```python
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import models,transforms
from torch.utils.data import Dataset, DataLoader
from PIL import Image
from torch.optim.lr_scheduler import CosineAnnealingLR
from tqdm import tqdm
import numpy as np

# 경로 설정
train_dir = './train'
test_dir = './test'

# 학습 데이터 증강
train_transform = transforms.Compose([
    transforms.Resize((256, 256)),  # 약간 더 큰 이미지로 변환
    transforms.RandomResizedCrop((224, 224)),  # 랜덤으로 크롭하여 ResNet 입력 크기 맞춤
    transforms.RandomHorizontalFlip(p=0.5),  # 좌우 반전
    transforms.RandomRotation(20),  # 랜덤 회전 (각도 ±20도)
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # 밝기, 대비, 채도, 색조 변화
    transforms.ToTensor(),  # Tensor로 변환
    transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))  # ImageNet 정규화
])

# 테스트 데이터 변환 (증강 없이 정규화만 적용)
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),  # ResNet 입력 크기와 동일하게 변환
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))
])

class ImageFolderWithTransforms(Dataset):
    def __init__(self, root, transform=None):
        self.root = root
        self.transform = transform
        self.classes = sorted(os.listdir(root))
        self.class_to_idx = {cls_name: i for i, cls_name in enumerate(self.classes)}
        self.image_paths = []
        self.labels = []

        for cls_name in self.classes:
            cls_path = os.path.join(root, cls_name)
            for img_name in os.listdir(cls_path):
                self.image_paths.append(os.path.join(cls_path, img_name))
                self.labels.append(self.class_to_idx[cls_name])

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

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img = Image.open(img_path).convert('RGB')

        if self.transform:
            img = self.transform(img)  # torchvision.transforms에 맞게 호출
        label = self.labels[idx]
        return img, label

train_dataset = ImageFolderWithTransforms(root=train_dir, transform=train_transform)
test_dataset = ImageFolderWithTransforms(root=test_dir, transform=test_transform)

# 데이터로더 정의
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=4)
```

1. **데이터 증강**

- 학습 데이터: 크기 조정, 랜덤 크롭, 좌우 반전, 회전 및 밝기/대비 변화 적용

- 테스트 데이터: 크기 조정과 정규화만 적용

2. 데이터 로더

- 배치 크기 32로 설정하여 데이터를 로드
- shuffle=True를 통해 학습 데이터 순서를 무작위로 변경

```python
num_classes = len(train_dataset.classes)

# 사전 학습된 ResNet50 불러오기
from torchvision.models import ResNet50_Weights, resnet50
model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)

# FC 레이어 수정 (Dropout 추가)
model.fc = nn.Sequential(
    nn.Dropout(p=0.5),
    nn.Linear(model.fc.in_features, num_classes)
)

# GPU 설정
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if torch.cuda.is_available():
    print("CUDA is available. Using GPU for training.")
else:
    print("CUDA is not available. Using CPU for training.")
model = model.to(device)

criterion = nn.CrossEntropyLoss()

# 초기: FC 레이어만 학습 -> 나중에 전체 레이어 학습
for param in model.parameters():
    param.requires_grad = False
for param in model.fc.parameters():
    param.requires_grad = True
```

1. **사전 학습된 ResNet50**
    - **ImageNet 가중치**를 불러와 네트워크 초기화
2. **FC 레이어 수정**
    - Fully Connected (FC) 레이어를 새로운 분류층으로 변경
    - Dropout 추가(p=0.5)로 과적합 방지

3. **학습 레이어 설정**
    - **초기 단계**: 최상위 FC 레이어만 학습
    - **이후 단계**: 전체 모델의 가중치를 미세 조정

```python
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3)
scheduler = CosineAnnealingLR(optimizer, T_max=10)

def evaluate(model, loader, criterion):
    model.eval()
    running_loss = 0.0
    running_corrects = 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            _, preds = torch.max(outputs, 1)
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)
    epoch_loss = running_loss / len(loader.dataset)
    epoch_acc = running_corrects.double() / len(loader.dataset)
    return epoch_loss, epoch_acc
```

1. 평가 함수

- 모델을 평가 모드로 전환 (model.eval())
- torch.no_grad()로 그래디언트 계산을 비활성화
- 손실값 및 정확도 계산

```python
def train_and_test(model, train_loader, test_loader, criterion, num_epochs=30, log_file='resnet_training_log.txt'):
    best_test_acc = 0.0

    # 로그 파일 초기화
    with open(log_file, mode='w') as file:
        file.write("Epoch\tTrain Loss\tTrain Acc\tTest Loss\tTest Acc\n")

    # 옵티마이저와 스케줄러 초기화 (초기에는 FC 레이어만 학습)
    optimizer = optim.Adam(model.fc.parameters(), lr=1e-3)
    scheduler = CosineAnnealingLR(optimizer, T_max=10)

    # 몇 epoch 이후 전체 레이어 언프리즈할지 결정 (예: 5epoch 이후)
    unfreeze_epoch = 5

    for epoch in range(num_epochs):
        print(f"\nEpoch {epoch+1}/{num_epochs}")
        print("-" * 30)

        # unfreeze_epoch 도달 시 전체 모델 학습 전환
        if epoch == unfreeze_epoch:
            for param in model.parameters():
                param.requires_grad = True
            # 옵티마이저와 스케줄러 재설정
            optimizer = optim.Adam(model.parameters(), lr=1e-4)
            scheduler = CosineAnnealingLR(optimizer, T_max=20)
            print("Unfreezing all layers and adjusting optimizer/lr scheduler")

        model.train()
        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in tqdm(train_loader, desc="Training"):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            _, preds = torch.max(outputs, 1)
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        train_loss = running_loss / len(train_loader.dataset)
        train_acc = running_corrects.double() / len(train_loader.dataset)

        # 테스트 평가
        test_loss, test_acc = evaluate(model, test_loader, criterion)

        # 학습률 스케줄러 스텝
        scheduler.step()

        print(f"Train Loss: {train_loss:.4f} | Train Acc: {train_acc:.4f}")
        print(f"Test Loss: {test_loss:.4f} | Test Acc: {test_acc:.4f}")

        # 로그 기록
        with open(log_file, 'a') as f:
            f.write(f"{epoch+1}\t{train_loss:.4f}\t{train_acc:.4f}\t{test_loss:.4f}\t{test_acc:.4f}\n")

        # 최고 성능 모델 저장
        if test_acc > best_test_acc:
            best_test_acc = test_acc
            torch.save(model.state_dict(), 'best_finetuned_resnet.pth')
            print("Best model saved.")

    print(f"\nTraining complete. Best Test Acc: {best_test_acc:.4f}")
    return model


if __name__ == '__main__':
    model = train_and_test(model, train_loader, test_loader, criterion, num_epochs=30, log_file='resnet_training_각종기법_log.txt')

```

1. **미세 조정 (Fine-Tuning)**
    - **Epoch 5** 이후 모든 레이어의 학습을 활성화하고 학습률을 조정합니다.
    - CosineAnnealingLR 스케줄러를 사용해 학습률을 조절합니다.

2. 훈련 및 테스트

- 학습 손실과 정확도 계산
- 평가 함수 호출을 통해 테스트 성능 확인

3. **모델 저장**
    - 테스트 정확도가 최고일 때 모델 가중치를 저장합니다.