```python
# ================== IMPORTS ===================
import os
import json
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import transforms
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import random

# Scikit-learn
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics.pairwise import cosine_similarity

# Transformers & PEFT
os.environ["TOKENIZERS_PARALLELISM"] = "false"
from transformers import AutoTokenizer, BertModel
from peft import get_peft_model, LoraConfig

print("PyTorch Version:", torch.__version__)
print("CUDA Available:", torch.cuda.is_available())
```

```python
# ================== CONFIG ===================
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# --- Paths ---
IMG_DIR = "/kaggle/input/dermnet/train"
TEST_DIR = "/kaggle/working/test_images"
TEXT_JSON = "/kaggle/input/text-description/text_final.json"
KB_JSON = "/kaggle/input/knowledge-base-new/kb_new.json"
TEST_TEXT_JSON = "/kaggle/input/test-2-final/test2_final.json"

# --- Checkpoints ---
DINO_CHECKPOINT_PATH = "/kaggle/working/dino_finetuned.pth"
MAGEVPRO_CHECKPOINT_PATH = "/kaggle/working/magevpro_best.pth"

# Add these lines to your CONFIG cell
S1_CURVES_PATH = "/kaggle/working/training_curves_S1.png"
S2_CURVES_PATH = "/kaggle/working/training_curves_S2.png"

# --- Stage 1: Vision Pre-training ---
BATCH_SIZE_S1 = 32
EPOCHS_S1 = 20
PATIENCE_S1 = 4
LEARNING_RATE_S1 = 1e-4
WEIGHT_DECAY_S1 = 1e-4
TEST_SIZE_S1 = 0.2
UNFREEZE_DINO_S1 = 2 # Unfreeze last 2 blocks

# --- Stage 2: Multimodal Fine-tuning ---
BATCH_SIZE_S2 = 16
EPOCHS_S2 = 30
PATIENCE_S2 = 5
LEARNING_RATE_S2 = 5e-5 # LR for head and text LoRA
VISION_LR_S2 = 1e-6       # A smaller LR for the vision backbone
WEIGHT_DECAY_S2 = 1e-2
TEST_SIZE_S2 = 0.2
UNFREEZE_DINO_S2 = 1 # Unfreeze only the last block

# --- RAG & Inference ---
CONFIDENCE_THRESHOLD = 0.6 # Threshold to trigger RAG
NUM_INFERENCE_SAMPLES = 5 # Number of validation samples to demonstrate RAG

# --- Class Mappings ---
CLASS_MAPPINGS = {
    'Acne and Rosacea Photos': 'acne',
    'Psoriasis pictures Lichen Planus and related diseases': 'psoriasis',
    'Eczema Photos': 'eczema',
    'Herpes HPV and other STDs Photos': 'stds',
    'Tinea Ringworm Candidiasis and other Fungal Infections': 'fungal',
    'Actinic Keratosis Basal Cell Carcinoma and other Malignant Lesions': 'bcc',
    'Seborrheic Keratoses and other Benign Tumors': 'seborrheic_keratosis'
}
FINAL_CLASSES = list(CLASS_MAPPINGS.values())
```

```
CLASS_TO_IDX = {cls: idx for idx, cls in enumerate(FINAL_CLASSES)}
IDX_TO_CLASS = {idx: cls for cls, idx in CLASS_TO_IDX.items()}
```

```
# ================= KNOWLEDGE BASE (for RAG) ==================
class KnowledgeBase:
    def __init__(self, kb_path, tokenizer, text_encoder):
        print("Initializing Knowledge Base...")
        with open(kb_path, 'r') as f:
            kb_data = json.load(f)
        self.entries = kb_data['dermatology_knowledge_base']
        self.tokenizer = tokenizer
        self.text_encoder = text_encoder.to(DEVICE)
        self.text_encoder.eval() # Ensure encoder is in eval mode
        self.embeddings = self._index_knowledge_base()
        print(f"Knowledge Base indexed with {len(self.embeddings)} entries.")

    def _index_knowledge_base(self):
        """Encodes all knowledge base entries into vector embeddings."""
        embeddings = []
        for entry in tqdm(self.entries, desc="Indexing KB"):
            # Use a descriptive combination of fields for embedding
            description = f"{entry['disease']}: {entry['lesion_morphology']['key_distinctions']}"
            inputs = self.tokenizer(description, return_tensors="pt", padding="max_length", truncation=True, max_length=128)
            inputs = {k: v.to(DEVICE) for k, v in inputs.items()}
            with torch.no_grad():
                embedding = self.text_encoder(**inputs).last_hidden_state[:, 0, :].cpu().numpy()
            embeddings.append(embedding)
        return np.vstack(embeddings)

    def retrieve(self, description, top_k=3):
        """Retrieves top_k most similar entries from the KB."""
        inputs = self.tokenizer(description, return_tensors="pt", padding="max_length", truncation=True, max_length=128)
        inputs = {k: v.to(DEVICE) for k, v in inputs.items()}
        with torch.no_grad():
            query_embedding = self.text_encoder(**inputs).last_hidden_state[:, 0, :].cpu().numpy()

        similarities = cosine_similarity(query_embedding, self.embeddings)[0]
        top_indices = np.argsort(similarities)[-top_k:][::-1]
        return [(self.entries[i], similarities[i]) for i in top_indices]

    def get_differential_questions(self, description):
        """Gets unique questions from retrieved entries to differentiate diagnoses."""
        retrieved = self.retrieve(description)
        questions = []
        for entry, _ in retrieved:
            questions.extend(q["question"] for q in entry.get("confounder_questions", []))
        return list(set(questions)) # Return unique questions

    def refine_prediction(self, description, answers):
        """Refines prediction based on answers to confounder questions."""
