training-capsule-vision

October 23, 2024

1 Initial Setup

1.1 Imports

[1]:
```
# 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!
```

1.2 Configs

[2]:
```
# Setup hyperparameters
NUM_EPOCHS = 20
BATCH_SIZE = 32
HIDDEN_UNITS = 32
LEARNING_RATE = 0.003
NUM_WORKERS = 4

# Setup directories
train_dir = "training"
test_dir = "validation"
train_xlsx_filename = "training_data.xlsx"
test_xlsx_filename = "validation_data.xlsx"

# data_dir = "../capsule-vision-2024/data/Dataset"
data_dir="/kaggle/input/capsule-vision-2024-data/Dataset"
# save_dir = "../capsule-vision-2024/models"
save_dir="/kaggle/working/models"
# logging_dir = "../capsule-vision-2024/logs"
logging_dir="/kaggle/working/logs"

# Setup target device
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Device: {device}\n\n")


data_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.3),
    transforms.RandomRotation(degrees=15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.
 ↪1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1), scale=(0.9, 1.1)),
    transforms.RandomPerspective(distortion_scale=0.2, p=0.5),
    transforms.ToTensor(),  # Convert the image to a tensor here
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    transforms.RandomErasing(p=0.2, scale=(0.02, 0.33), ratio=(0.3, 3.3),␣
 ↪value='random'),
    transforms.RandomApply([transforms.GaussianBlur(kernel_size=3)], p=0.3)
])
```

```
Device: cuda
```

2 Utility Functions

[3]:
```
def setup_logger(model_name: str) -> Logger:
    log_dir = os.path.join(logging_dir, model_name)
    os.makedirs(log_dir, exist_ok=True)
    log_file = os.path.join(log_dir, f"{datetime.now().
 ↪strftime('%Y%m%d_%H%M%S')}.log")

    logger = getLogger(model_name)

    # Only set up the logger if it hasn't been set up before
    if not logger.handlers:
        logger.setLevel(INFO)

        file_handler = FileHandler(log_file)
        stream_handler = StreamHandler()

        formatter = Formatter('%(asctime)s - %(name)s - %(levelname)s -␣
 ↪%(message)s')
        file_handler.setFormatter(formatter)
        stream_handler.setFormatter(formatter)

        logger.addHandler(file_handler)
        logger.addHandler(stream_handler)

    return logger


def save_model(model: nn.Module, target_dir: str, model_name: str):
    target_dir_path = Path(target_dir)
    target_dir_path.mkdir(parents=True, exist_ok=True)

    assert model_name.endswith(".pth") or model_name.endswith(".pt"),␣
 ↪"model_name should end with '.pt' or '.pth'"

    model_save_path = target_dir_path / model_name
    print(f"[INFO] Saving model to: {model_save_path}")
    torch.save(obj=model.state_dict(), f=model_save_path)


def save_metrics_report(report: Dict, model_name: str, epoch: int,
                        # save_dir: str = "../capsule-vision-2024/logs/reports"
                        save_dir: str = "/kaggle/working/logs/reports"):
    report_dir = os.path.join(save_dir, model_name)
    os.makedirs(report_dir, exist_ok=True)  # Create the directory if it doesn't␣
 ↪exist
```

```
    report_filename = f"metrics_epoch_{epoch+1}.json"  # Save report for each␣
 ↪epoch
    report_path = os.path.join(report_dir, report_filename)

    # Save the report as a JSON file
    with open(report_path, 'w') as report_file:
        json.dump(report, report_file, indent=4)

    print(f"[INFO] Saved metrics report for {model_name}, epoch {epoch+1} at␣
 ↪{report_path}")


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']
    y_pred_classes = y_pred.argmax(dim=1).cpu().numpy()
    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)}
    })
    df.to_excel(output_path, index=False)


def is_torch_available():
    return torch is not None


def seed_everything(seed=42):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True


def count_parameters(model: nn.Module):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


def model_size_mb(model: nn.Module):
    total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    size_in_bytes = total_params * 4
    size_in_mb = size_in_bytes / (1024 ** 2)
```

```
    return size_in_mb
```

