnel 0.0.0.0:7860 <> https://cb6408b6dc0fe87f9b.gradio.live
```

```
#  Run Test!
!python /content/drive/MyDrive/cassava/main.py --model_type deep_learning \
  --mode test --share
```

Show hidden output

```
#  Run Test!
!python /content/drive/MyDrive/cassava/main.py --mode test_both --share
```

```
    File "/usr/local/lib/python3.12/dist-packages/starlette/middleware/errors.py", line 186, in __call__
      raise exc
    File "/usr/local/lib/python3.12/dist-packages/starlette/middleware/errors.py", line 164, in __call__
      await self.app(scope, receive, _send)
    File "/usr/local/lib/python3.12/dist-packages/gradio/brotli_middleware.py", line 74, in __call__
      return await self.app(scope, receive, send)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    File "/usr/local/lib/python3.12/dist-packages/gradio/route_utils.py", line 882, in __call__
```