```python
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
from tqdm import tqdm
from torchvision.models import ViT_B_16_Weights  # ViT 모델의 가중치 가져오기

# 경로 설정
train_dir = './train'  # 학습 데이터 경로
test_dir = './test'    # 테스트 데이터 경로

# 데이터 전처리
transform_train = transforms.Compose([
    transforms.Resize((224, 224)),  # ViT의 입력 크기와 일치
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],  # ViT 사전 학습된 모델의 정규화 값
                         std=[0.229, 0.224, 0.225])
])

transform_test = transforms.Compose([
    transforms.Resize((224, 224)),  # ViT의 입력 크기와 일치
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],  # ViT 사전 학습된 모델의 정규화 값
                         std=[0.229, 0.224, 0.225])
])

# 데이터셋 및 데이터로더
train_dataset = datasets.ImageFolder(root=train_dir, transform=transform_train)
test_dataset = datasets.ImageFolder(root=test_dir, transform=transform_test)

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)

print(f"훈련셋 크기: {len(train_dataset)}")
print(f"테스트셋 크기: {len(test_dataset)}")

```

1. **데이터 전처리**

- **학습 데이터**: RandomHorizontalFlip과 RandomRotation을 적용하여 다양한 형태의 이미지 데이터를 제공하고 모델의 일반화 성능을 높였습니다.
- **테스트 데이터**: 증강을 배제하고 정규화(Normalize)만 적용했습니다.
- 정규화는 **ImageNet**에서 사전 학습된 모델과 일관성을 유지하기 위해 평균과 표준편차를 사용했습니다.

2. **데이터셋 및 데이터로더**

- PyTorch의 ImageFolder로 폴더 구조의 학습 및 테스트 데이터를 불러왔습니다.
- DataLoader를 사용하여 데이터를 배치 단위로 처리하며, 학습 데이터는 shuffle=True로 설정하여 데이터 순서를 무작위화했습니다.

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

# 사전 학습된 ViT 불러오기
model = models.vit_b_16(weights=ViT_B_16_Weights.IMAGENET1K_V1)

# 모든 파라미터 freeze (마지막 classification head 제외)
for param in model.parameters():
    param.requires_grad = False

# 최상위 레이어 변경 (랜덤 초기화)
# torchvision의 ViT에서는 `heads.head`로 접근
model.heads.head = nn.Linear(model.heads.head.in_features, num_classes)

# 마지막 레이어의 파라미터만 학습 가능하도록 설정
for param in model.heads.head.parameters():
    param.requires_grad = True

# 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)
```

1. **사전 학습된 ViT 모델 로드** : torchvision의 **ViT-B/16** 모델을 불러왔으며, 이는 ImageNet 데이터셋으로 사전 학습된 가중치를 사용합니다.

2. 모델 파라미터 동결 : **Transfer Learning**을 위해 기존 파라미터를 동결(freeze)하고, 마지막 레이어(heads.head)만 학습되도록 설정했습니다.

3. **상위 레이어 수정** : 새로운 **Fully Connected Layer**를 추가하여 데이터셋의 클래스 수에 맞게 조정하였습니다.

4. **GPU 설정** : GPU가 사용 가능한지 확인하고 모델을 CUDA 장치로 이동시켰습니다.

```python
criterion = nn.CrossEntropyLoss()

def train_and_test(model, train_loader, test_loader, criterion, num_epochs=30, log_file='training_log.txt'):
    best_test_acc = 0.0  # 최고 테스트 정확도 추적

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

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

        # **학습 단계**
        model.train()  # 학습 모드
        running_loss = 0.0
        running_corrects = 0

        # 단계에 따라 optimizer와 학습 가능한 파라미터 설정
        if epoch == 0:
            # 첫 번째 epoch: 마지막 레이어만 학습
            for param in model.parameters():
                param.requires_grad = False
            for param in model.heads.head.parameters():
                param.requires_grad = True
            optimizer = optim.Adam(model.heads.head.parameters(), lr=1e-4)
            print("첫 번째 epoch: 마지막 출력층만 학습")
        elif epoch == 1:
            # 두 번째 epoch부터 전체 모델을 미세 조정
            for param in model.parameters():
                param.requires_grad = True
            optimizer = optim.Adam(model.parameters(), lr=1e-5)
            print("두 번째 epoch부터 전체 모델을 미세 조정")

        for inputs, labels in tqdm(train_loader, desc="Training"):
            # 데이터를 CUDA 장치로 이동
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            _, preds = torch.max(outputs, 1)

            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_train_loss = running_loss / len(train_loader.dataset)
        epoch_train_acc = running_corrects.double() / len(train_loader.dataset)
        print(f"Train Loss: {epoch_train_loss:.4f} | Train Acc: {epoch_train_acc:.4f}")

        # **평가 단계**
        model.eval()  # 평가 모드
        test_loss = 0.0
        test_corrects = 0

        with torch.no_grad():
            for inputs, labels in tqdm(test_loader, desc="Testing"):
                inputs = inputs.to(device)
                labels = labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)
                _, preds = torch.max(outputs, 1)

                test_loss += loss.item() * inputs.size(0)
                test_corrects += torch.sum(preds == labels.data)

        epoch_test_loss = test_loss / len(test_loader.dataset)
        epoch_test_acc = test_corrects.double() / len(test_loader.dataset)
        print(f"Test Loss: {epoch_test_loss:.4f} | Test Acc: {epoch_test_acc:.4f}")

        # **로그 저장**
        with open(log_file, mode='a') as file:
            file.write(f"{epoch+1}\t{epoch_train_loss:.4f}\t{epoch_train_acc:.4f}\t{epoch_test_loss:.4f}\t{epoch_test_acc:.4f}\n")

        # **최고 테스트 정확도 모델 저장**
        if epoch_test_acc > best_test_acc:
            best_test_acc = epoch_test_acc
            torch.save(model.state_dict(), 'best_finetuned_vit.pth')
            print("Best model saved.")

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

    return model

if __name__ == '__main__':
    # 모델 학습 및 테스트 수행
    model = train_and_test(model, train_loader, test_loader, criterion, num_epochs=30, log_file='vit_training_log.txt')
    # os.system("shutdown /s /t 60")  # 학습 종료 시 Windows에서 60초 후 종료
```

1. **학습 및 평가 단계** : 첫 번째 Epoch에는 마지막 레이어만 학습하고, 두 번째 Epoch 이후에는 전체 모델을 미세 조정(fine-tuning)합니다.

2. **Train Step** : Forward → Loss 계산 → Backward → Optimizer 업데이트를 수행합니다.

3. **Test Step** : model.eval()로 설정하고 Gradient 계산을 비활성화한 상태에서 정확도와 손실을 평가합니다.

4. **최고 성능 모델 저장 :** 테스트 정확도가 갱신될 때마다 모델 가중치를 저장합니다.

5. 학습 실행 : train_and_test 함수를 호출하여 모델 학습과 평가를 수행합니다. **학습 로그**는 vit_training_log.txt 파일에 저장됩니다.

6. 결과 : 훈련 및 테스트 손실과 정확도를 출력하고, 최고 성능 모델을 파일로 저장합니다.