3 Data Loading

[4]:
```
class VCEDataset(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):
        img_path = os.path.join(self.root_dir, self.annotations.iloc[index, 0].
↪replace("\\", "/"))
        image = Image.open(img_path).convert('RGB')  # Ensure image is in RGB␣
↪format

        target = self.annotations.iloc[index, 2:].values
        y_label = torch.tensor(target.argmax(), dtype=torch.long)

        if self.transform:
            image = self.transform(image)

        return image, y_label

    def get_class_weights(self):
        class_counts = self.annotations.iloc[:, 2:].sum().values
        class_weights = 1.0 / class_counts
        class_weights = class_weights / class_weights.sum()  # Normalize
        return torch.FloatTensor(class_weights)

def create_dataloaders(
    train_xlsx: str,
    test_xlsx: str,
    train_root_dir: str,
    test_root_dir: str,
    data_root_dir: str,
    transform: transforms.Compose,
    batch_size: int,
```

```
    num_workers: int = 4
):
    # Create datasets
    train_dataset = VCEDataset(
        xlsx_file=train_xlsx,
        root_dir=data_root_dir,
        train_or_test=train_root_dir,
        transform=transform,
    )

    test_dataset = VCEDataset(
        xlsx_file=test_xlsx,
        root_dir=data_root_dir,
        train_or_test=test_root_dir,
        transform=transform,
    )

    # Calculate sample weights for training set
    class_weights = train_dataset.get_class_weights()
    train_targets = [train_dataset.annotations.iloc[i, 2:].values.argmax() for i␣
 ↪in range(len(train_dataset))]
    sample_weights = [class_weights[t] for t in train_targets]

    # Create weighted sampler for training set
    sampler = WeightedRandomSampler(weights=sample_weights,␣
 ↪num_samples=len(train_dataset), replacement=True)

    # Create dataloaders
    train_loader = DataLoader(
        dataset=train_dataset,
        batch_size=batch_size,
        sampler=sampler,
        num_workers=num_workers
    )

    test_loader = DataLoader(
        dataset=test_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers
    )

    return train_loader, test_loader
```

[5]:
```
# Create DataLoaders with help from data_setup.py
train_loader, test_loader = create_dataloaders(
    train_xlsx=train_xlsx_filename,
```

```
    test_xlsx=test_xlsx_filename,
    train_root_dir=train_dir,
    test_root_dir=test_dir,
    data_root_dir=data_dir,
    transform=data_transform,
    batch_size=BATCH_SIZE,
    num_workers=NUM_WORKERS,
)
print("Data Loaded!\n\n")


# Class labels (assuming these are your target classes)
class_columns = ['Angioectasia', 'Bleeding', 'Erosion', 'Erythema', 'Foreign␣
 ↪Body', 'Lymphangiectasia', 'Normal', 'Polyp', 'Ulcer', 'Worms']
```

```
Data Loaded!
```

4 Models

[6]:
```
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):
    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
```

```python
# 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

# 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)
    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)
```

[7]:
```
num_classes = len(class_columns)

# Define a list of models for training
model_list = {
    "EfficientNet": model_efficientnet(pretrained=True, num_classes=num_classes),
    "ResNet": model_resnet(pretrained=True, num_classes=num_classes),
    "MobileNetV3": model_mobilenetv3(pretrained=True, num_classes=num_classes),
    "RegNet": model_regnet(pretrained=True, num_classes=num_classes),
    "DenseNet": model_densenet(pretrained=True, num_classes=num_classes),
    "InceptionV3": model_inception_v3(pretrained=True, num_classes=num_classes),
    "ResNeXt": model_resnext(pretrained=True, num_classes=num_classes),
```

```
    "WideResNet": model_wide_resnet(pretrained=True, num_classes=num_classes),
    "MNASNet": model_mnasnet(pretrained=True, num_classes=num_classes),
    "SEResNet50": model_seresnet50(pretrained=True, num_classes=num_classes),
    "ConvNeXt": model_convnext(pretrained=True, num_classes=num_classes),

    "ViT": model_vit(pretrained=True, num_classes=num_classes),
    "SwinTransformer": model_swin(pretrained=True, num_classes=num_classes),
    "DeiT": model_deit(pretrained=True, num_classes=num_classes),
    "BEiT": model_beit(pretrained=True, num_classes=num_classes),
    "CaiT": model_cait(pretrained=True, num_classes=num_classes),
    "TwinsSVT": model_twins_svt(pretrained=True, num_classes=num_classes),
    "EfficientFormer": model_efficientformer(pretrained=True,␣
 ↪num_classes=num_classes)
}
print("Models Loaded!\n\n")
```

```
model.safetensors:   0%|          | 0.00/86.5M [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/102M [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/22.1M [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/346M [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/353M [00:00<?, ?B/s]
Models Loaded!
```

