validation-capsule-vision

October 23, 2024

1 Directory

[2]:
```
import os
for dirname, _, filenames in os.walk('/kaggle/input/capsule-vision-2024-models/
↪pytorch/updated/1'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
/kaggle/input/capsule-
vision-2024-models/pytorch/updated/1/SwinTransformer_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ResNeXt_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/WideResNet_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ResNet_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ViT_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/RegNet_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/BEiT_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/TwinsSVT_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/SEResNet50_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/MobileNetV3_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/MNASNet_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/CaiT_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/DeiT_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/DenseNet_best.pth
/kaggle/input/capsule-
vision-2024-models/pytorch/updated/1/EfficientFormer_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/InceptionV3_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ConvNeXt_best.pth
/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/EfficientNet_best.pth
```

2 Imports

[3]:
```
# Imports
import os
import json
import random

from typing import Dict, List, Tuple
```

```
from datetime import datetime
from pathlib import Path
from logging import getLogger, Logger, INFO, StreamHandler, FileHandler,␣
↪Formatter
from tqdm.auto import tqdm

import pandas as pd
import numpy as np
from PIL import Image

import timm

import torch
from torch import nn, optim
from torch.optim.lr_scheduler import CosineAnnealingLR
from torch.utils.data import Dataset, DataLoader
from torch.utils.data.sampler import WeightedRandomSampler
from torch.nn import functional as F

import torchvision
from torchvision import transforms, models

import torchmetrics
print("Libraries Imported Successfuly!\n\n")
```

```
Libraries Imported Successfuly!
```