        retrieved = self.retrieve(description)
        scores = {entry['disease']: sim for entry, sim in retrieved}

        for entry, _ in retrieved:
            for q in entry.get("confounder_questions", []):
                question_text = q["question"]
                if question_text in answers:
                    user_answer = answers[question_text].lower()
                    if user_answer == "yes" and q.get("yes_supports") in scores:
                        scores[q["yes_supports"]] *= 1.5 # Boost score
                    elif user_answer == "no" and q.get("no_supports") in scores:
                        scores[q["no_supports"]] *= 1.5 # Boost score

        if not scores: return None, {}
        refined_disease = max(scores, key=scores.get)
        return refined_disease, scores
```

```
# ================= DATASETS ==================
class VisionDataset(Dataset):
    """Stage-1: Vision-only dataset for images WITHOUT text descriptions."""
    def __init__(self, img_dir, json_path_to_exclude, transform):
        self.transform = transform
        self.samples = []

        with open(json_path_to_exclude, 'r') as f:
            images_with_text = set(json.load(f).keys())
        print(f"Excluding {len(images_with_text)} images that have text descriptions.")

        for class_folder, class_name in CLASS_MAPPINGS.items():
            label = CLASS_TO_IDX[class_name]
            class_path = os.path.join(img_dir, class_folder)
            if os.path.isdir(class_path):
                for img_file in os.listdir(class_path):
                    if img_file.lower().endswith(('.jpg', '.jpeg', '.png')):
                        relative_path = os.path.join(class_folder, img_file)
                        if relative_path not in images_with_text:
                            full_img_path = os.path.join(class_path, img_file)
                            self.samples.append((full_img_path, label))
        print(f"Found {len(self.samples)} images for Stage-1 vision pre-training.")

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

    def __getitem__(self, idx):
        path, label = self.samples[idx]
        image = Image.open(path).convert("RGB")
        return self.transform(image), label


class DermDataset(Dataset):
    """Stage-2: Multimodal dataset for images WITH text descriptions."""
    def __init__(self, img_dir, json_path, tokenizer, transform):
        self.transform = transform
        self.tokenizer = tokenizer
        with open(json_path, 'r') as f:
            self.data = json.load(f)

        self.samples = []
        for rel_path, meta in self.data.items():
            class_name = meta.get("class")
            text_desc = meta.get("description", "")
            if class_name in CLASS_TO_IDX:
                label = CLASS_TO_IDX[class_name]
                full_path = os.path.join(img_dir, rel_path)
                if os.path.exists(full_path):
                    self.samples.append((full_path, text_desc, label, rel_path))
        print(f"Found {len(self.samples)} multimodal samples for Stage-2 training.")

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

    def __getitem__(self, idx):
        img_path, text, label, rel_path = self.samples[idx]
        image = self.transform(Image.open(img_path).convert("RGB"))
        tokens = self.tokenizer(
            text, padding="max_length", truncation=True,
            max_length=128, return_tensors="pt"
        )
        tokens = {k: v.squeeze(0) for k, v in tokens.items()}
        return image, tokens, label, text, rel_path


def collate_fn(batch):
    """Custom collate function to handle multimodal batches."""
    batch = [b for b in batch if b is not None]
    if not batch: return None, None, None, None, None
    images, texts, labels, descriptions, paths = zip(*batch)
    images = torch.stack(images)
    labels = torch.tensor(labels)
    batched_texts = {k: torch.stack([d[k] for d in texts]) for k in texts[0]}
    return images, batched_texts, labels, descriptions, paths
```

```
# ================== MODELS ===================
class DinoVisionClassifier(nn.Module):
    """Stage-1 Vision-only Model."""
    def __init__(self, num_classes=len(FINAL_CLASSES), unfreeze_blocks=UNFREEZE_DINO_S1):
        super().__init__()
        self.vision_encoder = torch.hub.load('facebookresearch/dinov2', 'dinov2_vits14', trust_repo=True)
        # Freeze all parameters initially
        for param in self.vision_encoder.parameters():
            param.requires_grad = False
        # Unfreeze the last N blocks for fine-tuning
        if unfreeze_blocks > 0:
            for block in self.vision_encoder.blocks[-unfreeze_blocks:]:
                for param in block.parameters():
                    param.requires_grad = True
        self.classifier_head = nn.Linear(384, num_classes)

    def forward(self, x):
        features = self.vision_encoder(x)
        return self.classifier_head(features)
```

```python
class MAGEVPro(nn.Module):
    """Stage-2 Multimodal Model with FiLM Fusion and LoRA."""
    def __init__(self, num_classes=len(FINAL_CLASSES), dino_checkpoint_path=None):
        super().__init__()
        # 1. Initialize Vision Encoder from Stage 1
        dino_model = DinoVisionClassifier(num_classes=num_classes)
        if dino_checkpoint_path and os.path.exists(dino_checkpoint_path):
            print(f"Loading Stage-1 vision weights from {dino_checkpoint_path}")
            dino_model.load_state_dict(torch.load(dino_checkpoint_path, map_location=DEVICE))
        self.vision_encoder = dino_model.vision_encoder

        # Freeze all vision parameters initially
        for param in self.vision_encoder.parameters():
            param.requires_grad = False
        # Unfreeze a smaller number of blocks for Stage 2
        if UNFREEZE_DINO_S2 > 0:
            for block in self.vision_encoder.blocks[-UNFREEZE_DINO_S2:]:
                for param in block.parameters():
                    param.requires_grad = True

        # 2. Initialize Text Encoder with LoRA
        self.text_encoder = BertModel.from_pretrained("microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract")
        lora_config = LoraConfig(r=16, lora_alpha=32, target_modules=["query", "value"], lora_dropout=0.1, bias="none")
        self.text_encoder = get_peft_model(self.text_encoder, lora_config)
        self.text_encoder.print_trainable_parameters()

        # 3. Fusion and Classifier Layers
        self.film_gamma = nn.Linear(768, 384)
        self.film_beta = nn.Linear(768, 384)
        self.mlp = nn.Sequential(
            nn.ReLU(), nn.Dropout(0.5),
            nn.Linear(384, 256),
            nn.ReLU(), nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )

    def forward(self, image, text_inputs, return_attention=False):
        image = image.to(DEVICE)
        text_inputs = {k: v.to(DEVICE) for k, v in text_inputs.items()}

        vision_feat = self.vision_encoder(image)
        text_feat = self.text_encoder(**text_inputs).last_hidden_state[:, 0, :] # Get [CLS] token

        gamma = self.film_gamma(text_feat)
        beta = self.film_beta(text_feat)
        fused_features = gamma * vision_feat + beta

        logits = self.mlp(fused_features)

        if return_attention:
            # Use the magnitude of gamma as a proxy for attention
            attention_scores = torch.abs(gamma).mean(dim=0).cpu().detach().numpy()
            return logits, attention_scores

        return logits
```

```python
# ================== PLOTTING & UTILS ===================
def plot_conf_matrix(y_true, y_pred, save_path="confusion_matrix.png"):
    cm = confusion_matrix(y_true, y_pred, labels=np.arange(len(FINAL_CLASSES)))
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=FINAL_CLASSES, yticklabels=FINAL_CLASSES)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title("Confusion Matrix")
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()
    print(f"Confusion matrix saved to {save_path}")

def plot_training_curves(train_losses, val_losses, train_accs, val_accs, save_path="training_curves.png"):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.title("Loss vs. Epochs")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_accs, label='Train Accuracy')
    plt.plot(val_accs, label='Val Accuracy')
    plt.title("Accuracy vs. Epochs")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()
    print(f"Training curves saved to {save_path}")

def plot_attention_map(attention_scores, save_path="attention_map.png"):
    plt.figure(figsize=(10, 2))
    sns.heatmap([attention_scores], cmap="viridis", cbar=True)
    plt.title("Attention Map (FiLM Gamma Weights)")
    plt.xlabel("Vision Feature Dimension")
    plt.yticks([])
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()
```

```
# ================== TRAINING LOOPS ===================
def train_vision_model(model, train_loader, val_loader, criterion, optimizer, epochs, patience, ckpt_path):
    best_val_acc, patience_ctr = 0.0, 0
    history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

    for epoch in range(epochs):
        model.train()
        total_loss, y_true, y_pred = 0, [], []
        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs} [Train]"):
            images, labels = images.to(DEVICE), labels.to(DEVICE)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(outputs.argmax(1).cpu().numpy())

        train_loss = total_loss / len(train_loader)
        train_acc = accuracy_score(y_true, y_pred)
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)

        model.eval()
        total_loss, y_true, y_pred = 0, [], []
        with torch.no_grad():
            for images, labels in tqdm(val_loader, desc=f"Epoch {epoch+1}/{epochs} [Val]"):
                images, labels = images.to(DEVICE), labels.to(DEVICE)
                outputs = model(images)
                loss = criterion(outputs, labels)
                total_loss += loss.item()
                y_true.extend(labels.cpu().numpy())
                y_pred.extend(outputs.argmax(1).cpu().numpy())

        val_loss = total_loss / len(val_loader)
        val_acc = accuracy_score(y_true, y_pred)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)

        print(f"Epoch {epoch+1}: Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} | Val Loss: {val_loss:.4f}, Val Acc:

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_ctr = 0
            torch.save(model.state_dict(), ckpt_path)
            print(f"✅ Best model saved with Val Acc: {best_val_acc:.4f}")
        else:
            patience_ctr += 1
            if patience_ctr >= patience:
                print(f"⏹ Early stopping triggered after {epoch+1} epochs.")
                break
    return history

def train_multimodal_model(model, train_loader, val_loader, criterion, optimizer, epochs, patience, ckpt_path):
    best_val_acc, patience_ctr = 0.0, 0
    history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

    for epoch in range(epochs):
        model.train()
        total_loss, y_true, y_pred = 0, [], []
        for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs} [Train]"):
            images, texts, labels, _, _ = batch
            if images is None: continue

            images, labels = images.to(DEVICE), labels.to(DEVICE)
            optimizer.zero_grad()
            outputs = model(images, texts)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(outputs.argmax(1).cpu().numpy())

        train_loss = total_loss / len(train_loader)
        train_acc = accuracy_score(y_true, y_pred)
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)

        model.eval()
        total_loss, y_true, y_pred = 0, [], []
        with torch.no_grad():
            for batch in tqdm(val_loader, desc=f"Epoch {epoch+1}/{epochs} [Val]"):
                images, texts, labels, _, _ = batch
                if images is None: continue
                images, labels = images.to(DEVICE), labels.to(DEVICE)
                outputs = model(images, texts)
                loss = criterion(outputs, labels)
                total_loss += loss.item()
                y_true.extend(labels.cpu().numpy())
                y_pred.extend(outputs.argmax(1).cpu().numpy())

        val_loss = total_loss / len(val_loader)
        val_acc = accuracy_score(y_true, y_pred)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)

        print(f"Epoch {epoch+1}: Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} | Val Loss: {val_loss:.4f}, Val Acc:

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_ctr = 0
            torch.save(model.state_dict(), ckpt_path)
            print(f"✅ Best model saved with Val Acc: {best_val_acc:.4f}")
            plot_conf_matrix(y_true, y_pred, save_path=f"best_val_conf_matrix_S2.png")
        else:
            patience_ctr += 1
            if patience_ctr >= patience:
                print(f"⏹ Early stopping triggered after {epoch+1} epochs.")
                break
    return history
```

```
# ================== STAGE RUNNERS ===================
def run_stage1():
    print("--- Starting Stage 1: Vision Pre-training ---")
    train_tf = transforms.Compose([
        transforms.Resize((256, 256)), transforms.RandomCrop((224, 224)),
        transforms.RandomHorizontalFlip(), transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
        transforms.ToTensor(), transforms.Normalize([0.5]*3, [0.5]*3)
    ])
    val_tf = transforms.Compose([
        transforms.Resize((224, 224)), transforms.ToTensor(),
        transforms.Normalize([0.5]*3, [0.5]*3)
    ])

    full_dataset = VisionDataset(IMG_DIR, TEXT_JSON, transform=train_tf)
    full_dataset.transform = val_tf # Temporarily switch to val transforms for splitting

    labels = [s[1] for s in full_dataset.samples]
    sss = StratifiedShuffleSplit(n_splits=1, test_size=TEST_SIZE_S1, random_state=42)
    train_idx, val_idx = next(sss.split(np.zeros(len(labels)), labels))

    train_subset = Subset(full_dataset, train_idx)
    # Re-assign the training transforms back to the train_subset's underlying dataset
    train_subset.dataset.transform = train_tf
    val_subset = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=BATCH_SIZE_S1, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=BATCH_SIZE_S1, shuffle=False)

    train_labels = [labels[i] for i in train_idx]
```

```
        class_weights = compute_class_weight('balanced', classes=np.arange(len(FINAL_CLASSES)), y=train_labels)
        criterion = nn.CrossEntropyLoss(weight=torch.tensor(class_weights, dtype=torch.float).to(DEVICE))

        model = DinoVisionClassifier().to(DEVICE)
        optimizer = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()),
                                      lr=LEARNING_RATE_S1, weight_decay=WEIGHT_DECAY_S1)

        history = train_vision_model(model, train_loader, val_loader, criterion, optimizer,
                                     EPOCHS_S1, PATIENCE_S1, DINO_CHECKPOINT_PATH)

        # Use the new path variable here
        plot_training_curves(history['train_loss'], history['val_loss'], history['train_acc'], history['val_acc'], S1_CURVES_PATH)
        print("--- Stage 1 Complete ---")

    def run_stage2():
        print("\n--- Starting Stage 2: Multimodal Fine-tuning ---")
        tokenizer = AutoTokenizer.from_pretrained("microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract")

        # Load text metadata
        with open(TEXT_JSON, "r") as f:
            full_text_data = json.load(f)
        with open(TEST_TEXT_JSON, "r") as f:
            test_text_data = json.load(f)

        # Exclude test keys from full training JSON
        test_keys = set(test_text_data.keys())
        filtered_text_data = {k: v for k, v in full_text_data.items() if k not in test_keys}

        # Save a temporary filtered JSON (optional, for debugging)
        filtered_json_path = "/kaggle/working/train_text_filtered.json"
        with open(filtered_json_path, "w") as f:
            json.dump(filtered_text_data, f)

        # Define transforms
        train_tf = transforms.Compose([
            transforms.Resize((256, 256)), transforms.RandomCrop((224, 224)),
            transforms.RandomHorizontalFlip(), transforms.RandomVerticalFlip(),
            transforms.RandomRotation(20), transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3),
            transforms.ToTensor(), transforms.Normalize([0.5]*3, [0.5]*3)
        ])
        val_tf = transforms.Compose([
            transforms.Resize((224, 224)), transforms.ToTensor(),
            transforms.Normalize([0.5]*3, [0.5]*3)
        ])

        # Use filtered JSON (train only, no test leakage!)
        train_ds = DermDataset(IMG_DIR, filtered_json_path, tokenizer, train_tf)
        val_ds   = DermDataset(IMG_DIR, filtered_json_path, tokenizer, val_tf)

        labels = [s[2] for s in train_ds.samples]
        sss = StratifiedShuffleSplit(n_splits=1, test_size=TEST_SIZE_S2, random_state=42)
        train_idx, val_idx = next(sss.split(np.zeros(len(labels)), labels))

        train_loader = DataLoader(Subset(train_ds, train_idx), batch_size=BATCH_SIZE_S2, shuffle=True, collate_fn=collate_fn)
        val_loader   = DataLoader(Subset(val_ds, val_idx), batch_size=BATCH_SIZE_S2, shuffle=False, collate_fn=collate_fn)

        train_labels = [labels[i] for i in train_idx]
        class_weights = compute_class_weight('balanced', classes=np.arange(len(FINAL_CLASSES)), y=train_labels)
        criterion = nn.CrossEntropyLoss(weight=torch.tensor(class_weights, dtype=torch.float).to(DEVICE))

        model = MAGEVPro(dino_checkpoint_path=DINO_CHECKPOINT_PATH).to(DEVICE)

        # Differential learning rates
        vision_params = [p for n, p in model.vision_encoder.named_parameters() if p.requires_grad]
        text_params = [p for n, p in model.text_encoder.named_parameters() if p.requires_grad]
        head_params = list(model.film_gamma.parameters()) + list(model.film_beta.parameters()) + list(model.mlp.parameters())

        optimizer = torch.optim.AdamW([
            {'params': vision_params, 'lr': VISION_LR_S2},
            {'params': text_params, 'lr': LEARNING_RATE_S2},
            {'params': head_params, 'lr': LEARNING_RATE_S2}
        ], weight_decay=WEIGHT_DECAY_S2)

        history = train_multimodal_model(model, train_loader, val_loader, criterion, optimizer,
                                         EPOCHS_S2, PATIENCE_S2, MAGEVPRO_CHECKPOINT_PATH)

        # Use the new path variable here
        plot_training_curves(history['train_loss'], history['val_loss'], history['train_acc'], history['val_acc'], S2_CURVES_PATH)
        print("--- Stage 2 Complete ---")
```

```
    import torch.nn.functional as F
    import random
```

```
def infer_with_rag(model, kb, image_tensor, description, tokenizer, rel_path):
    """
    Performs inference, triggering a RAG-based refinement if confidence is low.
    """
    # --- 1. Initial Model Prediction ---
    model.eval()
    with torch.no_grad():
        # Prepare inputs
        img_input = image_tensor.unsqueeze(0).to(DEVICE)
        tokens = tokenizer(description, padding="max_length", truncation=True, max_length=128, return_tensors="pt")
        text_input = {k: v.to(DEVICE) for k, v in tokens.items()}

        # Get model output
        logits = model(img_input, text_input)
        probabilities = F.softmax(logits, dim=1)
        confidence, pred_idx = torch.max(probabilities, dim=1)

        initial_prediction = IDX_TO_CLASS[pred_idx.item()]
        confidence_score = confidence.item()

    print(f"Initial Prediction: '{initial_prediction}' with confidence: {confidence_score:.2%}")

    final_prediction = initial_prediction

    # --- 2. Check Confidence and Trigger RAG if Needed ---
    if confidence_score < CONFIDENCE_THRESHOLD:
        print(f"Confidence is below {CONFIDENCE_THRESHOLD:.0%}. Triggering RAG refinement... 🤔")

        # --- 3. Retrieve Questions from Knowledge Base ---
        questions = kb.get_differential_questions(description)

        if not questions:
            print("No relevant differential questions found in the KB.")
        else:
            print("\\nAsking clarifying questions:")
            for q in questions:
                print(f"  - {q}")

            # --- 4. Simulate User Answers ---
            # In a real app, you would capture user input here.
            # For this demonstration, we'll simulate the answers.
            answers = {q: random.choice(['yes', 'no']) for q in questions}
            print("\\nSimulated Answers:")
            for q, a in answers.items():
                print(f"  - Q: {q} -> A: {a}")

            # --- 5. Refine Prediction with Answers ---
            refined_disease, scores = kb.refine_prediction(description, answers)
            if refined_disease:
                print(f"\\nKB Refined Prediction: '{refined_disease}'")
                final_prediction = refined_disease
            else:
                print("\\nCould not refine prediction with the given answers.")
    else:
        print("Confidence is high. Sticking with the initial prediction. ✅")

    print(f"\\n--- Final Result for {os.path.basename(rel_path)} ---")
    print(f"==> Final Diagnosis: {final_prediction}")
    print("-" * 20)
```

```
# ================== MAIN EXECUTION ===================
if __name__ == "__main__":
    # --- Run Training Stages ---
    run_stage1()
    run_stage2()

    # --- Setup for RAG Demonstration ---
    print("\n--- Preparing for RAG Demonstration ---")
    tokenizer = AutoTokenizer.from_pretrained("microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract")
    text_encoder_for_kb = BertModel.from_pretrained("microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract")
    kb = KnowledgeBase(KB_JSON, tokenizer, text_encoder_for_kb)

    # Load the best model from Stage 2 for the demo
    final_model = MAGEVPro().to(DEVICE)
    final_model.load_state_dict(torch.load(MAGEVPRO_CHECKPOINT_PATH, map_location=DEVICE))
    print(f"\n✅ Loaded best model from {MAGEVPRO_CHECKPOINT_PATH} for RAG demo.")

    # --- RAG Demonstration on Validation Samples ---
    print(f"\n🔍 Running RAG inference demonstration on {NUM_INFERENCE_SAMPLES} validation samples...")
    val_tf = transforms.Compose([transforms.Resize((224, 224)), transforms.ToTensor(), transforms.Normalize([0.5]*3, [0.5]*3)])
    val_ds_for_demo = DermDataset(IMG_DIR, TEXT_JSON, tokenizer, val_tf)
```

```
        labels = [s[2] for s in val_ds_for_demo.samples]
        sss = StratifiedShuffleSplit(n_splits=1, test_size=TEST_SIZE_S2, random_state=42)
        _, val_idx = next(sss.split(np.zeros(len(labels)), labels))
        val_subset = Subset(val_ds_for_demo, val_idx)

        for i in range(min(NUM_INFERENCE_SAMPLES, len(val_subset))):
            image_tensor, _, true_label_idx, description, rel_path = val_subset[i]

            print(f"\n--- Sample {i+1} ---")
            print(f"Image Path: {rel_path}")
            print(f"True Label: {IDX_TO_CLASS[true_label_idx]}")
            print(f"Description: {description}")

            infer_with_rag(final_model, kb, image_tensor, description, tokenizer, rel_path)
```

```
--- Starting Stage 1: Vision Pre-training ---
Excluding 2100 images that have text descriptions.
Found 5605 images for Stage-1 vision pre-training.
Downloading: "https://github.com/facebookresearch/dinov2/zipball/main" to /root/.cache/torch/hub/main.zip
/root/.cache/torch/hub/facebookresearch_dinov2_main/dinov2/layers/swiglu_ffn.py:51: UserWarning: xFormers is not available (Swi
  warnings.warn("xFormers is not available (SwiGLU)")
/root/.cache/torch/hub/facebookresearch_dinov2_main/dinov2/layers/attention.py:33: UserWarning: xFormers is not available (Atte
  warnings.warn("xFormers is not available (Attention)")
/root/.cache/torch/hub/facebookresearch_dinov2_main/dinov2/layers/block.py:40: UserWarning: xFormers is not available (Block)
  warnings.warn("xFormers is not available (Block)")
Downloading: "https://dl.fbaipublicfiles.com/dinov2/dinov2_vits14/dinov2_vits14_pretrain.pth" to /root/.cache/torch/hub/checkp
100%|██████████| 84.2M/84.2M [00:00<00:00, 215MB/s]
Epoch 1/20 [Train]: 100%|██████████| 141/141 [01:31<00:00,  1.54it/s]
Epoch 1/20 [Val]: 100%|██████████| 36/36 [00:20<00:00,  1.71it/s]
Epoch 1: Train Loss: 1.6409, Train Acc: 0.4110 | Val Loss: 1.3096, Val Acc: 0.4469
✅ Best model saved with Val Acc: 0.4469
Epoch 2/20 [Train]: 100%|██████████| 141/141 [01:04<00:00,  2.19it/s]
Epoch 2/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.51it/s]
Epoch 2: Train Loss: 1.1719, Train Acc: 0.5408 | Val Loss: 1.1351, Val Acc: 0.5433
✅ Best model saved with Val Acc: 0.5433
Epoch 3/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.09it/s]
Epoch 3/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.46it/s]
Epoch 3: Train Loss: 0.9754, Train Acc: 0.6169 | Val Loss: 1.1493, Val Acc: 0.5468
✅ Best model saved with Val Acc: 0.5468
Epoch 4/20 [Train]: 100%|██████████| 141/141 [01:06<00:00,  2.12it/s]
Epoch 4/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.45it/s]
Epoch 4: Train Loss: 0.7999, Train Acc: 0.6804 | Val Loss: 1.0658, Val Acc: 0.5959
✅ Best model saved with Val Acc: 0.5959
Epoch 5/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.10it/s]
Epoch 5/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.47it/s]
Epoch 5: Train Loss: 0.6882, Train Acc: 0.7212 | Val Loss: 1.1990, Val Acc: 0.6450
✅ Best model saved with Val Acc: 0.6450
Epoch 6/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.10it/s]
Epoch 6/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.47it/s]
Epoch 6: Train Loss: 0.5583, Train Acc: 0.7743 | Val Loss: 1.0493, Val Acc: 0.6405
Epoch 7/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.10it/s]
Epoch 7/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.47it/s]
Epoch 7: Train Loss: 0.4557, Train Acc: 0.8158 | Val Loss: 1.1289, Val Acc: 0.6878
✅ Best model saved with Val Acc: 0.6878
Epoch 8/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.09it/s]
Epoch 8/20 [Val]: 100%|██████████| 36/36 [00:15<00:00,  2.38it/s]
Epoch 8: Train Loss: 0.3763, Train Acc: 0.8483 | Val Loss: 1.3715, Val Acc: 0.6762
Epoch 9/20 [Train]: 100%|██████████| 141/141 [01:10<00:00,  2.01it/s]
Epoch 9/20 [Val]: 100%|██████████| 36/36 [00:15<00:00,  2.32it/s]
Epoch 9: Train Loss: 0.2955, Train Acc: 0.8805 | Val Loss: 1.1992, Val Acc: 0.6851
Epoch 10/20 [Train]: 100%|██████████| 141/141 [01:08<00:00,  2.05it/s]
Epoch 10/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.43it/s]
Epoch 10: Train Loss: 0.2613, Train Acc: 0.8979 | Val Loss: 1.3246, Val Acc: 0.6922
✅ Best model saved with Val Acc: 0.6922
Epoch 11/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.08it/s]
Epoch 11/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.44it/s]
Epoch 11: Train Loss: 0.2077, Train Acc: 0.9159 | Val Loss: 1.3624, Val Acc: 0.7190
✅ Best model saved with Val Acc: 0.7190
Epoch 12/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.08it/s]
Epoch 12/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.44it/s]
Epoch 12: Train Loss: 0.1816, Train Acc: 0.9289 | Val Loss: 1.4955, Val Acc: 0.6271
Epoch 13/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.08it/s]
Epoch 13/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.45it/s]
Epoch 13: Train Loss: 0.2742, Train Acc: 0.8967 | Val Loss: 1.2937, Val Acc: 0.7021
Epoch 14/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.08it/s]
Epoch 14/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.41it/s]
Epoch 14: Train Loss: 0.1685, Train Acc: 0.9367 | Val Loss: 1.2807, Val Acc: 0.6967
Epoch 15/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.09it/s]
Epoch 15/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.46it/s]
Epoch 15: Train Loss: 0.1627, Train Acc: 0.9405 | Val Loss: 1.5468, Val Acc: 0.7270
✅ Best model saved with Val Acc: 0.7270
Epoch 16/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.10it/s]
Epoch 16/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.47it/s]
Epoch 16: Train Loss: 0.1191, Train Acc: 0.9565 | Val Loss: 1.2893, Val Acc: 0.7065
Epoch 17/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.10it/s]
Epoch 17/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.45it/s]
Epoch 17: Train Loss: 0.1408, Train Acc: 0.9529 | Val Loss: 1.5185, Val Acc: 0.6860
Epoch 18/20 [Train]: 100%|██████████| 141/141 [01:07<00:00,  2.10it/s]
Epoch 18/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.44it/s]
Epoch 18: Train Loss: 0.1134, Train Acc: 0.9538 | Val Loss: 1.4841, Val Acc: 0.7288
✅ Best model saved with Val Acc: 0.7288
Epoch 19/20 [Train]: 100%|██████████| 141/141 [01:06<00:00,  2.11it/s]
Epoch 19/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.47it/s]
Epoch 19: Train Loss: 0.0983, Train Acc: 0.9612 | Val Loss: 1.4545, Val Acc: 0.6940
Epoch 20/20 [Train]: 100%|██████████| 141/141 [01:06<00:00,  2.11it/s]
Epoch 20/20 [Val]: 100%|██████████| 36/36 [00:14<00:00,  2.45it/s]
Epoch 20: Train Loss: 0.1073, Train Acc: 0.9556 | Val Loss: 1.3999, Val Acc: 0.7154
Training curves saved to /kaggle/working/training_curves_S1.png
--- Stage 1 Complete ---

--- Starting Stage 2: Multimodal Fine-tuning ---
tokenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
config.json:   0%|          | 0.00/385 [00:00<?, ?B/s]
vocab.txt: 0.00B [00:00, ?B/s]
Found 2030 multimodal samples for Stage-2 training.
Found 2030 multimodal samples for Stage-2 training.
Using cache found in /root/.cache/torch/hub/facebookresearch_dinov2_main
Loading Stage-1 vision weights from /kaggle/working/dino_finetuned.pth
pytorch_model.bin:   0%|          | 0.00/440M [00:00<?, ?B/s]
trainable params: 589,824 || all params: 110,072,064 || trainable%: 0.5359
Epoch 1/30 [Train]: 100%|██████████| 102/102 [01:00<00:00,  1.68it/s]
Epoch 1/30 [Val]: 100%|██████████| 26/26 [00:10<00:00,  2.60it/s]
Epoch 1: Train Loss: 1.7814, Train Acc: 0.3079 | Val Loss: 1.4390, Val Acc: 0.6700
✅ Best model saved with Val Acc: 0.6700
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 2/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 2/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.18it/s]
Epoch 2: Train Loss: 1.3758, Train Acc: 0.5425 | Val Loss: 1.0354, Val Acc: 0.6946
✅ Best model saved with Val Acc: 0.6946
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 3/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 3/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.20it/s]
Epoch 3: Train Loss: 1.0552, Train Acc: 0.6601 | Val Loss: 0.8108, Val Acc: 0.7611
✅ Best model saved with Val Acc: 0.7611
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 4/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 4/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.23it/s]
Epoch 4: Train Loss: 0.8286, Train Acc: 0.7395 | Val Loss: 0.5671, Val Acc: 0.8473
✅ Best model saved with Val Acc: 0.8473
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 5/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.94it/s]
Epoch 5/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.12it/s]
Epoch 5: Train Loss: 0.6195, Train Acc: 0.8116 | Val Loss: 0.4231, Val Acc: 0.8744
✅ Best model saved with Val Acc: 0.8744
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 6/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 6/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.22it/s]
Epoch 6: Train Loss: 0.4896, Train Acc: 0.8461 | Val Loss: 0.2998, Val Acc: 0.9163
✅ Best model saved with Val Acc: 0.9163
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 7/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 7/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.20it/s]
Epoch 7: Train Loss: 0.3501, Train Acc: 0.8996 | Val Loss: 0.2204, Val Acc: 0.9458
✅ Best model saved with Val Acc: 0.9458
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 8/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.96it/s]
Epoch 8/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.18it/s]
Epoch 8: Train Loss: 0.2754, Train Acc: 0.9249 | Val Loss: 0.1794, Val Acc: 0.9409
Epoch 9/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 9/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.17it/s]
Epoch 9: Train Loss: 0.2250, Train Acc: 0.9347 | Val Loss: 0.1387, Val Acc: 0.9581
✅ Best model saved with Val Acc: 0.9581
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 10/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.94it/s]
Epoch 10/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 10: Train Loss: 0.1709, Train Acc: 0.9544 | Val Loss: 0.1186, Val Acc: 0.9631
✅ Best model saved with Val Acc: 0.9631
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 11/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.94it/s]
Epoch 11/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.21it/s]
Epoch 11: Train Loss: 0.1721, Train Acc: 0.9514 | Val Loss: 0.1070, Val Acc: 0.9704
✅ Best model saved with Val Acc: 0.9704
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 12/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 12/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.24it/s]
Epoch 12: Train Loss: 0.1266, Train Acc: 0.9674 | Val Loss: 0.0931, Val Acc: 0.9729
✅ Best model saved with Val Acc: 0.9729
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 13/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 13/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.22it/s]
Epoch 13: Train Loss: 0.1038, Train Acc: 0.9717 | Val Loss: 0.0896, Val Acc: 0.9778
✅ Best model saved with Val Acc: 0.9778
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 14/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.96it/s]
Epoch 14/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 14: Train Loss: 0.0857, Train Acc: 0.9766 | Val Loss: 0.0906, Val Acc: 0.9754
Epoch 15/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.96it/s]
Epoch 15/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.16it/s]
Epoch 15: Train Loss: 0.0745, Train Acc: 0.9778 | Val Loss: 0.0883, Val Acc: 0.9803
✅ Best model saved with Val Acc: 0.9803
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 16/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 16/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 16: Train Loss: 0.0721, Train Acc: 0.9815 | Val Loss: 0.0844, Val Acc: 0.9828
✅ Best model saved with Val Acc: 0.9828
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 17/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 17/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.21it/s]
Epoch 17: Train Loss: 0.0618, Train Acc: 0.9834 | Val Loss: 0.0779, Val Acc: 0.9828
Epoch 18/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 18/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 18: Train Loss: 0.0480, Train Acc: 0.9895 | Val Loss: 0.0623, Val Acc: 0.9828
Epoch 19/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 19/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.12it/s]
Epoch 19: Train Loss: 0.0465, Train Acc: 0.9865 | Val Loss: 0.0513, Val Acc: 0.9852
✅ Best model saved with Val Acc: 0.9852
✅ Best model saved with Val Acc: 0.9852
Confusion matrix saved to best_val_conf_matrix_S2.png
Epoch 20/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.95it/s]
Epoch 20/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.14it/s]
Epoch 20: Train Loss: 0.0423, Train Acc: 0.9901 | Val Loss: 0.0715, Val Acc: 0.9803
Epoch 21/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.94it/s]
Epoch 21/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 21: Train Loss: 0.0427, Train Acc: 0.9883 | Val Loss: 0.0872, Val Acc: 0.9778
Epoch 22/30 [Train]: 100%|██████████| 102/102 [00:52<00:00,  1.96it/s]
Epoch 22/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.20it/s]
Epoch 22: Train Loss: 0.0252, Train Acc: 0.9945 | Val Loss: 0.0697, Val Acc: 0.9803
Epoch 23/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.98it/s]
Epoch 23/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 23: Train Loss: 0.0286, Train Acc: 0.9926 | Val Loss: 0.0668, Val Acc: 0.9828
Epoch 24/30 [Train]: 100%|██████████| 102/102 [00:51<00:00,  1.97it/s]
Epoch 24/30 [Val]: 100%|██████████| 26/26 [00:08<00:00,  3.19it/s]
Epoch 24: Train Loss: 0.0335, Train Acc: 0.9920 | Val Loss: 0.0633, Val Acc: 0.9828
⏹️ Early stopping triggered after 24 epochs.
Training curves saved to /kaggle/working/training_curves_S2.png
--- Stage 2 Complete ---

--- Preparing for RAG Demonstration ---
Initializing Knowledge Base...
Indexing KB: 100%|██████████| 7/7 [00:00<00:00, 59.85it/s]
Knowledge Base indexed with 7 entries.
Using cache found in /root/.cache/torch/hub/facebookresearch_dinov2_main
trainable params: 589,824 || all params: 110,072,064 || trainable%: 0.5359

✅ Loaded best model from /kaggle/working/magevpro_best.pth for RAG demo.

🔍 Running RAG inference demonstration on 5 validation samples...
Found 2100 multimodal samples for Stage-2 training.

--- Sample 1 ---
Image Path: Actinic Keratosis Basal Cell Carcinoma and other Malignant Lesions/basal-cell-carcinoma-vulva-1.jpg
True Label: bcc
Description: This is a single, red bump or lump. It has a shiny, see-through quality to its edge, and I can see tiny blood vess
Initial Prediction: 'bcc' with confidence: 100.00%
Confidence is high. Sticking with the initial prediction. ✅
\n--- Final Result for basal-cell-carcinoma-vulva-1.jpg ---
==> Final Diagnosis: bcc
--------------------

--- Sample 2 ---
Image Path: Herpes HPV and other STDs Photos/herpes-type-2-recurrent-42.jpg
True Label: stds
Description: I have a cluster of shallow, open sores on a red, swollen patch of skin in my genital area. They look like blister
Initial Prediction: 'stds' with confidence: 99.97%
Confidence is high. Sticking with the initial prediction. ✅
\n--- Final Result for herpes-type-2-recurrent-42.jpg ---
==> Final Diagnosis: stds
--------------------

--- Sample 3 ---
Image Path: Herpes HPV and other STDs Photos/genital-warts-65.jpg
True Label: stds
Description: I have a single, pinkish, warty bump in my genital area. It feels firm and has a rough surface. The skin around it
Initial Prediction: 'stds' with confidence: 99.99%
Confidence is high. Sticking with the initial prediction. ✅
\n--- Final Result for genital-warts-65.jpg ---
==> Final Diagnosis: stds
--------------------

--- Sample 4 ---
Image Path: Eczema Photos/lichen-simplex-chronicus-184.jpg
True Label: eczema
Description: I have a red rash on the side of my ankle and foot with blurry edges. The skin is dry, flaky, and looks thick and
Initial Prediction: 'eczema' with confidence: 99.98%
Confidence is high. Sticking with the initial prediction. ✅
\n--- Final Result for lichen-simplex-chronicus-184.jpg ---
==> Final Diagnosis: eczema
--------------------

--- Sample 5 ---
Image Path: Tinea Ringworm Candidiasis and other Fungal Infections/erosio-interdigitalis-blastomycetica-37.jpg
True Label: fungal
Description: The skin between my toes is bright red, flaky, and looks wet and raw. It's very irritated from being in a moist ar
Initial Prediction: 'fungal' with confidence: 100.00%
Confidence is high. Sticking with the initial prediction. ✅
\n--- Final Result for erosio-interdigitalis-blastomycetica-37.jpg ---
==> Final Diagnosis: fungal
--------------------

==================================================
--- STARTING FINAL EVALUATION ON HOLD-OUT TEST SET ---
==================================================

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_36/1810583061.py in <cell line: 0>()
     41
```

```
import os
import shutil
import json

# ================== CONFIGURATION ===================
# The source directory where the original training images are located.
SOURCE_BASE_DIR = "/kaggle/input/dermnet/train"

# The JSON file that lists which images to include in the test set.
TEST_TEXT_JSON = "/kaggle/input/test-2-final/test2_final.json"

# The destination directory where the new test set will be created.
TEST_DIR = "/kaggle/working/test_images"

# ================== SCRIPT ===================

def create_test_set_from_json(source_dir, dest_dir, json_path):
    """
    Creates a persistent test dataset by copying files listed in a JSON file.
    """
    # 1. Clean up and create the destination directory for a fresh start.
    if os.path.exists(dest_dir):
        print(f"Directory '{dest_dir}' already exists. Removing it...")
        shutil.rmtree(dest_dir)
    os.makedirs(dest_dir)
    print(f"Successfully created empty directory: {dest_dir}")

    # 2. Load the list of test images from the JSON file.
    try:
        with open(json_path, "r") as f:
            images_to_copy = json.load(f)
    except FileNotFoundError:
        print(f"🛑 Error: The JSON file was not found at '{json_path}'")
        return

    # 3. Iterate, create subfolders, and copy images.
    total_copied = 0
    total_missing = 0
    print(f"\nReading {len(images_to_copy)} entries from JSON and copying files...")

    for relative_path, info in images_to_copy.items():
        class_name = info.get("class")
        if not class_name:
            print(f"  [Skipping] Missing 'class' for entry: {relative_path}")
            continue

        # Create the class-specific subfolder (e.g., /kaggle/working/test_images/acne)
        class_dir = os.path.join(dest_dir, class_name)
        os.makedirs(class_dir, exist_ok=True)

        # Define source and destination paths for the image
        source_path = os.path.join(source_dir, relative_path)
        destination_path = os.path.join(class_dir, os.path.basename(relative_path))

        # Copy the file if it exists
        if os.path.exists(source_path):
            shutil.copy2(source_path, destination_path)
            total_copied += 1
        else:
            print(f"  [Warning] Source file not found and was skipped: {source_path}")
            total_missing += 1

    # 4. Print a final summary.
    print("\n--- Test Set Creation Summary ---")
    print(f"✅ Successfully copied {total_copied} images.")
    if total_missing > 0:
        print(f"⚠️ Could not find {total_missing} source images.")

# --- Execute the function ---
create_test_set_from_json(
    source_dir=SOURCE_BASE_DIR,
    dest_dir=TEST_DIR,
    json_path=TEST_TEXT_JSON
)
```

```
Successfully created empty directory: /kaggle/working/test_images

Reading 70 entries from JSON and copying files...

--- Test Set Creation Summary ---
✅ Successfully copied 70 images.
```