5 Metrics

[8]:
```
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

    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)

        # 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)
```

6 Training

6.1 Engine

[13]:
```python
from tqdm import tqdm

def train_step(model: nn.Module,
               dataloader: DataLoader,
               loss_fn: nn.Module,
               optimizer: optim.Optimizer,
               device: torch.device) -> Tuple[float, float, torch.Tensor, torch.
 ↪Tensor]:
    model.train()
    train_loss, correct = 0, 0
    total = 0
    all_predictions = []
    all_labels = []

    train_progress = tqdm(dataloader, desc="Training", leave=False)
    for batch_idx, (X, y) in enumerate(train_progress):
        X, y = X.to(device), y.to(device)
        optimizer.zero_grad(set_to_none=True)

        y_pred_logits = model(X)
        loss = loss_fn(y_pred_logits, y)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * X.size(0)
        _, predicted = y_pred_logits.max(1)
        total += y.size(0)
        correct += predicted.eq(y).sum().item()

        all_predictions.append(y_pred_logits)
        all_labels.append(y)

        train_progress.set_postfix(
            loss=f"{loss.item():.4f}",
            acc=f"{correct/total:.4f}"
        )

    train_loss /= total
    train_acc = correct / total
    train_preds = torch.cat(all_predictions, dim=0)
    train_labels = torch.cat(all_labels, dim=0)

    return train_loss, train_acc, train_preds, train_labels
```

```python
def test_step(model: nn.Module,
              dataloader: DataLoader,
              loss_fn: nn.Module,
              device: torch.device) -> Tuple[float, float, torch.Tensor, torch.
 ↪Tensor]:
    model.eval()
    test_loss, correct = 0, 0
    total = 0
    all_predictions = []
    all_labels = []

    val_progress = tqdm(dataloader, desc="Validation", leave=False)
    with torch.inference_mode():
        for batch_idx, (X, y) in enumerate(val_progress):
            X, y = X.to(device), y.to(device)

            y_pred_logits = model(X)
            loss = loss_fn(y_pred_logits, y)

            test_loss += loss.item() * X.size(0)
            _, predicted = y_pred_logits.max(1)
            total += y.size(0)
            correct += predicted.eq(y).sum().item()

            all_predictions.append(y_pred_logits)
            all_labels.append(y)

            val_progress.set_postfix(
                loss=f"{loss.item():.4f}",
                acc=f"{correct/total:.4f}"
            )

    test_loss /= total
    test_acc = correct / total
    test_preds = torch.cat(all_predictions, dim=0)
    test_labels = torch.cat(all_labels, dim=0)

    return test_loss, test_acc, test_preds, test_labels


def train(model: nn.Module,
          train_dataloader: DataLoader,
          test_dataloader: DataLoader,
          optimizer: optim.Optimizer,
          loss_fn: nn.Module,
```

```
        epochs: int,
        device: torch.device,
        model_name: str,
        save_dir: str,
        patience: int = 5,
        tolerance: float = 1e-4
        ) -> Dict[str, List]:

    results = {
        "train_loss": [],
        "train_acc": [],
        "test_loss": [],
        "test_acc": [],
        "mean_auc": [],
        "balanced_accuracy": []
    }

    model.to(device)
    scheduler = CosineAnnealingLR(optimizer, T_max=epochs)
    logger = setup_logger(model_name)
    logger.info(f"Training started for model: {model_name}")

    best_score = -float('inf')
    best_epoch = 0
    no_improvement_count = 0

    for epoch in range(epochs):
        logger.info(f"Epoch {epoch+1}/{epochs}")

        train_loss, train_acc, train_preds, train_labels = train_step(
            model, train_dataloader, loss_fn, optimizer, device
        )
        test_loss, test_acc, test_preds, test_labels = test_step(
            model, test_dataloader, loss_fn, device
        )

        scheduler.step()

        train_preds_probs = torch.softmax(train_preds, dim=1)
        test_preds_probs = torch.softmax(test_preds, dim=1)

        train_metrics = generate_metrics_report(train_labels, train_preds_probs)
        test_metrics = generate_metrics_report(test_labels, test_preds_probs)

        logger.info(f"Train Metrics:\n{train_metrics}")
        logger.info(f"Test Metrics:\n{test_metrics}")
        logger.info(
```

```
            f"train_loss: {train_loss:.4f} | "
            f"train_acc: {train_acc:.4f} | "
            f"test_loss: {test_loss:.4f} | "
            f"test_acc: {test_acc:.4f}"
        )

        test_metrics_dict = json.loads(test_metrics)
        current_mean_auc = test_metrics_dict['mean_auc']
        current_balanced_accuracy = test_metrics_dict['balanced_accuracy']

        current_score = (current_mean_auc + current_balanced_accuracy) / 2

        results["train_loss"].append(train_loss)
        results["train_acc"].append(train_acc)
        results["test_loss"].append(test_loss)
        results["test_acc"].append(test_acc)
        results["mean_auc"].append(current_mean_auc)
        results["balanced_accuracy"].append(current_balanced_accuracy)

        if current_score > best_score + tolerance:
            best_score = current_score
            best_epoch = epoch + 1
            no_improvement_count = 0
            save_model(model, save_dir, f"{model_name}_best.pth")
            logger.info(f"Best model saved with combined score: {best_score:.4f}␣
↪(Mean AUC: {current_mean_auc:.4f}, Balanced Accuracy:␣
↪{current_balanced_accuracy:.4f})")
        else:
            no_improvement_count += 1
            logger.info(f"No improvement for {no_improvement_count} consecutive␣
↪epochs.")

        if no_improvement_count >= patience:
            logger.info(f"Early stopping after {patience} epochs of no␣
↪improvement.")
            break

        save_metrics_report({
            "epoch": epoch + 1,
            "train_metrics": train_metrics,
            "test_metrics": test_metrics
        }, model_name, epoch)

    logger.info(f"Training completed. Best model at epoch {best_epoch} with␣
↪combined score: {best_score:.4f}")

    del model, optimizer
```

```
    torch.cuda.empty_cache()
    torch.cuda.synchronize()

    return results
```