3 Models

[4]:
```
import torch
import torch.nn as nn
import timm
import warnings

warnings.filterwarnings('ignore')

# 1. ViT (Vision Transformer)
def model_vit(pretrained=True, num_classes=10):
    model = timm.create_model('vit_base_patch16_224', pretrained=pretrained)
    model.head = nn.Linear(model.head.in_features, num_classes)
    return model

# 2. Swin Transformer
def model_swin(pretrained=True, num_classes=10):
```

```python
    model = timm.create_model('swin_base_patch4_window7_224',␣
↪pretrained=pretrained)
    model.head.fc = nn.Linear(model.head.fc.in_features, num_classes)
    return model

# 3. DeiT (Data-efficient Image Transformers)
def model_deit(pretrained=True, num_classes=10):
    model = timm.create_model('deit_base_patch16_224', pretrained=pretrained)
    model.head = nn.Linear(model.head.in_features, num_classes)
    return model

# 4. ConvNeXt
def model_convnext(pretrained=True, num_classes=10):
    model = timm.create_model('convnext_base', pretrained=pretrained)
    model.head.fc = nn.Linear(model.head.fc.in_features, num_classes)
    return model

# 5. EfficientNet
def model_efficientnet(pretrained=True, num_classes=10):
    model = timm.create_model('tf_efficientnetv2_s_in21ft1k',␣
↪pretrained=pretrained)
    model.classifier = nn.Linear(model.classifier.in_features, num_classes)
    return model

# 6. ResNet
def model_resnet(pretrained=True, num_classes=10):
    model = timm.create_model('resnet50', pretrained=pretrained)
    model.fc = nn.Linear(model.fc.in_features, num_classes)
    return model

# 7. MobileNetV3
def model_mobilenetv3(pretrained=True, num_classes=10):
    model = timm.create_model('mobilenetv3_large_100', pretrained=pretrained)
    model.classifier = nn.Linear(model.classifier.in_features, num_classes)
    return model

# 8. RegNet
def model_regnet(pretrained=True, num_classes=10):
    model = timm.create_model('regnetx_032', pretrained=pretrained)
    model.head.fc = nn.Linear(model.head.fc.in_features, num_classes)
    return model

# 9. DenseNet
def model_densenet(pretrained=True, num_classes=10):
    model = timm.create_model('densenet121', pretrained=pretrained)
    model.classifier = nn.Linear(model.classifier.in_features, num_classes)
    return model
```

```python
# 10. Inception v3
def model_inception_v3(pretrained=True, num_classes=10):
    model = timm.create_model('inception_v3', pretrained=pretrained)
    model.fc = nn.Linear(model.fc.in_features, num_classes)
    return model

# 11. ResNeXt
def model_resnext(pretrained=True, num_classes=10):
    model = timm.create_model('resnext50_32x4d', pretrained=pretrained)
    model.fc = nn.Linear(model.fc.in_features, num_classes)
    return model

# 12. Wide ResNet
def model_wide_resnet(pretrained=True, num_classes=10):
    model = timm.create_model('wide_resnet50_2', pretrained=pretrained)
    model.fc = nn.Linear(model.fc.in_features, num_classes)
    return model

# 13. MNASNet
def model_mnasnet(pretrained=True, num_classes=10):
    model = timm.create_model('mnasnet_100', pretrained=pretrained)
    model.classifier = nn.Linear(model.classifier.in_features, num_classes)
    return model

# 14. SEResNet50 (Replaces SqueezeNet)
def model_seresnet50(pretrained=True, num_classes=10):
    model = timm.create_model('seresnet50', pretrained=pretrained)
    model.fc = nn.Linear(model.fc.in_features, num_classes)
    return model

# 15. BEiT (Bidirectional Encoder Representation from Image Transformers)
def model_beit(pretrained=True, num_classes=10):
    model = timm.create_model('beit_base_patch16_224', pretrained=pretrained)
    model.head = nn.Linear(model.head.in_features, num_classes)
    return model

# 16. CaiT (Class-Attention in Image Transformers)
def model_cait(pretrained=True, num_classes=10):
    model = timm.create_model('cait_s24_224', pretrained=pretrained)
    model.head = nn.Linear(model.head.in_features, num_classes)
    return model

# 17. Twins-SVT (Spatially Separable Vision Transformer)
def model_twins_svt(pretrained=True, num_classes=10):
    model = timm.create_model('twins_svt_base', pretrained=pretrained)
    model.head = nn.Linear(model.head.in_features, num_classes)
```

```python
    return model

# 18. EfficientFormer
def model_efficientformer(pretrained=True, num_classes=10):
    model = timm.create_model('efficientformerv2_s0', pretrained=pretrained,␣
↪num_classes=num_classes)

    # Ensure the classifier is set to the correct number of classes
    if hasattr(model, 'head'):
        in_features = model.head.in_features
        model.head = nn.Linear(in_features, num_classes)
    elif hasattr(model, 'classifier'):
        in_features = model.classifier.in_features
        model.classifier = nn.Linear(in_features, num_classes)
    else:
        raise AttributeError("Model doesn't have a 'head' or 'classifier'␣
↪attribute")

    return model

# if __name__ == "__main__":
#     # Test the models with random input
#     input_tensor = torch.randn(1, 3, 224, 224)  # Batch size of 1, 3 color␣
↪channels, 224x224 image size

#     models_to_test = [
#         model_vit, model_swin, model_deit, model_convnext, model_efficientnet,
#         model_resnet, model_mobilenetv3, model_regnet, model_densenet,␣
↪model_inception_v3,
#         model_resnext, model_wide_resnet, model_mnasnet,
#         model_seresnet50,
#         model_beit, model_cait,
#         model_twins_svt, model_pnasnet,
#         model_xcit
#     ]

#     expected_shape = (1, 10)  # Expected output shape

#     for model_func in models_to_test:
#         model = model_func()
#         output = model(input_tensor)
#         if output.shape != expected_shape:
#             print(f"Model {model_func.__name__} failed with output shape:␣
↪{output.shape}")
#             break
#         print(f"{model_func.__name__} Output Shape:", output.shape)
```

[ ]:

```python

```

4 Initial Setup

[5]:

```python
batch_size = 32
num_workers = 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

test_xlsx = 'validation_data.xlsx'
test_root_dir = 'validation'
#data_root_dir = "../capsule-vision-2024/data/Dataset"
data_root_dir = '/kaggle/input/capsule-vision-2024-data/Dataset'

# model_paths = [
#     '../capsule-vision-2024/models/SwinTransformer_best.pth',
#     '../capsule-vision-2024/models/ResNeXt_best.pth',
#     '../capsule-vision-2024/models/WideResNet_best.pth',
#     '../capsule-vision-2024/models/ResNet_best.pth',
#     '../capsule-vision-2024/models/ViT_best.pth',
#     '../capsule-vision-2024/models/RegNet_best.pth',
#     '../capsule-vision-2024/models/BEiT_best.pth',
#     '../capsule-vision-2024/models/TwinsSVT_best.pth',
#     '../capsule-vision-2024/models/SEResNet50_best.pth',
#     '../capsule-vision-2024/models/MobileNetV3_best.pth',
#     '../capsule-vision-2024/models/MNASNet_best.pth',
#     '../capsule-vision-2024/models/CaiT_best.pth',
#     '../capsule-vision-2024/models/DeiT_best.pth',
#     '../capsule-vision-2024/models/DenseNet_best.pth',
#     '../capsule-vision-2024/models/EfficientFormer_best.pth',
#     '../capsule-vision-2024/models/InceptionV3_best.pth',
#     '../capsule-vision-2024/models/ConvNeXt_best.pth',
#     '../capsule-vision-2024/models/EfficientNet_best.pth'
# ]

model_paths = [
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/
↪SwinTransformer_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ResNeXt_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/WideResNet_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ResNet_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ViT_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/RegNet_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/BEiT_best.pth',
```

```
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/TwinsSVT_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/SEResNet50_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/MobileNetV3_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/MNASNet_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/CaiT_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/DeiT_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/DenseNet_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/
↪EfficientFormer_best.pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/InceptionV3_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/ConvNeXt_best.
↪pth',
    '/kaggle/input/capsule-vision-2024-models/pytorch/updated/1/
↪EfficientNet_best.pth'
]

# metrics_report_dir = "../capsule-vision-2024/reports/metrics_report.json"
metrics_report_dir = "/kaggle/working/metrics_report.json"

model_classes = [
    model_swin,          # Swin Transformer
    model_resnext,       # ResNeXt
    model_wide_resnet,   # Wide ResNet
    model_resnet,        # ResNet
    model_vit,           # Vision Transformer
    model_regnet,        # RegNet
    model_beit,          # BEiT
    model_twins_svt,     # Twins-SVT
    model_seresnet50,    # SEResNet50
    model_mobilenetv3,   # MobileNetV3
    model_mnasnet,       # MNASNet
    model_cait,          # CaiT
    model_deit,          # DeiT
    model_densenet,      # DenseNet
    model_efficientformer, # EfficientFormer
    model_inception_v3,  # Inception v3
    model_convnext,      # ConvNeXt
    model_efficientnet   # EfficientNet
]
```

5 Data Setup

[6]:
```
# New Dataset class to include image paths
class VCEDatasetWithPaths(torch.utils.data.Dataset):
    def __init__(self, xlsx_file, root_dir, train_or_test: str, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.xlsx_file_path = os.path.join(self.root_dir, train_or_test,␣
↪xlsx_file)
        self.annotations = pd.read_excel(io=self.xlsx_file_path, sheet_name=0)
        self.class_columns = self.annotations.columns[2:]  # Assuming class␣
↪columns start from the 3rd column
        self.num_classes = len(self.class_columns)

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

    def __getitem__(self, index):
        # Get image path
        img_path = os.path.join(self.root_dir, self.annotations.iloc[index, 0].
↪replace("\\", "/"))
        only_image_path = self.annotations.iloc[index, 0]
        # Load the image and ensure it's in RGB format
        image = Image.open(img_path).convert('RGB')

        # Get the target label, assuming one-hot encoding in the Excel
        target = self.annotations.iloc[index, 2:].values
        y_label = torch.tensor(target.argmax(), dtype=torch.long)

        # Apply transformation, if provided
        if self.transform:
            image = self.transform(image)

        # Return image, label, and the image path
        return image, y_label, only_image_path

# Preprocess data using VCEDatasetWithPaths
def preprocess_data_with_paths(xlsx_file, data_dir):
    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])
    ])

    val_dataset = VCEDatasetWithPaths(
        xlsx_file=xlsx_file,
```

```
        root_dir=data_dir,
        train_or_test='validation',
        transform=transform
    )

    dataloader = DataLoader(
        dataset=val_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers
    )

    return dataloader, val_dataset
```

6 Metrics

[7]:
```
import torch
import torchmetrics
from torch import nn
import json
from typing import List


class FocalLoss(nn.Module):
    def __init__(self, alpha=1, gamma=2, reduction='mean'):
        super(FocalLoss, self).__init__()
        self.alpha = alpha
        self.gamma = gamma
        self.reduction = reduction

    def forward(self, inputs, targets):
        CE_loss = nn.CrossEntropyLoss(reduction='none')(inputs, targets)
        p_t = torch.exp(-CE_loss)
        loss = self.alpha * (1 - p_t) ** self.gamma * CE_loss

        if self.reduction == 'mean':
            return torch.mean(loss)
        else:
            return loss


class MetricsCalculator:
    def __init__(self, num_classes: int, class_names: List[str]):
        self.num_classes = num_classes
        self.class_names = class_names
        self.metrics = None
```

```python
    def _initialize_metrics(self, device):
        self.metrics = {
            'confusion_matrix': torchmetrics.ConfusionMatrix(task="multiclass", num_classes=self.num_classes).to(device),
            'accuracy': torchmetrics.Accuracy(task="multiclass", num_classes=self.num_classes).to(device),
            'precision': torchmetrics.Precision(task="multiclass", num_classes=self.num_classes, average=None).to(device),
            'recall': torchmetrics.Recall(task="multiclass", num_classes=self.num_classes, average=None).to(device),
            'f1_score': torchmetrics.F1Score(task="multiclass", num_classes=self.num_classes, average=None).to(device),
            'specificity': torchmetrics.Specificity(task="multiclass", num_classes=self.num_classes, average=None).to(device),
            'auroc': torchmetrics.AUROC(task="multiclass", num_classes=self.num_classes, average=None).to(device),
            'auprc': torchmetrics.AveragePrecision(task="multiclass", num_classes=self.num_classes, average=None).to(device)
        }

    @staticmethod
    def to_cpu(t):
        return t.cpu().tolist() if isinstance(t, torch.Tensor) else t

    def compute_metrics(self, y_true: torch.Tensor, y_pred: torch.Tensor):
        device = y_true.device
        y_pred = y_pred.to(device)

        if self.metrics is None or next(iter(self.metrics.values())).device != device:
            self._initialize_metrics(device)

        # For AUROC, Average Precision, and probability-based metrics, use raw logits.
        y_pred_softmax = torch.softmax(y_pred, dim=1)

        # For class-prediction-based metrics (Accuracy, Precision, Recall), use argmax of logits.
        y_pred_classes = torch.argmax(y_pred, dim=1)

        # Ensure y_true is a long tensor with shape [batch_size]
        y_true = y_true.long()
        if y_true.dim() == 2:
            y_true = y_true.squeeze(1)
```

```python
        # Compute the metrics (class-prediction metrics on argmax,␣
↪probability-based on softmax)
        metrics_values = {
            'confusion_matrix': self.metrics['confusion_matrix'](y_pred_classes,␣
↪y_true),
            'accuracy': self.metrics['accuracy'](y_pred_classes, y_true),
            'precision': self.metrics['precision'](y_pred_classes, y_true),
            'recall': self.metrics['recall'](y_pred_classes, y_true),
            'f1_score': self.metrics['f1_score'](y_pred_classes, y_true),
            'specificity': self.metrics['specificity'](y_pred_classes, y_true),
            'auroc': self.metrics['auroc'](y_pred_softmax, y_true),
            'auprc': self.metrics['auprc'](y_pred_softmax, y_true),
        }

        # Balanced accuracy manually using recall
        balanced_accuracy = metrics_values['recall'].mean()  # Mean recall␣
↪across all classes
        metrics_values['balanced_accuracy'] = balanced_accuracy

        return metrics_values

    def generate_metrics_report(self, y_true: torch.Tensor, y_pred: torch.
↪Tensor) -> str:
        metrics_values = self.compute_metrics(y_true, y_pred)

        metrics_report = {}

        # Class-wise metrics
        for i, class_name in enumerate(self.class_names):
            metrics_report[class_name] = {
                'precision': self.to_cpu(metrics_values['precision'][i]),
                'recall': self.to_cpu(metrics_values['recall'][i]),
                'f1-score': self.to_cpu(metrics_values['f1_score'][i]),
                'specificity': self.to_cpu(metrics_values['specificity'][i])
            }

        # Macro (mean) averages for class-wise metrics
        metrics_report['macro avg'] = {
            'precision': self.to_cpu(metrics_values['precision'].mean()),
            'recall': self.to_cpu(metrics_values['recall'].mean()),
            'f1-score': self.to_cpu(metrics_values['f1_score'].mean()),
            'specificity': self.to_cpu(metrics_values['specificity'].mean())
        }

        # Overall accuracy
        metrics_report['accuracy'] = self.to_cpu(metrics_values['accuracy'])
```

```
        # AUROC per class and mean
        metrics_report['auc_roc_scores'] = {class_name: self.to_cpu(score) for␣
↪class_name, score in zip(self.class_names, metrics_values['auroc'])}
        metrics_report['mean_auc'] = self.to_cpu(metrics_values['auroc'].mean())

        # Average precision (AUPRC) per class and mean
        metrics_report['average_precision_scores'] = {class_name: self.
↪to_cpu(score) for class_name, score in zip(self.class_names,␣
↪metrics_values['auprc'])}
        metrics_report['mean_average_precision'] = self.
↪to_cpu(metrics_values['auprc'].mean())

        # Mean values for F1, Specificity, Sensitivity
        metrics_report['mean_f1_score'] = self.to_cpu(metrics_values['f1_score'].
↪mean())
        metrics_report['mean_specificity'] = self.
↪to_cpu(metrics_values['specificity'].mean())
        metrics_report['mean_sensitivity'] = self.
↪to_cpu(metrics_values['recall'].mean())  # Sensitivity is equivalent to recall

        # Balanced accuracy
        metrics_report['balanced_accuracy'] = self.
↪to_cpu(metrics_values['balanced_accuracy'])

        return json.dumps(metrics_report, indent=4)


def generate_metrics_report(y_true: torch.Tensor, y_pred: torch.Tensor) -> str:
    class_columns = ['Angioectasia', 'Bleeding', 'Erosion', 'Erythema', 'Foreign␣
↪Body', 'Lymphangiectasia', 'Normal', 'Polyp', 'Ulcer', 'Worms']
    calculator = MetricsCalculator(num_classes=len(class_columns),␣
↪class_names=class_columns)
    return calculator.generate_metrics_report(y_true, y_pred)


# Example usage (uncomment to run):
# if __name__ == "__main__":
#     num_samples = 100
#     num_classes = 10
#     y_true = torch.randint(0, num_classes, (num_samples,))
#     y_pred = torch.randn(num_samples, num_classes)

#     # CPU Test
#     report_cpu = generate_metrics_report(y_true, y_pred)
#     print("CPU Report:")
#     print(report_cpu)
```

```python
#     # Test on GPU if available
#     if torch.cuda.is_available():
#         y_true_gpu = y_true.cuda()
#         y_pred_gpu = y_pred.cuda()
#         report_gpu = generate_metrics_report(y_true_gpu, y_pred_gpu)
#         print("\nGPU Report:")
#         print(report_gpu)
```

[12]:
```python
# Load models
def load_model(model_class, model_path, device):
    model = model_class()
    model.load_state_dict(torch.load(model_path, map_location=device))
    model.eval()
    model.to(device)
    return model


# Preprocess data using VCEDatasetWithPaths
def preprocess_data_with_paths(xlsx_file, data_dir):
    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])
    ])

    val_dataset = VCEDatasetWithPaths(
        xlsx_file=xlsx_file,
        root_dir=data_dir,
        train_or_test='validation',
        transform=transform
    )

    dataloader = DataLoader(
        dataset=val_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers
    )

    return dataloader, val_dataset


from tqdm import tqdm

def ensemble_test_step(models, dataloader, device):
```

```
    all_predictions = []
    all_labels = []
    all_image_paths = []

    # Set all models to eval mode once before inference
    for model_idx, model in enumerate(models):
        model.eval()
        print(f"Model {model_idx + 1} set to eval mode.")

    with torch.no_grad():
        # Initialize the progress bar with total number of batches, leave=True␣
↪ensures it stays after completion
        with tqdm(total=len(dataloader), desc="Ensemble Inference",␣
↪unit="batch", leave=True, miniters=1, smoothing=0) as pbar:
            for batch_idx, (X, y, image_paths) in enumerate(dataloader):

                # Move data to device (GPU or CPU)
                X, y = X.to(device), y.to(device)

                # Calculate predictions for all models and average them
                ensemble_preds = torch.stack([
                    torch.softmax(model(X), dim=1)  # Softmax for probabilities
                    for model_idx, model in enumerate(models)
                ]).mean(dim=0)  # Average over the models


                # Print first 5 predictions for debugging

                # Append predictions, labels, and image paths to lists
                all_predictions.append(ensemble_preds.cpu())
                all_labels.append(y.cpu())
                all_image_paths.extend(image_paths)


                # Update the progress bar for each batch
                pbar.update(1)

    # Concatenate all predictions and labels
    predictions = torch.cat(all_predictions, dim=0)
    labels = torch.cat(all_labels, dim=0)

    return predictions, labels, all_image_paths


# Save predictions to Excel
def save_predictions_to_excel(image_paths, y_pred: torch.Tensor, output_path:␣
↪str):
```

```
    class_columns = ['Angioectasia', 'Bleeding', 'Erosion', 'Erythema', 'Foreign␣
↪Body', 'Lymphangiectasia', 'Normal', 'Polyp', 'Ulcer', 'Worms']

    # Convert logits to class predictions
    y_pred_classes = y_pred.argmax(dim=1).cpu().numpy()

    # Create a DataFrame to store image paths, predicted class, and prediction␣
↪probabilities
    df = pd.DataFrame({
        'image_path': image_paths,
        'predicted_class': [class_columns[i] for i in y_pred_classes],
        **{col: y_pred[:, i].cpu().numpy() for i, col in␣
↪enumerate(class_columns)}
    })

    # Save to Excel file
    df.to_excel(output_path, index=False)
    print(f"Predictions saved to {output_path}")

# Main function
def main():
    models = [load_model(cls, path, device) for cls, path in zip(model_classes,␣
↪model_paths)]

    # Use the new preprocessing function that returns image paths
    dataloader, dataset = preprocess_data_with_paths(test_xlsx, data_root_dir)

    # Run the ensemble test step, which now also returns image paths
    predictions, true_labels, image_paths = ensemble_test_step(models,␣
↪dataloader, device)

    # Generate metrics report (using logits as predictions)
    metrics_report = generate_metrics_report(true_labels, predictions)
    print("Metrics Report:\n", metrics_report)

    with open(metrics_report_dir, 'w') as f:
        f.write(metrics_report)

    print(f"Metrics report saved to {metrics_report_dir}.")

    # Save predictions to Excel (using argmax of predictions)
    # output_val_predictions = "../capsule-vision-2024/reports/validation_excel.
↪xlsx"
    output_val_predictions = "/kaggle/working/validation_excel.xlsx"
    save_predictions_to_excel(image_paths, predictions, output_val_predictions)
```

```python
# Run the script
if __name__ == "__main__":
    main()
```

```
Model 1 set to eval mode.
Model 2 set to eval mode.
Model 3 set to eval mode.
Model 4 set to eval mode.
Model 5 set to eval mode.
Model 6 set to eval mode.
Model 7 set to eval mode.
Model 8 set to eval mode.
Model 9 set to eval mode.
Model 10 set to eval mode.
Model 11 set to eval mode.
Model 12 set to eval mode.
Model 13 set to eval mode.
Model 14 set to eval mode.
Model 15 set to eval mode.
Model 16 set to eval mode.
Model 17 set to eval mode.
Model 18 set to eval mode.
```

```
Ensemble Inference: 100%|| 505/505 [16:12<00:00,  1.92s/batch]
```

```
Metrics Report:
 {
    "Angioectasia": {
        "precision": 0.8672199249267578,
        "recall": 0.8410462737083435,
        "f1-score": 0.8539325594902039,
        "specificity": 0.9959065914154053
    },
    "Bleeding": {
        "precision": 0.8689458966255188,
        "recall": 0.8495821952819824,
        "f1-score": 0.8591549396514893,
        "specificity": 0.9970836043357849
    },
    "Erosion": {
        "precision": 0.8001729846000671,
        "recall": 0.8008658289909363,
        "f1-score": 0.8005192279815674,
        "specificity": 0.9845763444900513
    },
    "Erythema": {
        "precision": 0.6909090876579285,
        "recall": 0.6397306323051453,
```

```
    "f1-score": 0.6643356680870056,
    "specificity": 0.9946321249008179
},
"Foreign Body": {
    "precision": 0.8622589707374573,
    "recall": 0.9205882549285889,
    "f1-score": 0.8904694318771362,
    "specificity": 0.996833860874176
},
"Lymphangiectasia": {
    "precision": 0.8510638475418091,
    "recall": 0.9329445958137512,
    "f1-score": 0.8901251554489136,
    "specificity": 0.9964532256126404
},
"Normal": {
    "precision": 0.9816513657569885,
    "recall": 0.9840481877326965,
    "f1-score": 0.9828483462333679,
    "specificity": 0.9412223696708679
},
"Polyp": {
    "precision": 0.7770700454711914,
    "recall": 0.7319999933242798,
    "f1-score": 0.7538620233535767,
    "specificity": 0.9932830333709717
},
"Ulcer": {
    "precision": 0.9816176295280457,
    "recall": 0.9335664510726929,
    "f1-score": 0.9569892287254333,
    "specificity": 0.9996844530105591
},
"Worms": {
    "precision": 0.9855072498321533,
    "recall": 1.0,
    "f1-score": 0.9927007555961609,
    "specificity": 0.9999377727508545
},
"macro avg": {
    "precision": 0.866641640663147,
    "recall": 0.863437294960022,
    "f1-score": 0.864493727684021,
    "specificity": 0.9899613261222839
},
"accuracy": 0.9461318850517273,
"auc_roc_scores": {
    "Angioectasia": 0.987118124961853,
```

```
        "Bleeding": 0.9854714870452881,
        "Erosion": 0.9848971366882324,
        "Erythema": 0.9891839623451233,
        "Foreign Body": 0.9885151386260986,
        "Lymphangiectasia": 0.9940540790557861,
        "Normal": 0.9960157871246338,
        "Polyp": 0.9857802391052246,
        "Ulcer": 0.997887909412384,
        "Worms": 0.9999945163726807
    },
    "mean_auc": 0.9908918142318726,
    "average_precision_scores": {
        "Angioectasia": 0.9045501351356506,
        "Bleeding": 0.9020779132843018,
        "Erosion": 0.8722904324531555,
        "Erythema": 0.7163441777229309,
        "Foreign Body": 0.9387128353118896,
        "Lymphangiectasia": 0.9464475512504578,
        "Normal": 0.9987960457801819,
        "Polyp": 0.8086738586425781,
        "Ulcer": 0.9672346711158752,
        "Worms": 0.9987329244613647
    },
    "mean_average_precision": 0.9053860902786255,
    "mean_f1_score": 0.864493727684021,
    "mean_specificity": 0.9899613261222839,
    "mean_sensitivity": 0.863437294960022,
    "balanced_accuracy": 0.863437294960022
}
```


```
Metrics report saved to /kaggle/working/metrics_report.json.
Predictions saved to /kaggle/working/validation_excel.xlsx
```

[ ]:

```

```