6.2 Main

[14]:
```
# Dictionary to store results for each model
results_dict = {}

# Loop through each model and train them
for model_name, model in model_list.items():
    print(f"\nTraining model: {model_name}")

    # Move model to target device
    model = model.to(device)

    # Define optimizer (AdamW as an example) and loss function (Cross Entropy)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4, weight_decay=0.05)
    loss_fn = FocalLoss() # CrossEntropyLoss()

    # Train the model using engine.train function
    results = train(
        model=model,
        train_dataloader=train_loader,
        test_dataloader=test_loader,
        optimizer=optimizer,
        loss_fn=loss_fn,
        epochs=NUM_EPOCHS,
        device=device,
        model_name=model_name,
        save_dir=save_dir,
    )

    # Store the results
    results_dict[model_name] = results

# After the training loop, results_dict will contain training history for each␣
 ↪model
print("Training complete for all models.")
```

```
2024-10-20 09:04:09,812 - EfficientNet - INFO - Training started for model:
EfficientNet
2024-10-20 09:04:09,813 - EfficientNet - INFO - Epoch 1/20


Training model: EfficientNet
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[14], line 16
     13 loss_fn = FocalLoss() # CrossEntropyLoss()
     15 # Train the model using engine.train function
---> 16 results = train(
     17     model=model,
     18     train_dataloader=train_loader,
     19     test_dataloader=test_loader,
     20     optimizer=optimizer,
     21     loss_fn=loss_fn,
     22     epochs=NUM_EPOCHS,
     23     device=device,
     24     model_name=model_name,
     25     save_dir=save_dir,
     26 )
     28 # Store the results
     29 results_dict[model_name] = results

Cell In[13], line 118, in train(model, train_dataloader, test_dataloader,␣
 ↪optimizer, loss_fn, epochs, device, model_name, save_dir, patience, tolerance)
    115 for epoch in range(epochs):
    116     logger.info(f"Epoch {epoch+1}/{epochs}")
--> 118     train_loss, train_acc, train_preds, train_labels = train_step(
    119         model, train_dataloader, loss_fn, optimizer, device
    120     )
    121     test_loss, test_acc, test_preds, test_labels = test_step(
    122         model, test_dataloader, loss_fn, device
    123     )
    125     scheduler.step()

Cell In[13], line 24, in train_step(model, dataloader, loss_fn, optimizer, device
     21 loss.backward()
     22 optimizer.step()
---> 24 train_loss += loss.item() * X.size(0)
     25 _, predicted = y_pred_logits.max(1)
     26 total += y.size(0)

KeyboardInterrupt:
```

[ ]:

[ ]: