

# Multi Class Classification

## (다중 분류)

수업:	기	계	학	습
교수:	홍	충	표	
학과:	컴	퓨	터	공 학 부
학번:	2	0	2	2 3 1 8 1
이름:	이	성	준	

## 서론

- 1.1 F1 Score 정의 및 중요성
- 1.2 개발 환경

## 방법론

- 2.1 MoE & Multimodal Train
- 2.2 Simple CNN
- 2.3 Transfer Learning
  - 2.3.1 MobileNet
  - 2.3.2 EfficientNet
  - 2.3.3 ViT (Vision Transformer)
  - 2.3.4 ResNet-50

## 데이터 전처리

- 3.1 데이터 정제 및 증강

## 실험 결과

- 4.1 MoE & Multimodal Train 결과
- 4.2 Simple CNN 결과
- 4.3 Transfer Learning 결과
  - 4.3.1 MobileNet 성능
  - 4.3.2 EfficientNet 성능
  - 4.3.3 ViT 성능
  - 4.3.4 ResNet-50 성능

## 결론

- 5.1 결과 요약
- 5.2 방법론 비교 및 한계

## 1.1 F1 Score 정의 및 중요성

F1 Score 정의 :

F1 Score는 분류 모델의 성능을 평가하는 대표적인 지표로, 정밀도(Precision)와 재현율(Recall)의 조화 평균을 나타냅니다. 이는 특히 불균형 데이터 셋에서 모델의 성능을 공정하게 평가하는 데 유용하며, 다음과 같은 수식으로 계산됩니다:

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- 정밀도(Precision)는 모델이 양성으로 예측한 샘플 중 실제로 양성인 비율을 의미합니다.

$$Precision = \frac{TP}{TP + FP}$$

- 재현율(Recall)은 실제 양성 샘플 중 모델이 정확히 예측한 비율을 나타냅니다.

$$Recall = \frac{TP}{TP + FN}$$

TP(True Positive), FP(False Positive), FN(False Negative)은 각각 분류 결과에서 나타나는 지표들입니다.

F1 Score의 중요성

1. 불균형 데이터에서의 효과적 평가 데이터 클래스 간 비율이 불균형한 상황에서는 단순히 정확도(Accuracy)만으로 모델의 성능을 제대로 평가하기 어렵습니다. F1 Score는 정밀도와 재현율을 모두 고려하여 극단적인 경우를 방지할 수 있습니다.

2. 정밀도와 재현율 간의 균형 평가 F1 Score는 정밀도와 재현율 사이의 트레이드오프를 평가하는 데 적합합니다. 예를 들어, 정밀도는 높지만 재현율이 낮은 모델이나, 그 반대의 경우에도 F1 Score를 통해 균형 잡힌 성능을 확인할 수 있습니다.

3. 다중 클래스 분류에서의 F1 Score:

- Macro-averaging: 각 클래스별 F1 Score의 평균
- Micro-averaging: 전체 TP, FP, FN을 계산한 후 F1 계산
- Weighted-averaging: 각 클래스의 샘플 수를 고려한 가중 평균

4. 다양한 응용 분야에서의 활용성 F1 Score는 의료, 금융, 정보 검색 등 잘못된 분류로 인한 비용이 큰 분야에서 널리 사용됩니다. 예를 들어, 암 진단에서는 재현율이 중요하지만, 스팸 메일 분류에서는 정밀도가 더 중요할 수 있습니다. F1 Score는 두 지표를 모두 고려하여 최적의 모델을 선택하는 데 도움을 줍니다.

F1 Score의 한계

- 정밀도와 재현율의 상대적 중요도가 다른 경우, F1 Score만으로는 충분한 평가가 어려울 수 있습니다.
- 모델 성능을 비교할 때는 F1 Score 외에도 ROC-AUC, PR 곡선 등 추가적인 지표와 함께 사용하여 신뢰도를 높이는 것이 좋습니다.

## 1.1 개발 환경

Cloud Colab : GPU - A100 40GB / RAM 83.5GB, GPU - L4 22.5GB / RAM 53.5 GB

Local : GPU - 4060Ti 16GB / RAM 96GB

라이브러리 : Pytorch, OpenCV

## 2.1 MoE & Multimodal Train

### -MoE(Mixture of Experts)-

개념:

MoE는 여러 전문가(Expert) 모델을 활용하고, 게이트(Gate) 네트워크를 통해 입력에 가장 적합한 전문가를 동적으로 선택하는 아키텍처입니다. 즉, 하나의 거대한 모델 대신, 각기 다른 특성에 특화된 소규모 모델들(전문가들)의 집합을 통해 문제를 해결하는 방식입니다.

과일 분류 문제에서의 활용: 과일마다 색, 질감, 모양 등의 고유한 특성이 다릅니다. 단일 네트워크가 모든 과일의 특징을 균등하게 잘 학습하기 어렵다면, 과일별 또는 특징 그룹별로 특화된 전문가 모듈을 두어 성능을 개선할 수 있습니다. 예를 들어, 사과 전문가, 바나나 전문가, 딸기 전문가 등으로 나누어 각 과일의 특징(색상, 모양, 질감)에 최적화된 모델을 만들고, 게이트 네트워크가 입력 이미지(또는 멀티모달 입력)에 따라 해당 과일 인식에 가장 적합한 전문가를 활성화하도록 합니다.

### -멀티모달(Multimodal) 학습-

개념:

멀티모달 학습은 이미지, 텍스트, 음성, 또는 이미지 내 색상값·형상 벡터 등 서로 다른 형태의 데이터를 동시에 활용하여 모델을 학습시키는 방법입니다.

과일 분류 문제에서의 활용: 일반적인 과일 이미지 분류는 RGB 이미지만을 사용합니다. 하지만 추가 정보(예: 평균 색상 채널 값, 외형을 숫자 벡터로 추출한 형상 데이터 등)를 함께 입력받으면 모델이 더 풍부한 특징 표현을 학습할 수 있습니다. MNIST처럼 단순화된 형태(실루엣)를 데이터로 만들어, 이미지 특징뿐 아니라 형태 기반의 벡터 데이터를 함께 활용하면 과일의 전반적인 특성을 훨씬 더 잘 파악할 수 있습니다. 이를 통해 단순 픽셀 패턴 인식에 그치지 않고, 색상, 윤곽, 형태 등 다양한 정보를 종합적으로 반영하는 모델을 구현할 수 있습니다.

## 2.2 Simple CNN

개념 및 특징:

가장 기본적인 형태의 합성곱 신경망(CNN)을 사용한 과일 분류 모델입니다. 입력으로 RGB 이미지(과일 사진)를 받고, 여러 합성곱 및 풀링 계층을 거쳐 특징을 추출한 뒤, 완전연결층(Fully Connected Layer)을 통해 클래스별 출력을 생성합니다. 단순 CNN은 모델 구조가 비교적 간단하며, 초기 베이스라인으로 활용하거나 MoE나 멀티모달 등의 더 복잡한 접근법과 성능을 비교하는 기준으로 사용하기에 적합합니다.

## 2.3 Transfer Learning

개념:

전이 학습은 대규모 데이터셋(예: ImageNet)으로 사전 학습된 모델의 가중치를 활용하여, 상대적으로 적은 데이터로도 특정 문제(이 경우 과일 분류)에 최적화하는 기법입니다. 사전 학습된 모델이 이미지 인식에 필요한 일반적인 특징(에지, 질감, 윤곽 등)을 이미 잘 학습하고 있어, 이를 기반으로 특정 클래스(과일 종류) 구분 성능을 더 빠르고 효율적으로 향상시킬 수 있습니다.

### 2.3.1 MobileNet:

모바일 및 임베디드 환경에서도 실시간 추론이 가능하도록 경량화된 CNN 구조입니다. Depthwise Separable Convolution을 통해 계산량을 크게 줄여 파라미터 수와 연산량을 감소시킵니다. 과일 분류 활용: 빠른 추론이 필요하거나 자원이 제한된 환경에서 과일 분류 모델을 배포할 때 효과적입니다.

### 2.3.2 EfficientNet:

특징: 모델의 깊이(Depth), 너비(Width), 해상도(Resolution)를 균형 있게 확장(scale)하여 네트워크 효율성을 극대화한 CNN 계열 모델입니다. 과일 분류 활용: 비슷한 파라미터 규모 내에서 높은 정확도를 제공하며, GPU 메모리 및 연산 효율이 우수합니다. 다양한 과일 이미지를 처리할 때 효율적으로 높은 성능을 발휘할 수 있습니다.

### 2.3.3 ViT (Vision Transformer):

특징: Transformer 구조를 이미지 분류 문제에 적용한 모델입니다. 이미지를 패치(patch) 단위로 나누고 이를 Transformer 입력 토큰으로 변환하여, self-attention 메커니즘을 통해 전역적 이미지 특징을 포착합니다.

과일 분류 활용: 전통적 CNN에 비해 전역적 특징 인식이 강점입니다.

### 2.3.4 ResNet-50:

특징: Residual Connection을 도입하여 깊은 신경망의 학습 어려움(Vanishing Gradient)을 해결한 CNN 구조입니다. 깊은 레이어 구조를 안정적으로 학습하여 ImageNet 등 대규모 이미지 분류 벤치마크에서 우수한 성능을 보입니다.

과일 분류 활용: 전통적이면서도 여전히 강력한 기본 모델로, 전이 학습을 통해 과일 분류 정확도를 빠르게 높일 수 있으며, 다른 최신 모델들과의 성능 비교 기준점(benchmark)으로 활용하기 적합합니다.

요약하면, MoE & 멀티모달 학습은 각 과일에 특화된 전문가 네트워크를 두고, 다양한 형태의 특징(이미지, 색상 통계, 형태 정보 등)을 결합하여 분류 성능을 극대화하는 전략입니다. 이를 보완하거나 비교하기 위해 Simple CNN을 활용한 기본 베이스라인 모델부터, MobileNet, EfficientNet, ViT, ResNet-50과 같은 다양한 전이 학습 기반 모델들을 활용하여 모델 성능과 효율성을 검증할 수 있습니다. 이러한 접근을 통해 궁극적으로 가장 정확하고 안정적인 과일 분류 모델을 개발하고자 했습니다..

### 3.1 데이터 정제 및 증강

본 과제에서는 과일의 컬러 값 범위를 잡아두고 과일과 유사하지 않은 컬러 값을 지우는 즉, 배경을 지우는 선행하여 진행했습니다. 그리고 데이터셋의 클래스 간 불균형 문제를 해결하기 위해 2단계 증강 전략을 적용했습니다. 데이터셋은 단일 과일 클래스(사과, 오렌지, 바나나)가 각각 약 75개의 샘플을 가진 반면, 혼합 클래스는 20개의 샘플만을 보유하고 있어 상당한 불균형이 존재했습니다.

첫 번째 단계에서는 기본적인 기하학적 변환을 통해 클래스 간 샘플 수를 균등하게 맞추는 작업을 수행했습니다. 이 과정에서는 과도한 변형을 피하기 위해 회전( $\pm 10^\circ$ ), 크기 조절(0.9~1.1배), 밝기 조정( $\pm 10$ ) 등 최소한의 변환만을 적용했습니다. 이를 통해 혼합 클래스의 샘플 수를 다른 클래스 수준으로 늘렸습니다.

두 번째 단계에서는 데이터의 다양성을 확보하기 위해 더욱 복잡한 증강 기법을 적용했습니다. 모든 클래스에 대해 기본 변환(회전, 밝기, 대비)과 복합 변환(회전-크기 조절 조합, 밝기-블러 조합, 탄성 변환)을 적용했으며, 특히 혼합 클래스에 대해서는 과일 간 겹침, 부분 가림, 원근 변환 등 클래스 특성을 고려한 추가적인 증강 기법을 사용했습니다.

각각의 증강 기법은 이미지의 본질적인 특성을 유지하면서도 실제 환경에서 발생할 수 있는 다양한 변화를 반영할 수 있도록 설계되었습니다. 특히 혼합 클래스에 적용된 특화 증강 기법들은 여러 과일이 혼합된 실제 상황에서 발생할 수 있는 겹침이나 가림 현상을 시뮬레이션하여 모델의 견고성을 향상시키는 데 중점을 두었습니다.

코드 - 증강함수
<pre>import cv2 import numpy as np import random from scipy.ndimage import gaussian_filter, map_coordinates import os  def augment_with_noise(image):     """이미지에 랜덤 노이즈 추가      Args:         image: 입력 이미지 (numpy array)      Returns:         노이즈가 추가된 이미지      Note:         - 가우시안 노이즈를 생성하여 이미지에 추가         - 노이즈의 표준편차는 5~20 사이에서 랜덤하게 선택     """     # 가우시안 분포를 따르는 랜덤 노이즈 생성 (평균 0, 표준편차 5~20)     noise = np.random.normal(0, random.uniform(5, 20), image.shape).astype(np.uint8)</pre>

```

# 원본 이미지에 노이즈를 추가
noisy_img = cv2.add(image, noise)
return noisy_img

def basic_geometric_augment(image):
    """샘플 수 맞추기용 기본 지오메트릭 변환

    Args:
        image: 입력 이미지 (numpy array)

    Returns:
        변환된 이미지

    Note:
        다음 세 가지 변환 중 하나를 랜덤하게 선택하여 적용:
        1. 회전: -10도 ~ +10도 사이의 최소한의 회전
        2. 스케일: 0.9배 ~ 1.1배 사이의 미세한 크기 조정
        3. 밝기: 0.9배 ~ 1.1배의 대비(alpha)와 -10 ~ +10 사이의 밝기(beta) 조정
    """
    h, w = image.shape[:2]
    center = (w // 2, h // 2) # 이미지 중심점 계산

    transforms = [
        # 회전 변환 (최소한의 각도로 제한)
        lambda img: cv2.warpAffine(
            img,
            cv2.getRotationMatrix2D(center, np.random.uniform(-10, 10), 1.0), # (중심점, 각도, 스케일)
            (w, h) # 출력 이미지 크기
        ),
        # 스케일 변환 (작은 변화로 제한)
        lambda img: cv2.resize(
            img,
            None,
            fx=np.random.uniform(0.9, 1.1), # x축 스케일 팩터
            fy=np.random.uniform(0.9, 1.1) # y축 스케일 팩터
        ),
        # 밝기 변환 (최소 조정으로 제한)
        lambda img: cv2.convertScaleAbs(
            img,
            alpha=np.random.uniform(0.9, 1.1), # 대비 조정
            beta=np.random.randint(-10, 10) # 밝기 조정
        )
    ]

    # 랜덤하게 하나의 변환을 선택하여 적용

```

```

return random.choice(transforms)(image)

def apply_elastic_transform(image, alpha, sigma):
    """안전한 탄성 변환 적용

    Args:
        image: 입력 이미지
        alpha: 변형 강도 파라미터
        sigma: 가우시안 필터의 표준편차 (변형의 부드러움 정도)

    Returns:
        탄성 변형이 적용된 이미지

    Note:
        - 각 픽셀에 대해 랜덤한 변위를 생성하고 가우시안 필터로 스무딩
        - 변위는 이미지 크기의 1/4로 제한하여 과도한 왜곡 방지
        - 각 컬러 채널에 대해 개별적으로 처리
    """
    try:
        result = np.zeros_like(image) # 결과 이미지를 저장할 배열
        random_state = np.random.RandomState(None)

        # 가우시안 필터링된 랜덤 변위 필드 생성
        shape = image.shape[:2]
        dx = gaussian_filter((random_state.rand(*shape) * 2 - 1), sigma) * alpha # x 방향 변위
        dy = gaussian_filter((random_state.rand(*shape) * 2 - 1), sigma) * alpha # y 방향 변위

        # 픽셀 좌표 매트릭스 생성
        x, y = np.meshgrid(np.arange(shape[1]), np.arange(shape[0]))

        # 변위를 이미지 크기의 1/4로 제한하여 과도한 왜곡 방지
        dx = np.clip(dx, -shape[0]//4, shape[0]//4)
        dy = np.clip(dy, -shape[1]//4, shape[1]//4)

        # 변위된 좌표 계산
        indices = np.reshape(y+dy, (-1, 1)), np.reshape(x+dx, (-1, 1))

        # 각 컬러 채널에 대해 개별적으로 변환 적용
        for i in range(3):
            result[..., i] = map_coordinates(image[..., i], indices, order=1).reshape(shape)

        # 픽셀 값을 0-255 범위로 제한
        result = np.clip(result, 0, 255)

    return result.astype(np.uint8)

```



```

except Exception as e:
    print(f"Error in elastic transform: {e}")
    return image # 오류 발생 시 원본 이미지 반환
def simulate_overlap(image):
    """과일 겹침 시뮬레이션

    Args:
        image: 입력 이미지 (numpy array)

    Returns:
        겹침 효과가 적용된 이미지

    Note:
        - 실제 과일이 겹쳐있는 상황을 시뮬레이션
        - 이미지의 중앙부 영역에 반투명한 원형 오버레이를 생성
        - alpha값(0.3~0.7)을 통해 겹침 정도를 랜덤하게 조절
    """
    h, w = image.shape[:2]
    overlay = np.zeros_like(image) # 원본 이미지와 같은 크기의 빈 오버레이 생성

    # 이미지 중앙 영역에 랜덤하게 원형 위치 선택 (가장자리 제외)
    center = (np.random.randint(w//4, 3*w//4), np.random.randint(h//4, 3*h//4))
    # 원의 크기는 이미지 너비의 1/6 ~ 1/3 사이로 설정
    radius = np.random.randint(w//6, w//3)

    # 오버레이에 흰색 원 그리기
    cv2.circle(overlay, center, radius, (255, 255, 255), -1)
    # 0.3~0.7 사이의 랜덤한 투명도 설정
    alpha = np.random.uniform(0.3, 0.7)

    # 원본 이미지와 오버레이를 블렌딩
    return cv2.addWeighted(image, 1, overlay, alpha, 0)

def apply_partial_occlusion(image):
    """부분 가림 효과 적용

    Args:
        image: 입력 이미지 (numpy array)

    Returns:
        부분적으로 가려진 이미지

    Note:
        - 실제 환경에서 발생할 수 있는 부분 가림을 시뮬레이션
    """

```

- 이미지의 랜덤한 위치에 사각형 모양의 가림 영역 생성
- 가림 영역의 크기는 이미지의 1/4에서 1/2 사이로 제한

"""

```
h, w = image.shape[:2]
```

```
mask = np.ones_like(image) # 1로 채워진 마스크 생성
```

```
# 가림 영역의 시작점 (이미지의 좌측 상단 절반 영역 내에서 선택)
```

```
x1, y1 = np.random.randint(0, w//2), np.random.randint(0, h//2)
```

```
# 가림 영역의 크기 (너비와 높이를 1/4에서 1/2 사이로 제한)
```

```
x2, y2 = x1 + np.random.randint(w//4, w//2), y1 + np.random.randint(h//4, h//2)
```

```
# 선택된 영역을 마스크에서 0으로 설정 (완전 가림)
```

```
mask[y1:y2, x1:x2] = 0
```

```
# 마스크를 이미지에 적용
```

```
return image * mask
```

```
def apply_perspective_transform(image):
```

```
    """원근 변환 적용
```

Args:

image: 입력 이미지 (numpy array)

Returns:

원근 변환이 적용된 이미지

Note:

- 이미지에 약간의 3D 효과를 주는 원근 변환 적용
- 네 모서리 포인트를 이미지 크기의 1/8 이내에서 랜덤하게 이동
- 변환의 강도를 제한하여 자연스러운 원근감 생성

"""

```
h, w = image.shape[:2]
```

```
# 원본 이미지의 네 모서리 좌표 설정
```

```
src_points = np.float32([[0, 0], [w, 0], [w, h], [0, h]])
```

```
# 목표 포인트 설정 (각 모서리를 이미지 크기의 1/8 이내에서 랜덤하게 이동)
```

```
dst_points = np.float32([
```

```
    [np.random.randint(0, w//8), np.random.randint(0, h//8)],          # 좌상단
```

```
    [w - np.random.randint(0, w//8), np.random.randint(0, h//8)],      # 우상단
```

```
    [w - np.random.randint(0, w//8), h - np.random.randint(0, h//8)],  # 우하단
```

```
    [np.random.randint(0, w//8), h - np.random.randint(0, h//8)]       # 좌하단
```

```
])
```

```
# 원근 변환 행렬 계산 및 변환 적용
```

```
matrix = cv2.getPerspectiveTransform(src_points, dst_points)
```

```
return cv2.warpPerspective(image, matrix, (w, h))
```

## 코드 - 전처리

```
# 필요한 라이브러리 импорт
from scipy.ndimage import gaussian_filter, map_coordinates
import os, cv2, numpy as np
from pathlib import Path
import shutil
from tqdm import tqdm
import matplotlib.pyplot as plt
import random
from sklearn.model_selection import train_test_split

# 기본 설정
TRAIN_DIR = '/tmp/train_zip/train'      # 원본 학습 데이터 경로
TEST_DIR = '/tmp/test_zip/test'        # 원본 테스트 데이터 경로
PREPROCESS_DIR = 'data/preprocess_data' # 전처리된 데이터 저장 경로
TARGET_SIZE = 256                      # 이미지 크기 표준화 목표 크기

random.seed(42) # 재현성을 위한 랜덤 시드 설정

def create_directories():
    """전처리된 데이터를 저장할 디렉토리 구조 생성

    - 기존 전처리 디렉토리가 있다면 삭제
    - train, validation, test 세 가지 분할을 위한 디렉토리 생성
    """
    if os.path.exists(PREPROCESS_DIR):
        shutil.rmtree(PREPROCESS_DIR)
    os.makedirs(os.path.join(PREPROCESS_DIR, 'train'), exist_ok=True)
    os.makedirs(os.path.join(PREPROCESS_DIR, 'val'), exist_ok=True)
    os.makedirs(os.path.join(PREPROCESS_DIR, 'test'), exist_ok=True)

def remove_background(image):
    """이미지에서 과일 영역을 추출하고 배경 제거

    Args:
        image: 입력 BGR 이미지

    Returns:
        배경이 제거된 이미지

    Note:
        1. HSV 색공간에서 과일 색상 범위에 해당하는 마스크 생성
        2. 엣지 검출을 통한 윤곽선 마스크 생성
        3. 두 마스크를 결합하고 모폴로지 연산으로 정제
```

#### 4. 최종 마스크를 이용해 원본에서 과일 영역만 추출

```
"""
# BGR에서 HSV로 변환
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# 과일 색상 범위에 대한 마스크들
masks = [
    cv2.inRange(hsv, np.array([0, 20, 20]), np.array([15, 255, 255])), # 빨강1
    cv2.inRange(hsv, np.array([165, 20, 20]), np.array([180, 255, 255])), # 빨강2
    cv2.inRange(hsv, np.array([15, 20, 20]), np.array([40, 255, 255])), # 노랑
    cv2.inRange(hsv, np.array([5, 20, 20]), np.array([25, 255, 255])) # 주황
]

# 모든 색상 마스크 통합
color_mask = masks[0]
for mask in masks[1:]:
    color_mask = cv2.bitwise_or(color_mask, mask)

# 엣지 검출을 통한 윤곽선 마스크 생성
edges = cv2.Canny(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY), 100, 200)
final_mask = cv2.bitwise_or(color_mask, edges)

# 모폴로지 연산으로 마스크 정제
kernel = np.ones((5, 5), np.uint8)
final_mask = cv2.morphologyEx(final_mask, cv2.MORPH_CLOSE, kernel) # 작은 구멍 제거
final_mask = cv2.dilate(final_mask, kernel, iterations=1) # 마스크 영역 확장

# 마스크 적용하여 배경 제거
return cv2.bitwise_and(image, image, mask=final_mask)
```

```
def center_on_black(image):
```

```
    """이미지를 검은 배경 중앙에 배치하고 크기 표준화
```

```
    Args:
```

```
        image: 입력 이미지
```

```
    Returns:
```

```
        표준화된 크기(TARGET_SIZE x TARGET_SIZE)의 검은 배경 중앙에 배치된 이미지
```

```
    Note:
```

1. TARGET\_SIZE x TARGET\_SIZE 크기의 검은 배경 생성
2. 입력 이미지의 비율을 유지하면서 크기 조정 (80% 스케일)
3. 조정된 이미지를 검은 배경 중앙에 배치

```
    """
```

```
    # 검은 배경 생성
```

```

black_bg = np.zeros((TARGET_SIZE, TARGET_SIZE, 3), dtype=np.uint8)

# 입력 이미지 크기
h, w = image.shape[:2]

# 비율을 유지하면서 크기 조정 (80% 스케일)
scale = min(TARGET_SIZE / h, TARGET_SIZE / w) * 0.8
new_h, new_w = int(h * scale), int(w * scale)

# 고품질 리사이즈 수행
resized = cv2.resize(image, (new_w, new_h), interpolation=cv2.INTER_LANCZOS4)

# 중앙 배치를 위한 오프셋 계산
y_offset = (TARGET_SIZE - new_h) // 2
x_offset = (TARGET_SIZE - new_w) // 2

# 리사이즈된 이미지를 검은 배경 중앙에 배치
black_bg[y_offset:y_offset + new_h, x_offset:x_offset + new_w] = resized

return black_bg
def split_train_val_files(input_dir):
    """학습 데이터를 학습/검증 세트로 분할

    Args:
        input_dir: 원본 학습 데이터 디렉토리 경로

    Returns:
        train_files: 학습용 파일 리스트
        val_files: 검증용 파일 리스트

    Note:
        - 클래스별로 파일을 그룹화하여 분할 (클래스 밸런스 유지)
        - 각 클래스에서 80%는 학습용, 20%는 검증용으로 분할
        - 재현성을 위해 random_state=42 사용
    """
    class_files = {}
    # 클래스별로 파일 분류
    for filename in os.listdir(input_dir):
        if filename.endswith('.jpg'):
            class_name = filename.split('_')[0] # 파일명의 첫 부분이 클래스명
            if class_name not in class_files:
                class_files[class_name] = []
            class_files[class_name].append(filename)

    train_files = []

```

```

val_files = []

# 클래스별로 train/val 분할
for class_name, files in class_files.items():
    files.sort() # 파일 순서 보장
    train_class_files, val_class_files = train_test_split(
        files, test_size=0.2, random_state=42, shuffle=True
    )
    train_files.extend(train_class_files)
    val_files.extend(val_class_files)

return train_files, val_files

def preprocess_and_select(files, split='train'):
    """데이터 전처리 및 기본 증강 수행

    Args:
        files: 처리할 파일 리스트
        split: 데이터 분할 종류 ('train', 'val', 'test')

    Note:
        1. 클래스별로 파일 분류
        2. 각 이미지에 대해:
            - 배경 제거
            - 크기 표준화 및 중앙 정렬
        3. 학습 데이터의 경우:
            - 클래스별 최대 샘플 수 계산
            - 부족한 클래스는 기본 증강으로 보충
    """
    output_dir = os.path.join(PREPROCESS_DIR, split)
    os.makedirs(output_dir, exist_ok=True)

    # 클래스별 파일 분류
    class_files = {}
    for filename in files:
        class_name = filename.split('_')[0]
        if class_name not in class_files:
            class_files[class_name] = []
        class_files[class_name].append(filename)

    # 학습 데이터의 경우 최대 샘플 수 계산
    if split == 'train':
        max_samples = max(len(files) for files in class_files.values())
        print(f"Maximum samples per class in {split}: {max_samples}")

```

```

# 클래스별 처리
for class_name, class_files_list in class_files.items():
    class_dir = os.path.join(output_dir, class_name)
    os.makedirs(class_dir, exist_ok=True)

    processed_images = []
    # 각 이미지 전처리
    for filename in tqdm(class_files_list, desc=f"Preprocessing {class_name} ({split})", leave=False):
        image_path = os.path.join(TRAIN_DIR, filename)
        image = cv2.imread(image_path)
        if image is None:
            continue
        # 배경 제거 및 표준화
        processed = remove_background(image)
        processed = center_on_black(processed)
        processed_images.append(processed)

    # 학습 데이터의 경우 샘플 수 보충
    if split == 'train':
        current_count = len(processed_images)
        if current_count < max_samples:
            needed = max_samples - current_count
            # 기본 증강으로 부족한 샘플 보충
            for _ in range(needed):
                base_img = random.choice(processed_images)
                processed_images.append(basic_geometric_augment(base_img))

    # 처리된 이미지 저장
    for i, img in enumerate(processed_images):
        save_name = f"{class_name}_{i+1}.jpg"
        cv2.imwrite(os.path.join(class_dir, save_name), img)

def preprocess_test_data():
    """테스트 데이터 전처리"""
    output_dir = os.path.join(PREPROCESS_DIR, 'test')
    os.makedirs(output_dir, exist_ok=True)

    for filename in tqdm(os.listdir(TEST_DIR), desc="Processing test data"):
        if not filename.endswith('.jpg'):
            continue

        class_name = filename.split('_')[0]
        class_dir = os.path.join(output_dir, class_name)
        os.makedirs(class_dir, exist_ok=True)

        image_path = os.path.join(TEST_DIR, filename)

```

```

image = cv2.imread(image_path)
if image is None:
    continue

processed = remove_background(image)
processed = center_on_black(processed)
cv2.imwrite(os.path.join(class_dir, filename), processed)

```

```
def final_augment_image(image, class_name=None):
```

```
    """고급 증강 기법 적용
```

```
    Args:
```

```
        image: 입력 이미지
```

```
        class_name: 클래스 이름 (mixed 클래스 특화 증강을 위해 필요)
```

```
    Returns:
```

```
        augmented_results: (변환명, 변환된 이미지) 튜플의 리스트
```

```
    Note:
```

```
        세 가지 종류의 변환 적용:
```

1. 기본 변환 (회전, 밝기, 대비)
2. 복합 변환 (회전-스케일, 밝기-블러, 탄성)
3. Mixed 클래스 특화 변환 (겹침, 가림, 원근)

```
    """
```

```
    h, w = image.shape[:2]
```

```
    center = (w // 2, h // 2)
```

```
    # 1. 기본 변환 정의 (더 강화된 파라미터)
```

```
    basic_transforms = {
```

```
        'rotation': lambda img: cv2.warpAffine(
            img,
            cv2.getRotationMatrix2D(center, np.random.randint(-45, 45), 1.0),
            (w, h)

```

```
        ),
```

```
        'brightness': lambda img: cv2.convertScaleAbs(
            img,
            alpha=np.random.uniform(0.6, 1.4),
            beta=np.random.randint(-50, 50)

```

```
        ),
```

```
        'contrast': lambda img: cv2.convertScaleAbs(
            img,
            alpha=np.random.uniform(0.5, 1.5)

```

```
        )
```

```
    }
```



# 2. 복합 변환 개선

```
composite_transforms = {
    'rotation_scale': lambda img: cv2.resize(
        cv2.warpAffine(
            img,
            cv2.getRotationMatrix2D(
                (img.shape[1] // 2, img.shape[0] // 2),
                np.random.uniform(-45, 45),
                np.random.uniform(0.8, 1.2)
            ),
            (img.shape[1], img.shape[0])
        ),
        None,
        fx=np.random.uniform(0.9, 1.1),
        fy=np.random.uniform(0.9, 1.1)
    ),
    'brightness_blur': lambda img: cv2.GaussianBlur(
        cv2.convertScaleAbs(
            img,
            alpha=np.random.uniform(0.6, 1.4),
            beta=np.random.randint(-30, 30)
        ),
        (5, 5),
        sigmaX=np.random.uniform(0.5, 1.5)
    ),
    'elastic_transform': lambda img: apply_elastic_transform(
        img,
        alpha=np.random.uniform(20, 50),
        sigma=np.random.uniform(5, 10)
    ),
}
```

# 3. Mixed 클래스 특화 변환

```
mixed_specific_transforms = {
    'overlap_simulation': lambda img: simulate_overlap(img),
    'partial_occlusion': lambda img: apply_partial_occlusion(img),
    'perspective_transform': lambda img: apply_perspective_transform(img)
}
```

augmented\_results = []

# 기본 변환 적용

```
for key in basic_transforms.keys():
    aug_img = basic_transforms[key](image)
    augmented_results.append((f'basic_{key}', aug_img))
```

```

# 복합 변환 적용
for key in composite_transforms.keys():
    aug_img = composite_transforms[key](image)
    augmented_results.append((f'composite_{key}', aug_img))

# Mixed 클래스 특화 변환 적용
if class_name == 'mixed':
    for key in mixed_specific_transforms.keys():
        aug_img = mixed_specific_transforms[key](image)
        augmented_results.append((f'mixed_{key}', aug_img))

return augmented_results

def final_augmentation(split='train'):
    """최종 증강 (학습 데이터만)"""
    if split != 'train':
        return

    train_dir = os.path.join(PREPROCESS_DIR, 'train')
    for class_name in os.listdir(train_dir):
        class_path = os.path.join(train_dir, class_name)
        if not os.path.isdir(class_path):
            continue

        # 원본 이미지만 선택 (class_숫자.jpg 형태)
        images = [f for f in os.listdir(class_path) if f.endswith('.jpg') and len(f.split('_')) == 2]
        images.sort()

        for filename in tqdm(images, desc=f"Final Augmenting {class_name}", leave=False):
            img_path = os.path.join(class_path, filename)
            img = cv2.imread(img_path)
            if img is None:
                continue

            aug_imgs = final_augment_image(img, class_name)
            base_name = os.path.splitext(filename)[0]

            for (transform_name, aimg) in aug_imgs:
                aug_path = os.path.join(class_path, f"{base_name}_aug_{transform_name}.jpg")
                cv2.imwrite(aug_path, aimg)

def plot_examples_for_mixed():
    class_dir = os.path.join(PREPROCESS_DIR, 'train', 'mixed')
    if not os.path.exists(class_dir):

```

```

        print("No mixed class directory found.")
        return
images = [f for f in os.listdir(class_dir) if f.endswith('.jpg') and '_aug' not in f]
if not images:
    print("No images found in mixed class.")
    return
img_path = os.path.join(class_dir, images[0])
img = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
aug_imgs = final_augment_image(img)

plt.figure(figsize=(15, 3))
plt.subplot(1, len(aug_imgs) + 1, 1)
plt.imshow(img_rgb)
plt.title("Original")
plt.axis('off')

for i, (name, aimg) in enumerate(aug_imgs, 1):
    plt.subplot(1, len(aug_imgs) + 1, i+1)
    plt.imshow(cv2.cvtColor(aimg, cv2.COLOR_BGR2RGB))
    plt.title(name)
    plt.axis('off')
plt.show()

if __name__ == "__main__":
    create_directories()

    print("\nSplitting training data into train/val...")
    train_files, val_files = split_train_val_files(TRAIN_DIR)

    print("\nPreprocessing and augmenting training data...")
    preprocess_and_select(train_files, split='train')

    print("\nPreprocessing validation data...")
    preprocess_and_select(val_files, split='val')

    print("\nPreprocessing test data...")
    preprocess_test_data()

    print("\nApplying final augmentation to training data...")
    final_augmentation(split='train')

    # 최종 데이터 수 출력
    for split in ['train', 'val', 'test']:
        split_dir = os.path.join(PREPROCESS_DIR, split)

```

```

for class_name in os.listdir(split_dir):
    class_path = os.path.join(split_dir, class_name)
    if os.path.isdir(class_path):
        count = len([f for f in os.listdir(class_path) if f.endswith('.jpg')])
        print(f"{split.capitalize()} - Class {class_name}: {count} images")

print("\nShowing examples for mixed class final augmentation...")
plot_examples_for_mixed()

```

## 전처리 실행 결과

Preprocessing and augmenting training data...

Maximum samples per class in train: 60

Preprocessing validation data...

Preprocessing test data...

Processing test data: 100% ██████████ 120/120 [00:00<00:00, 230.25it/s]

Applying final augmentation to training data...

Train - Class banana: 420 images

Train - Class orange: 420 images

Train - Class mixed: 600 images

Train - Class apple: 420 images

Val - Class banana: 15 images

Val - Class orange: 15 images

Val - Class mixed: 4 images

Val - Class apple: 15 images

Test - Class banana: 18 images

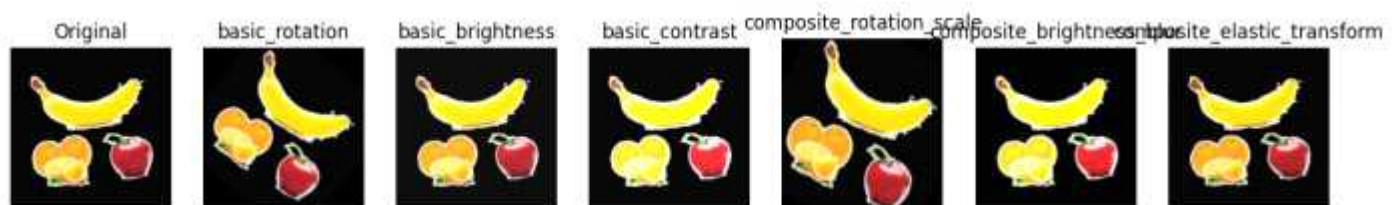
Test - Class orange: 18 images

Test - Class mixed: 5 images

Test - Class apple: 19 images

Showing examples for mixed class final augmentation...

실행 결과 샘플 수가 각각 맞춰지고 증강 결과를 보여주고 있습니다.



## 코드 - for Multimodal extracted feature

```
class FeatureExtractor:
```

```
    """이미지에서 다중 모달 특징을 추출하는 클래스
```

```
    이미지로부터 색상, 텍스처, 형태 관련 특징을 추출하여  
    과일 분류에 사용할 수 있는 특징 벡터를 생성합니다.
```

```
Attributes:
```

```
    input_size (int): 입력 이미지의 크기 (정사각형 가정)
```

```
    label_encoder: 클래스 레이블을 숫자로 변환하는 인코더
```

```
"""
```

```
def __init__(self, input_size=256):
```

```
    self.input_size = input_size
```

```
    self.label_encoder = LabelEncoder()
```

```
def extract_color_features(self, img):
```

```
    """컬러 관련 특징 추출
```

```
Args:
```

```
    img: RGB 형식의 입력 이미지
```

```
Returns:
```

```
    numpy.ndarray: 컬러 특징 벡터
```

```
Note:
```

```
    추출되는 특징:
```

1. 각 채널(R,G,B)별 32개 구간의 정규화된 히스토그램
2. 각 채널별 평균값
3. 각 채널별 표준편차

```
"""
```

```
# 각 채널별 평균과 표준편차 계산
```

```
means = img.mean(axis=(0, 1)) # 각 채널의 평균
```

```
stds = img.std(axis=(0, 1)) # 각 채널의 표준편차
```

```
# 각 채널별 히스토그램 계산
```

```
features = []
```

```
for i in range(3): # RGB 각 채널에 대해
```

```
    hist = cv2.calcHist([img], [i], None, [32], [0, 256]) # 32개 구간으로 히스토그램 계산
```

```
    hist = hist.flatten() / hist.sum() # 정규화
```

```
    features.extend(hist)
```

```
# 모든 특징을 하나의 벡터로 결합
```

```
features.extend(means)
```

```
features.extend(stds)
```

```
return np.array(features)
```

```
def extract_texture_features(self, img):
```

```
    """텍스처 특징 추출
```

```
    Args:
```

```
        img: RGB 형식의 입력 이미지
```

```
    Returns:
```

```
        numpy.ndarray: 텍스처 특징 벡터
```

```
    Note:
```

```
        추출되는 특징:
```

```
        1. Sobel 엣지 검출을 통한 특징
```

- 평균 엣지 강도
- 엣지 강도의 표준편차
- 엣지 강도의 90번째 퍼센타일
- 평균 엣지 방향
- 엣지 방향의 표준편차

```
        2. Local Binary Pattern과 유사한 지역 패턴 분석
```

```
    """
```

```
    # 그레이스케일 변환
```

```
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

```
    # Sobel 엣지 검출 (x, y 방향)
```

```
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
```

```
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
```

```
    # 엣지 강도와 방향 계산
```

```
    magnitude = np.sqrt(sobelx**2 + sobely**2)
```

```
    direction = np.arctan2(sobely, sobelx)
```

```
    # 통계적 특징 추출
```

```
    features = [
```

np.mean(magnitude),	# 평균 엣지 강도
np.std(magnitude),	# 엣지 강도의 표준편차
np.percentile(magnitude, 90),	# 90번째 퍼센타일
np.mean(direction),	# 평균 엣지 방향
np.std(direction),	# 엣지 방향의 표준편차

```
]
```

```
    # 지역 패턴 분석 (LBP와 유사)
```

```
    kernel_size = 3
```

```
    local_mean = cv2.blur(gray, (kernel_size, kernel_size))
```

```
    pattern = (gray > local_mean).astype(np.uint8)
```

```

pattern_hist = cv2.calcHist([pattern], [0], None, [2], [0, 2])
pattern_hist = pattern_hist.flatten() / pattern_hist.sum()

features.extend(pattern_hist)
return np.array(features)
def extract_shape_features(self, img):
    """형태 관련 특징 추출

    Args:
        img: RGB 형식의 입력 이미지

    Returns:
        numpy.ndarray: 형태 특징 벡터 (5차원)

    Note:
        추출되는 특징:
        1. 정규화된 면적 (전체 이미지 크기 대비)
        2. 정규화된 둘레 길이
        3. 정규화된 컨벡스 헐(convex hull) 면적
        4. 원형도 (circularity):  $4\pi \times (\text{면적}/\text{둘레}^2)$ 
        5. 볼록도 (convexity): 실제 면적/컨벡스 헐 면적
        """
    # RGB to 그레이스케일 변환
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # 이진화 (배경이 이미 검은색이므로 낮은 임계값 사용)
    _, binary = cv2.threshold(gray, 10, 255, cv2.THRESH_BINARY)

    # 외곽선 검출
    contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # 컨투어가 없는 경우 0으로 채운 특징 벡터 반환
    if not contours:
        return np.zeros(5)

    # 가장 큰 컨투어 선택 (주요 과일 영역)
    largest_contour = max(contours, key=cv2.contourArea)

    # 기본 측정값 계산
    area = cv2.contourArea(largest_contour)
    perimeter = cv2.arcLength(largest_contour, True)
    hull = cv2.convexHull(largest_contour)
    hull_area = cv2.contourArea(hull)

    # 특징 계산 및 정규화

```

```

features = [
    area / (self.input_size ** 2), # 면적 비율
    perimeter / (4 * self.input_size), # 둘레 비율
    hull_area / (self.input_size ** 2), # 컨벡스 헐 면적 비율
    4 * np.pi * area / (perimeter ** 2) if perimeter > 0 else 0, # 원형도
    area / hull_area if hull_area > 0 else 0 # 볼록도
]

```

```

return np.array(features)

```

```

def process_dataset(self, split='train'):

```

```

    """데이터셋 처리 및 특징 추출

```

```

    Args:

```

```

        split: 데이터셋 분할 유형 ('train', 'val', 'test')

```

```

    Returns:

```

```

        tuple:

```

```

            X: 추출된 특징 행렬

```

```

            y: 인코딩된 레이블

```

```

            image_paths: 처리된 이미지 경로 리스트

```

```

    Note:

```

- 각 이미지에 대해 컬러, 텍스처, 형태 특징을 모두 추출하여 결합
- 학습 데이터의 경우에만 레이블 인코더를 새로 학습

```

    """

```

```

    data_dir = os.path.join(PREPROCESS_DIR, split)

```

```

    features_list = []

```

```

    labels = []

```

```

    image_paths = []

```

```

    # 각 클래스 디렉토리 처리

```

```

    for class_name in os.listdir(data_dir):

```

```

        class_dir = os.path.join(data_dir, class_name)

```

```

        if not os.path.isdir(class_dir):

```

```

            continue

```

```

        print(f"Processing {split} - {class_name}")

```

```

        # 각 이미지 처리

```

```

        for img_name in tqdm(os.listdir(class_dir)):

```

```

            if not img_name.endswith('.jpg'):

```

```

                continue

```

```

            # 이미지 로드 및 RGB 변환

```

```

            img_path = os.path.join(class_dir, img_name)

```



```

img = cv2.imread(img_path)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# 모든 특징 추출 및 결합
color_features = self.extract_color_features(img)
texture_features = self.extract_texture_features(img)
shape_features = self.extract_shape_features(img)

all_features = np.concatenate([
    color_features,
    texture_features,
    shape_features
])

features_list.append(all_features)
labels.append(class_name)
image_paths.append(img_path)

# 특징과 레이블 변환
X = np.array(features_list)
if split == 'train':
    y = self.label_encoder.fit_transform(labels) # 학습 데이터의 경우 인코더 학습
else:
    y = self.label_encoder.transform(labels)      # 검증/테스트의 경우 기존 인코더 사용

return X, y, image_paths
def prepare_data(self):
    """전체 데이터셋 준비 및 저장

    Returns:
        dict: 다음 키를 포함하는 데이터 사전
            - X_train, X_val, X_test: 특징 행렬
            - y_train, y_val, y_test: 레이블
            - train_paths, val_paths, test_paths: 이미지 경로
            - label_encoder: 레이블 인코더 객체

    Note:
        - 학습/검증/테스트 세트 각각에 대해 특징을 추출
        - 추출된 모든 데이터를 pickle 파일로 저장
        - 데이터 크기 및 샘플 수 정보 출력
    """
    # 학습 데이터 처리
    print("Processing training data...")
    X_train, y_train, train_paths = self.process_dataset('train')

```

```

# 검증 데이터 처리
print("Processing validation data...")
X_val, y_val, val_paths = self.process_dataset('val')

# 테스트 데이터 처리
print("Processing test data...")
X_test, y_test, test_paths = self.process_dataset('test')

# 모든 데이터를 하나의 사전에 저장
data = {
    'X_train': X_train,
    'X_val': X_val,
    'X_test': X_test,
    'y_train': y_train,
    'y_val': y_val,
    'y_test': y_test,
    'train_paths': train_paths,
    'val_paths': val_paths,
    'test_paths': test_paths,
    'label_encoder': self.label_encoder
}

# 데이터를 파일로 저장
with open(os.path.join(FEATURE_DIR, 'multimodal_data.pkl'), 'wb') as f:
    pickle.dump(data, f)

# 데이터 크기 정보 출력
print("\nFeature shapes:")
print(f"X_train: {X_train.shape}")
print(f"X_val: {X_val.shape}")
print(f"X_test: {X_test.shape}")

# 샘플 수 정보 출력
print("\nSample counts:")
print(f"Training samples: {len(X_train)}")
print(f"Validation samples: {len(X_val)}")
print(f"Test samples: {len(X_test)}")

return data

# 메인 실행 블록
if __name__ == "__main__":
    """특징 추출 프로세스 실행"""

```

Note:

- FeatureExtractor 인스턴스 생성
- 전체 데이터셋에 대해 특징 추출 수행
- 추출된 특징을 파일로 저장

```
"""
```

```
extractor = FeatureExtractor()
data = extractor.prepare_data()
print("\nFeature extraction completed and saved!")
```

## 특징 추출 실행결과

Processing training data...

Processing train - banana

100% ██████████ 420/420 [00:03<00:00, 113.16it/s]

Processing train - orange

100% ██████████ 420/420 [00:03<00:00, 111.10it/s]

Processing train - mixed

100% ██████████ 600/600 [00:05<00:00, 111.11it/s]

Processing train - apple

100% ██████████ 420/420 [00:03<00:00, 111.45it/s]

Processing validation data...

Processing val - banana

100% ██████████ 15/15 [00:00<00:00, 112.14it/s]

Processing val - orange

100% ██████████ 15/15 [00:00<00:00, 108.79it/s]

Processing val - mixed

100% ██████████ 4/4 [00:00<00:00, 108.67it/s]

Processing val - apple

100% ██████████ 15/15 [00:00<00:00, 110.02it/s]

Processing test data...

Processing test - banana

100%: 18/18 [00:00<00:00, 81.75it/s]

Processing test - orange

100%: 18/18 [00:00<00:00, 109.57it/s]

Processing test - mixed

100%: 5/5 [00:00<00:00, 111.38it/s]

Processing test - apple

100%: 19/19 [00:00<00:00, 111.01it/s]

Feature shapes:

X\_train: (1860, 114)

X\_val: (49, 114)

X\_test: (60, 114)

Sample counts:

Training samples: 1860

Validation samples: 49

Test samples: 60

Feature extraction completed and saved!

## 실험결과

### MoE & Multimodal Train 결과

#### 코드 - MoE & Multimodal Train

```
# 필요한 라이브러리 импорт
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import pickle
import os
import cv2
import numpy as np
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import matplotlib.pyplot as plt
import seaborn as sns
from torchvision import transforms
from tqdm import tqdm
```

```

import warnings

# 경고 메시지 무시
warnings.filterwarnings('ignore')

# Config 클래스 정의: 하이퍼파라미터 및 모델 저장 디렉토리 설정
class Config:
    IMAGE_SIZE = 256 # 입력 이미지 크기
    BATCH_SIZE = 8 # 학습 시 한 번에 처리할 배치 크기
    NUM_WORKERS = 2 # 데이터 로딩 시 병렬 프로세스 개수

    DROPOUT_RATE = 0.4 # 드롭아웃 확률
    HIDDEN_DIM = 256 # 히든 레이어 차원 수
    BN_MOMENTUM = 0.1 # Batch Normalization 모멘텀

    LEARNING_RATE = 0.0001 # 초기 학습률
    NUM_EPOCHS = 150 # 학습 에폭 수
    EARLY_STOP_PATIENCE = 10 # 얼리 스톱 기준: 성능 개선이 없을 때 기다리는 에폭 수
    WEIGHT_DECAY = 0.001 # 가중치 감소 (L2 정규화)

    LABEL_SMOOTHING = 0.1 # 레이블 스무딩 비율
    GRAD_CLIP = 1.0 # 그래디언트 클리핑 값

    WARMUP_EPOCHS = 5 # 워밍업 에폭 수
    USE_SCHEDULER = True # 학습률 스케줄러 사용 여부
    SCHEDULER_PATIENCE = 5 # 스케줄러 참을성
    SCHEDULER_FACTOR = 0.3 # 스케줄러 감소 비율
    MIN_LR = 1e-6 # 스케줄러 최소 학습률

    SPARSITY_WEIGHT = 0.0005 # 희소성 규제 가중치
    SAVE_DIR = 'model_checkpoints' # 모델 저장 디렉토리
    os.makedirs(SAVE_DIR, exist_ok=True)

# 데이터셋 클래스 정의: 이미지 및 특징 데이터를 포함한 데이터셋 로드
class MultiModalDataset(Dataset):
    def __init__(self, images_paths, features, labels, image_size=256):
        self.image_paths = images_paths
        self.features = torch.FloatTensor(features) # 특징 데이터를 텐서로 변환
        self.labels = torch.LongTensor(labels) # 레이블을 텐서로 변환
        self.image_size = image_size

    # 이미지 변환 파이프라인 정의
    self.transform = transforms.Compose([
        transforms.ToTensor(), # 이미지 텐서로 변환
        transforms.Normalize(mean=[0.485, 0.456, 0.406], # 정규화

```

```

        std=[0.229, 0.224, 0.225])

    ])

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

def __getitem__(self, idx):
    # 이미지 로드 및 전처리
    image = cv2.imread(self.image_paths[idx])
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # BGR -> RGB 변환
    image = cv2.resize(image, (self.image_size, self.image_size)) # 크기 조정
    image = self.transform(image) # 변환 적용

    return {
        'image': image, # 전처리된 이미지
        'features': self.features[idx], # 특징 데이터
        'label': self.labels[idx] # 레이블
    }

# Expert 네트워크 정의: 이미지와 특징 데이터를 처리
class Expert(nn.Module):
    def __init__(self, feature_dim, config):
        super().__init__()

    # CNN 블록 정의: 이미지 데이터를 인코딩
    def conv_block(in_c, out_c):
        return nn.Sequential(
            nn.Conv2d(in_c, out_c, 3, padding=1), # Convolutional 레이어
            nn.BatchNorm2d(out_c, momentum=config.BN_MOMENTUM), # Batch Normalization
            nn.ReLU(), # 활성화 함수
            nn.Conv2d(out_c, out_c, 3, padding=1), # 추가 Convolutional 레이어
            nn.BatchNorm2d(out_c, momentum=config.BN_MOMENTUM), # Batch Normalization
            nn.ReLU(),
            nn.MaxPool2d(2), # MaxPooling
            nn.Dropout2d(config.DROPOUT_RATE) # 드롭아웃
        )

    # 이미지 인코더 구성
    self.image_encoder = nn.Sequential(
        conv_block(3, 64),
        conv_block(64, 128),
        conv_block(128, 256),
        conv_block(256, 512),
        nn.AdaptiveAvgPool2d((1, 1)), # 평균 풀링
        nn.Flatten() # 평탄화

```

```

)

# 특징 데이터 인코더 구성
self.feature_encoder = nn.Sequential(
    nn.Linear(feature_dim, config.HIDDEN_DIM * 4), # 특징 차원 확장
    nn.BatchNorm1d(config.HIDDEN_DIM * 4, momentum=config.BN_MOMENTUM),
    nn.ReLU(),
    nn.Dropout(config.DROPOUT_RATE),
    nn.Linear(config.HIDDEN_DIM * 4, config.HIDDEN_DIM * 2),
    nn.BatchNorm1d(config.HIDDEN_DIM * 2, momentum=config.BN_MOMENTUM),
    nn.ReLU(),
    nn.Linear(config.HIDDEN_DIM * 2, config.HIDDEN_DIM),
    nn.BatchNorm1d(config.HIDDEN_DIM, momentum=config.BN_MOMENTUM),
    nn.ReLU()
)

```

```

# 주의 메커니즘 정의: 이미지와 특징 데이터를 결합
self.attention = nn.Sequential(
    nn.Linear(512 + config.HIDDEN_DIM, 256),
    nn.Tanh(),
    nn.Linear(256, 1),
    nn.Sigmoid() # 주의 가중치 계산
)

```

```

# 최종 결합 레이어 구성
self.combined = nn.Sequential(
    nn.Linear(512 + config.HIDDEN_DIM, config.HIDDEN_DIM * 4),
    nn.BatchNorm1d(config.HIDDEN_DIM * 4, momentum=config.BN_MOMENTUM),
    nn.ReLU(),
    nn.Dropout(config.DROPOUT_RATE),
    nn.Linear(config.HIDDEN_DIM * 4, config.HIDDEN_DIM * 2),
    nn.BatchNorm1d(config.HIDDEN_DIM * 2, momentum=config.BN_MOMENTUM),
    nn.ReLU(),
    nn.Linear(config.HIDDEN_DIM * 2, 1) # 최종 출력
)

```

```

def forward(self, images, features):
    img_feat = self.image_encoder(images) # 이미지 특징 인코딩
    num_feat = self.feature_encoder(features) # 숫자 특징 인코딩

    # 이미지와 특징 데이터 결합 및 주의 가중치 계산
    combined = torch.cat([img_feat, num_feat], dim=1)
    attention_weights = self.attention(combined)
    attended_features = combined * attention_weights

```

```

# 최종 출력
score = self.combined(attended_features)
return score

# Focal Loss 정의: 불균형 클래스 문제를 해결하기 위한 손실 함수
class FocalLoss(nn.Module):
    def __init__(self, alpha=1, gamma=2, class_weights=None):
        super().__init__()
        self.alpha = alpha
        self.gamma = gamma
        self.class_weights = torch.tensor([1.0, 1.0, 3.0, 1.0]) if class_weights is None else class_weights

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(
            inputs, targets,
            weight=self.class_weights.to(inputs.device),
            label_smoothing=Config.LABEL_SMOOTHING, # 레이블 스무딩 적용
            reduction='none'
        )
        pt = torch.exp(-ce_loss) # 예측 확률
        focal_loss = self.alpha * (1 - pt)**self.gamma * ce_loss # Focal Loss 계산
        return focal_loss.mean()

# EarlyStopping 클래스 정의: 학습 중 조기 종료를 위한 로직
class EarlyStopping:
    def __init__(self, patience=7, min_delta=0, path='best_model.pth'):
        self.patience = patience # 개선되지 않는 에폭 수 허용치
        self.min_delta = min_delta # 개선이라고 간주할 최소 변화량
        self.path = path # 모델 저장 경로
        self.counter = 0 # 개선되지 않은 에폭 수
        self.best_loss = None # 현재까지의 최저 손실값
        self.early_stop = False # 조기 종료 플래그

    def __call__(self, val_loss, model):
        if self.best_loss is None:
            # 첫 에폭에서 최적 손실값 초기화 및 모델 저장
            self.best_loss = val_loss
            self.save_checkpoint(model)
        elif val_loss > self.best_loss + self.min_delta:
            # 손실값이 개선되지 않을 경우 카운터 증가
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True # 조기 종료 플래그 설정
        else:
            # 손실값이 개선되었을 경우 업데이트 및 카운터 리셋
            self.best_loss = val_loss

```



```

        self.save_checkpoint(model)
        self.counter = 0

def save_checkpoint(self, model):
    # 모델의 상태 저장
    torch.save(model.state_dict(), self.path)

# MoEMultiModal 클래스 정의: Mixture of Experts (MoE) 기반 멀티모달 모델
class MoEMultiModal(nn.Module):
    def __init__(self, num_classes, feature_dim, config):
        super().__init__()
        self.num_classes = num_classes

        # 클래스별 전문가 네트워크 생성
        self.experts = nn.ModuleList([
            Expert(feature_dim, config) for _ in range(num_classes)
        ])

        # 게이트 네트워크 정의: 입력 특징에 따라 전문가 선택 가중치 계산
        self.gate = nn.Sequential(
            nn.Linear(feature_dim, config.HIDDEN_DIM * 2),
            nn.BatchNorm1d(config.HIDDEN_DIM * 2, momentum=config.BN_MOMENTUM),
            nn.ReLU(),
            nn.Dropout(config.DROPOUT_RATE),
            nn.Linear(config.HIDDEN_DIM * 2, config.HIDDEN_DIM),
            nn.BatchNorm1d(config.HIDDEN_DIM, momentum=config.BN_MOMENTUM),
            nn.ReLU(),
            nn.Linear(config.HIDDEN_DIM, num_classes),
            nn.Softmax(dim=1) # 클래스별 선택 확률 계산
        )

    def forward(self, images, features):
        # 게이트 네트워크에서 전문가 선택 가중치 계산
        gate_weights = self.gate(features)

        # 전문가 네트워크의 출력 계산
        expert_outputs = []
        for expert in self.experts:
            out = expert(images, features)
            expert_outputs.append(out)

        expert_outputs = torch.cat(expert_outputs, dim=1) # 전문가 출력 결합
        final_output = expert_outputs * gate_weights # 가중치 적용

        return final_output

```

```

# 한 에폭 동안 모델 평가 함수
def evaluate_one_epoch(model, data_loader, criterion, device):
    model.eval() # 평가 모드로 전환
    total_loss = 0.0
    all_preds = [] # 예측값 저장
    all_labels = [] # 실제 레이블 저장

    with torch.no_grad():
        for batch in data_loader:
            images = batch['image'].to(device) # 이미지 데이터 로드
            features = batch['features'].to(device) # 특징 데이터 로드
            labels = batch['label'].to(device) # 레이블 로드

            outputs = model(images, features) # 모델 출력
            loss = criterion(outputs, labels) # 손실 계산
            total_loss += loss.item()

            _, preds = torch.max(outputs, 1) # 예측값 계산
            all_preds.extend(preds.cpu().numpy()) # 예측값 저장
            all_labels.extend(labels.cpu().numpy()) # 실제 레이블 저장

    avg_loss = total_loss / len(data_loader) # 평균 손실값 계산
    accuracy = accuracy_score(all_labels, all_preds) # 정확도 계산
    f1 = f1_score(all_labels, all_preds, average='macro') # F1 점수 계산

    return avg_loss, accuracy, f1

# 학습률 조정을 위한 워밍업 스케줄 계산
def get_lr_multiplier(epoch, warmup_epochs):
    if epoch < warmup_epochs:
        return (epoch + 1) / warmup_epochs
    return 1.0

# MoE 모델 학습 함수
def train_moe_model(model, train_loader, val_loader, criterion, optimizer, config, device):
    early_stopping = EarlyStopping(
        patience=config.EARLY_STOP_PATIENCE,
        path=os.path.join(config.SAVE_DIR, 'best_moe_model.pth')
    )

    # 학습률 스케줄러 설정
    if config.USE_SCHEDULER:
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer, mode='min', patience=config.SCHEDULER_PATIENCE,

```

```

        factor=config.SCHEDULER_FACTOR, min_lr=config.MIN_LR, verbose=True
    )

# 학습 및 검증 성능 기록을 위한 리스트 초기화
train_losses, train_accs, train_f1s = [], [], []
val_losses, val_accs, val_f1s = [], [], []

for epoch in range(config.NUM_EPOCHS):
    # 학습률 워밍업 적용
    if epoch < config.WARMUP_EPOCHS:
        lr_multiplier = get_lr_multiplier(epoch, config.WARMUP_EPOCHS)
        for param_group in optimizer.param_groups:
            param_group['lr'] = config.LEARNING_RATE * lr_multiplier

    # 학습 모드
    model.train()
    train_loss = 0.0
    train_preds = []
    train_labels = []

    for batch in tqdm(train_loader, desc=f'Epoch {epoch+1}/{config.NUM_EPOCHS}'):
        images = batch['image'].to(device) # 이미지 데이터
        features = batch['features'].to(device) # 특징 데이터
        labels = batch['label'].to(device) # 레이블 데이터

        optimizer.zero_grad() # 그래디언트 초기화
        outputs = model(images, features) # 모델 출력
        loss = criterion(outputs, labels) # 손실 계산

        # 게이트 희소성 규제 추가
        gate_sparsity = 0.0
        for expert in model.experts:
            gate_sparsity += torch.mean(torch.abs(expert.combined[-1].weight))

        total_loss = loss + config.SPARSITY_WEIGHT * gate_sparsity # 총 손실 계산
        total_loss.backward() # 역전파

        # 그래디언트 클리핑
        torch.nn.utils.clip_grad_norm_(model.parameters(), config.GRAD_CLIP)
        optimizer.step() # 가중치 업데이트

    train_loss += loss.item()
    _, preds = torch.max(outputs, 1)
    train_preds.extend(preds.cpu().numpy())
    train_labels.extend(labels.cpu().numpy())

```

```

train_loss /= len(train_loader)
train_acc = accuracy_score(train_labels, train_preds)
train_f1 = f1_score(train_labels, train_preds, average='macro')

# 검증 단계
val_loss, val_acc, val_f1 = evaluate_one_epoch(model, val_loader, criterion, device)

# 성능 기록
train_losses.append(train_loss)
train_accs.append(train_acc)
train_f1s.append(train_f1)
val_losses.append(val_loss)
val_accs.append(val_acc)
val_f1s.append(val_f1)

# 진행 상태 출력
print(f'\nEpoch {epoch+1}/{config.NUM_EPOCHS}:')
print(f'Train - Loss: {train_loss:.4f}, Acc: {train_acc:.4f}, F1: {train_f1:.4f}')
print(f'Val   - Loss: {val_loss:.4f}, Acc: {val_acc:.4f}, F1: {val_f1:.4f}')

if config.USE_SCHEDULER:
    scheduler.step(val_loss)

early_stopping(val_loss, model)
if early_stopping.early_stop:
    print("\nEarly stopping triggered")
    break

# 학습 곡선 저장
plot_metrics(train_losses, val_losses, 'Loss', os.path.join(config.SAVE_DIR, 'loss_curves.png'))
plot_metrics(train_accs, val_accs, 'Accuracy', os.path.join(config.SAVE_DIR, 'accuracy_curves.png'))
plot_metrics(train_f1s, val_f1s, 'F1 Score', os.path.join(config.SAVE_DIR, 'f1_curves.png'))

```

# 혼동 행렬 시각화 함수

```

def plot_confusion_matrix(true_labels, pred_labels, class_names, save_path):
    cm = confusion_matrix(true_labels, pred_labels)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.savefig(save_path)

```

```

plt.close()

# 학습 곡선 시각화 함수
def plot_metrics(train_metrics, val_metrics, metric_name, save_path):
    plt.figure(figsize=(10, 5))
    plt.plot(train_metrics, label=f'Train {metric_name}')
    plt.plot(val_metrics, label=f'Val {metric_name}')
    plt.xlabel('Epoch')
    plt.ylabel(metric_name)
    plt.legend()
    plt.title(f'{metric_name} over Training')
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

# 클래스 가중치 계산 함수
def calculate_class_weights(y_train):
    class_counts = np.bincount(y_train)
    total = len(y_train)
    weights = total / (len(class_counts) * class_counts)
    return torch.FloatTensor(weights)

# 메인 함수
def main():
    torch.cuda.empty_cache() # CUDA 메모리 초기화
    config = Config() # Config 인스턴스 생성

    # 데이터 로드
    with open(os.path.join(FEATURE_DIR, 'multimodal_data.pkl'), 'rb') as f:
        data = pickle.load(f)

    # 클래스 가중치 계산
    class_weights = calculate_class_weights(data['y_train'])

    # 데이터셋 및 데이터로더 설정
    train_dataset = MultiModalDataset(
        data['train_paths'], data['X_train'], data['y_train'],
        image_size=config.IMAGE_SIZE
    )
    val_dataset = MultiModalDataset(
        data['val_paths'], data['X_val'], data['y_val'],
        image_size=config.IMAGE_SIZE
    )
    test_dataset = MultiModalDataset(
        data['test_paths'], data['X_test'], data['y_test'],

```

```

        image_size=config.IMAGE_SIZE
    )

    train_loader = DataLoader(
        train_dataset, batch_size=config.BATCH_SIZE,
        shuffle=True, num_workers=config.NUM_WORKERS
    )
    val_loader = DataLoader(
        val_dataset, batch_size=config.BATCH_SIZE,
        shuffle=False, num_workers=config.NUM_WORKERS
    )
    test_loader = DataLoader(
        test_dataset, batch_size=config.BATCH_SIZE,
        shuffle=False, num_workers=config.NUM_WORKERS
    )

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = MoEMultiModal(
        num_classes=len(data['label_encoder'].classes_),
        feature_dim=data['X_train'].shape[1],
        config=config
    ).to(device)

    criterion = FocalLoss(gamma=2, class_weights=class_weights) # Focal Loss 설정
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=config.LEARNING_RATE,
        weight_decay=config.WEIGHT_DECAY
    )

    # 모델 학습
    print("Starting training...")
    train_moe_model(
        model=model,
        train_loader=train_loader,
        val_loader=val_loader,
        criterion=criterion,
        optimizer=optimizer,
        config=config,
        device=device
    )

    # 최적 모델 로드 및 최종 평가
    print("\nLoading best model for final evaluation...")
    model.load_state_dict(torch.load(os.path.join(config.SAVE_DIR, 'best_moe_model.pth')))

```

```

print("\nValidation Set Performance:")
val_loss, val_acc, val_f1 = evaluate_one_epoch(model, val_loader, criterion, device)
print(f"Loss: {val_loss:.4f}, Accuracy: {val_acc:.4f}, F1 Score: {val_f1:.4f}")

print("\nTest Set Performance:")
test_loss, test_acc, test_f1 = evaluate_one_epoch(model, test_loader, criterion, device)
print(f"Loss: {test_loss:.4f}, Accuracy: {test_acc:.4f}, F1 Score: {test_f1:.4f}")

# 테스트 세트에서 세부 분류 결과 출력
all_preds = []
all_labels = []
model.eval()
with torch.no_grad():
    for batch in test_loader:
        images = batch['image'].to(device)
        features = batch['features'].to(device)
        labels = batch['label'].to(device)

        outputs = model(images, features)
        _, preds = torch.max(outputs, 1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

print("\nDetailed Classification Report:")
print(classification_report(all_labels, all_preds,
                           target_names=data['label_encoder'].classes_))

# 혼동 행렬 저장
plot_confusion_matrix(
    all_labels, all_preds,
    data['label_encoder'].classes_,
    os.path.join(config.SAVE_DIR, 'confusion_matrix.png')
)

# 스크립트 실행
if __name__ == "__main__":
    main()


```


## MoE & Multimodal Train 결과


Starting training...


Epoch 1/150: 100% 210/210 [00:20<00:00, 10.48it/s]


Epoch 1/150:


Train - Loss: 0.7316, Acc: 0.3881, F1: 0.3835  
Val - Loss: 0.6823, Acc: 0.4694, F1: 0.3588  
Epoch 2/150: 100% 210/210 [00:20<00:00, 10.49it/s]


Epoch 2/150:  
Train - Loss: 0.5544, Acc: 0.5893, F1: 0.5759  
Val - Loss: 0.3361, Acc: 0.7347, F1: 0.6491  
Epoch 3/150: 100% 210/210 [00:19<00:00, 10.53it/s]


Epoch 3/150:  
Train - Loss: 0.4155, Acc: 0.6976, F1: 0.6954  
Val - Loss: 0.2464, Acc: 0.8367, F1: 0.7309  
Epoch 4/150: 100% 210/210 [00:19<00:00, 10.51it/s]


Epoch 4/150:  
Train - Loss: 0.3729, Acc: 0.7262, F1: 0.7245  
Val - Loss: 0.1736, Acc: 0.8776, F1: 0.7702  
Epoch 5/150: 100% 210/210 [00:19<00:00, 10.53it/s]

Epoch 5/150:  
Train - Loss: 0.3302, Acc: 0.7732, F1: 0.7727  
Val - Loss: 0.1673, Acc: 0.9184, F1: 0.8120  
Epoch 6/150: 100% 210/210 [00:19<00:00, 10.51it/s]

Epoch 6/150:  
Train - Loss: 0.3058, Acc: 0.7935, F1: 0.7925  
Val - Loss: 0.1598, Acc: 0.8980, F1: 0.7970  
Epoch 7/150: 100% 210/210 [00:19<00:00, 10.54it/s]


Epoch 7/150:  
Train - Loss: 0.3060, Acc: 0.8071, F1: 0.8073  
Val - Loss: 0.1393, Acc: 0.9184, F1: 0.8520  
Epoch 8/150: 100% 210/210 [00:19<00:00, 10.53it/s]


Epoch 8/150:  
Train - Loss: 0.2857, Acc: 0.8202, F1: 0.8200  
Val - Loss: 0.1527, Acc: 0.9184, F1: 0.8890  
Epoch 9/150: 100% 210/210 [00:19<00:00, 10.55it/s]


Epoch 9/150:  
Train - Loss: 0.2471, Acc: 0.8351, F1: 0.8350  
Val - Loss: 0.1643, Acc: 0.8776, F1: 0.8106  
Epoch 10/150: 100% 210/210 [00:19<00:00, 10.53it/s]


Epoch 10/150:





Train - Loss: 0.2595, Acc: 0.8488, F1: 0.8488  
Val - Loss: 0.1475, Acc: 0.8980, F1: 0.8249  
Epoch 11/150: 100%| 210/210 [00:19<00:00, 10.51it/s]


Epoch 11/150:  
Train - Loss: 0.2334, Acc: 0.8506, F1: 0.8505  
Val - Loss: 0.1155, Acc: 0.9388, F1: 0.9042  
Epoch 12/150: 100%| 210/210 [00:19<00:00, 10.55it/s]


Epoch 12/150:  
Train - Loss: 0.2375, Acc: 0.8583, F1: 0.8584  
Val - Loss: 0.1668, Acc: 0.9184, F1: 0.8682  
Epoch 13/150: 100%| 210/210 [00:19<00:00, 10.53it/s]


Epoch 13/150:  
Train - Loss: 0.2272, Acc: 0.8619, F1: 0.8616  
Val - Loss: 0.1156, Acc: 0.9388, F1: 0.8844  
Epoch 14/150: 100%| 210/210 [00:19<00:00, 10.53it/s]

Epoch 14/150:  
Train - Loss: 0.2200, Acc: 0.8726, F1: 0.8727  
Val - Loss: 0.1378, Acc: 0.9388, F1: 0.8844  
Epoch 15/150: 100%| 210/210 [00:19<00:00, 10.53it/s]


Epoch 15/150:  
Train - Loss: 0.2040, Acc: 0.8964, F1: 0.8965  
Val - Loss: 0.1134, Acc: 0.9184, F1: 0.8520  
Epoch 16/150: 100%| 210/210 [00:19<00:00, 10.53it/s]

Epoch 16/150:  
Train - Loss: 0.1978, Acc: 0.8881, F1: 0.8882  
Val - Loss: 0.1412, Acc: 0.9388, F1: 0.8844  
Epoch 17/150: 100%| 210/210 [00:19<00:00, 10.53it/s]


Epoch 17/150:  
Train - Loss: 0.2084, Acc: 0.8917, F1: 0.8916  
Val - Loss: 0.1200, Acc: 0.9184, F1: 0.8107  
Epoch 18/150: 100%| 210/210 [00:19<00:00, 10.50it/s]

Epoch 18/150:  
Train - Loss: 0.1756, Acc: 0.9030, F1: 0.9030  
Val - Loss: 0.1137, Acc: 0.9388, F1: 0.8844  
Epoch 19/150: 100%| 210/210 [00:20<00:00, 10.47it/s]


Epoch 19/150:

Train - Loss: 0.2037, Acc: 0.8964, F1: 0.8960  
Val - Loss: 0.1387, Acc: 0.9388, F1: 0.8844  
Epoch 20/150: 100%| 210/210 [00:20<00:00, 10.46it/s]


Epoch 20/150:

Train - Loss: 0.1745, Acc: 0.9125, F1: 0.9126  
Val - Loss: 0.1045, Acc: 0.9592, F1: 0.9396  
Epoch 21/150: 100%| 210/210 [00:20<00:00, 10.45it/s]


Epoch 21/150:

Train - Loss: 0.1719, Acc: 0.9119, F1: 0.9120  
Val - Loss: 0.1466, Acc: 0.9184, F1: 0.8520  
Epoch 22/150: 100%| 210/210 [00:19<00:00, 10.50it/s]


Epoch 22/150:

Train - Loss: 0.1857, Acc: 0.9119, F1: 0.9121  
Val - Loss: 0.1298, Acc: 0.9388, F1: 0.8844  
Epoch 23/150: 100%| 210/210 [00:20<00:00, 10.48it/s]


Epoch 23/150:

Train - Loss: 0.1590, Acc: 0.9196, F1: 0.9199  
Val - Loss: 0.1228, Acc: 0.9388, F1: 0.8844  
Epoch 24/150: 100%| 210/210 [00:19<00:00, 10.53it/s]


Epoch 24/150:

Train - Loss: 0.1630, Acc: 0.9220, F1: 0.9219  
Val - Loss: 0.1264, Acc: 0.9388, F1: 0.8844  
Epoch 25/150: 100%| 210/210 [00:19<00:00, 10.52it/s]


Epoch 25/150:

Train - Loss: 0.1444, Acc: 0.9262, F1: 0.9262  
Val - Loss: 0.0932, Acc: 0.9388, F1: 0.8844  
Epoch 26/150: 100%| 210/210 [00:19<00:00, 10.50it/s]


Epoch 26/150:

Train - Loss: 0.1642, Acc: 0.9179, F1: 0.9179  
Val - Loss: 0.1136, Acc: 0.9388, F1: 0.8844  
Epoch 27/150: 100%| 210/210 [00:19<00:00, 10.52it/s]


Epoch 27/150:

Train - Loss: 0.1493, Acc: 0.9274, F1: 0.9273  
Val - Loss: 0.1471, Acc: 0.9184, F1: 0.8102  
Epoch 28/150: 100%| 210/210 [00:19<00:00, 10.53it/s]


Epoch 28/150:

Train - Loss: 0.1391, Acc: 0.9375, F1: 0.9376  
Val - Loss: 0.1760, Acc: 0.9184, F1: 0.8520  
Epoch 29/150: 100%| 210/210 [00:19<00:00, 10.51it/s]


Epoch 29/150:

Train - Loss: 0.1536, Acc: 0.9280, F1: 0.9280  
Val - Loss: 0.1086, Acc: 0.9592, F1: 0.9005  
Epoch 30/150: 100%| 210/210 [00:19<00:00, 10.52it/s]


Epoch 30/150:

Train - Loss: 0.1422, Acc: 0.9333, F1: 0.9333  
Val - Loss: 0.1334, Acc: 0.9388, F1: 0.8844  
Epoch 31/150: 100%| 210/210 [00:19<00:00, 10.52it/s]


Epoch 31/150:

Train - Loss: 0.1895, Acc: 0.9268, F1: 0.9271  
Val - Loss: 0.1454, Acc: 0.9388, F1: 0.8844  
Epoch 32/150: 100%| 210/210 [00:20<00:00, 10.49it/s]


Epoch 32/150:

Train - Loss: 0.1381, Acc: 0.9381, F1: 0.9383  
Val - Loss: 0.1402, Acc: 0.9388, F1: 0.8844  
Epoch 33/150: 100%| 210/210 [00:19<00:00, 10.53it/s]

Epoch 33/150:

Train - Loss: 0.1244, Acc: 0.9488, F1: 0.9490  
Val - Loss: 0.1341, Acc: 0.9388, F1: 0.8844  
Epoch 34/150: 100%| 210/210 [00:19<00:00, 10.53it/s]

Epoch 34/150:

Train - Loss: 0.1279, Acc: 0.9470, F1: 0.9471  
Val - Loss: 0.1235, Acc: 0.9388, F1: 0.8844  
Epoch 35/150: 100%| 210/210 [00:19<00:00, 10.51it/s]

Epoch 35/150:

Train - Loss: 0.1264, Acc: 0.9446, F1: 0.9447  
Val - Loss: 0.1247, Acc: 0.9388, F1: 0.8844

Early stopping triggered

Loading best model for final evaluation...

Validation Set Performance:

Loss: 0.0932, Accuracy: 0.9388, F1 Score: 0.8844

Test Set Performance:

Loss: 0.1822, Accuracy: 0.9000, F1 Score: 0.8216

Detailed Classification Report:

	precision	recall	f1-score	support
apple	0.90	0.95	0.92	19
banana	0.89	0.94	0.92	18
mixed	0.67	0.40	0.50	5
orange	0.94	0.94	0.94	18
accuracy			0.90	60
macro avg	0.85	0.81	0.82	60
weighted avg	0.89	0.90	0.89	60

세 개의 명확한 클래스에서는 준수한 성능을 보이는 반면 Mixed 클래스에서 성능이 확연히 떨어지는 모습을 보입니다.

## 코드 - Simple CNN Train

```
# 필요한 라이브러리 импорт
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision.transforms import Compose, Resize, ToTensor, Normalize
from PIL import Image
import os
import numpy as np
from tqdm import tqdm
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt

# 기본 설정
BATCH_SIZE = 32 # 배치 크기
NUM_EPOCHS = 10 # 학습 에폭 수
LEARNING_RATE = 2e-4 # 학습률
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # 디바이스 설정
PREPROCESS_DIR = 'data/preprocess_data' # 전처리된 데이터 경로
PATIENCE = 5 # Early stopping patience

# Custom CNN 모델 정의
class SimplifiedCNN(nn.Module):
    """간단한 CNN 모델로 이미지를 분류."""
    def __init__(self, num_classes):
        super(SimplifiedCNN, self).__init__()
```

```

# Convolutional Layers
self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
self.pool = nn.MaxPool2d(2, 2) # Max Pooling
self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
# Fully Connected Layers
self.fc1 = nn.Linear(32 * 56 * 56, 64) # 첫 번째 FC Layer
self.fc2 = nn.Linear(64, num_classes) # 두 번째 FC Layer

def forward(self, x):
    # Convolution + ReLU + Pooling
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(x.size(0), -1) # Flatten
    x = F.relu(self.fc1(x)) # Fully Connected Layer 1
    x = self.fc2(x) # Fully Connected Layer 2 (출력)
    return x

# 데이터셋 클래스 정의
class CustomImageDataset(Dataset):
    """이미지 데이터셋을 정의. 각 이미지와 레이블 반환."""
    def __init__(self, root_dir, split='train'):
        self.root_dir = os.path.join(root_dir, split) # 데이터 분할 경로
        self.image_paths = [] # 이미지 경로 리스트
        self.labels = [] # 레이블 리스트
        self.class_to_idx = {} # 클래스 이름과 인덱스 매핑

        # 클래스별로 이미지 경로 및 레이블 수집
        for idx, class_name in enumerate(sorted(os.listdir(self.root_dir))):
            self.class_to_idx[class_name] = idx
            class_dir = os.path.join(self.root_dir, class_name)
            if os.path.isdir(class_dir):
                for img_name in os.listdir(class_dir):
                    if img_name.endswith('.jpg'):
                        self.image_paths.append(os.path.join(class_dir, img_name))
                        self.labels.append(idx)

# 데이터 변환 파이프라인 정의
self.transform = Compose([
    Resize((224, 224)), # 이미지 크기 조정
    ToTensor(), # 텐서로 변환
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # 정규화
])

def __len__(self):
    return len(self.image_paths) # 데이터셋 크기 반환

```

```

def __getitem__(self, idx):
    img_path = self.image_paths[idx]
    image = Image.open(img_path).convert('RGB') # 이미지 로드 및 RGB로 변환
    image = self.transform(image) # 변환 적용
    label = self.labels[idx] # 레이블
    return image, label

# 학습 루프
def train_one_epoch(model, train_loader, optimizer, criterion, device):
    """한 에폭 동안 모델 학습."""
    model.train()
    total_loss = 0
    all_preds = []
    all_labels = []

    progress_bar = tqdm(train_loader, desc='Training')
    for images, labels in progress_bar:
        images, labels = images.to(device), labels.to(device)

        # 옵티마이저 초기화 및 순전파
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward() # 역전파
        optimizer.step() # 매개변수 업데이트

        total_loss += loss.item()
        preds = torch.argmax(outputs, dim=1).cpu().numpy()
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy())
        progress_bar.set_postfix({'loss': loss.item()})

    epoch_loss = total_loss / len(train_loader)
    epoch_acc = accuracy_score(all_labels, all_preds)
    return epoch_loss, epoch_acc

# 검증 루프
def evaluate(model, val_loader, criterion, device):
    """모델 평가."""
    model.eval()
    total_loss = 0
    all_preds = []
    all_labels = []

```

```

with torch.no_grad():
    for images, labels in tqdm(val_loader, desc='Evaluating'):
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        total_loss += loss.item()
        preds = torch.argmax(outputs, dim=1).cpu().numpy()
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy())

val_loss = total_loss / len(val_loader)
val_acc = accuracy_score(all_labels, all_preds)
val_f1 = f1_score(all_labels, all_preds, average='weighted')
return val_loss, val_acc, val_f1, all_preds, all_labels

# Early Stopping 클래스 정의
class EarlyStopping:
    """검증 손실 개선 여부에 따라 학습 중단."""
    def __init__(self, patience=7, min_delta=0, path='checkpoint.pth'):
        self.patience = patience
        self.min_delta = min_delta
        self.path = path
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        if self.best_loss is None or val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.save_checkpoint(model)
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True

    def save_checkpoint(self, model):
        """최고 성능 모델 저장."""
        torch.save(model.state_dict(), self.path)

# 메인 함수
def main():
    print(f"Using device: {DEVICE}")

    # 데이터셋 및 데이터 로더 초기화

```

```

train_dataset = CustomImageDataset(PREPROCESS_DIR, split='train')
val_dataset = CustomImageDataset(PREPROCESS_DIR, split='val')
test_dataset = CustomImageDataset(PREPROCESS_DIR, split='test')

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

# 모델 초기화
num_classes = len(os.listdir(os.path.join(PREPROCESS_DIR, 'train')))
model = SimplifiedCNN(num_classes).to(DEVICE)

# 손실 함수 및 옵티마이저 정의
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
early_stopping = EarlyStopping(patience=PATIENCE, path='best_model.pth')

# 학습 루프
for epoch in range(NUM_EPOCHS):
    print(f"\nEpoch {epoch+1}/{NUM_EPOCHS}")
    train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion, DEVICE)
    val_loss, val_acc, val_f1, _, _ = evaluate(model, val_loader, criterion, DEVICE)

    print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
    print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}, Val F1: {val_f1:.4f}")

    early_stopping(val_loss, model)
    if early_stopping.early_stop:
        print("Early stopping triggered")
        break

# 테스트
model.load_state_dict(torch.load('best_model.pth'))
test_loss, test_acc, test_f1, _, _ = evaluate(model, test_loader, criterion, DEVICE)
print(f"Test Loss: {test_loss:.4f}, Test Acc: {test_acc:.4f}, Test F1: {test_f1:.4f}")

if __name__ == "__main__":
    main()

```

## Simple CNN Train 결과

Using device: cuda

Epoch 1/10



Training: 100% ██████████ 59/59 [00:03<00:00, 18.27it/s, loss=0.598]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 8.31it/s]

Train Loss: 0.9495, Train Acc: 0.6414

Val Loss: 0.6973, Val Acc: 0.7959, Val F1: 0.7904

Epoch 2/10

Training: 100% ██████████ 59/59 [00:01<00:00, 30.99it/s, loss=0.619]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 7.94it/s]

Train Loss: 0.3457, Train Acc: 0.8962

Val Loss: 0.5249, Val Acc: 0.7959, Val F1: 0.7923

Epoch 3/10

Training: 100% ██████████ 59/59 [00:01<00:00, 31.92it/s, loss=0.141]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 7.56it/s]

Train Loss: 0.1533, Train Acc: 0.9608

Val Loss: 0.5708, Val Acc: 0.7347, Val F1: 0.7366

Epoch 4/10

Training: 100% ██████████ 59/59 [00:02<00:00, 28.60it/s, loss=0.000772]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 7.87it/s]

Train Loss: 0.0884, Train Acc: 0.9860

Val Loss: 0.6888, Val Acc: 0.7143, Val F1: 0.7259

Epoch 5/10

Training: 100% ██████████ 59/59 [00:02<00:00, 28.88it/s, loss=0.033]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 7.74it/s]

Train Loss: 0.0486, Train Acc: 0.9968

Val Loss: 0.5464, Val Acc: 0.7755, Val F1: 0.7818

Epoch 6/10

Training: 100% ██████████ 59/59 [00:01<00:00, 30.53it/s, loss=0.0172]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 7.71it/s]

Train Loss: 0.0330, Train Acc: 0.9968

Val Loss: 0.6182, Val Acc: 0.8163, Val F1: 0.8219

Epoch 7/10

Training: 100% ██████████ 59/59 [00:01<00:00, 31.32it/s, loss=0.0632]

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 8.12it/s]

Train Loss: 0.0231, Train Acc: 0.9978

Val Loss: 0.6032, Val Acc: 0.8163, Val F1: 0.8219

Early stopping triggered

Evaluating: 100% ██████████ 2/2 [00:00<00:00, 7.87it/s]

Test Loss: 0.6926, Test Acc: 0.7500, Test F1: 0.7434

데이터셋이 다양한 구조를 포함하는게 아니라서 간소화된 버전으로 Custom 구조를 구축하여 학습한 결과 성능이 준수하지는 않습니다.

## 코드 - MobileNet Train

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import timm # PyTorch Image Models 라이브러리
import os
from PIL import Image
import numpy as np
from tqdm import tqdm
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
import random

# 기본 설정
BATCH_SIZE = 16 # 배치 크기
NUM_EPOCHS = 30 # 학습 에폭 수
LEARNING_RATE = 1e-3 # MobileNet에 맞는 학습률
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # GPU 사용 여부 확인
MODEL_NAME = "mobilenetv3_large_100" # MobileNetV3 모델 사용
PREPROCESS_DIR = 'data/preprocess_data' # 데이터 전처리 디렉터리
PATIENCE = 5 # Early Stopping patience
SEED = 42 # 랜덤 시드

# 시드 고정 함수
def seed_everything(seed):
    """재현성을 위해 모든 랜덤 시드를 고정."""
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
```

```

torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# 커스텀 데이터셋 클래스 정의
class CustomImageDataset(Dataset):
    """이미지 데이터셋을 처리하기 위한 클래스."""
    def __init__(self, root_dir, split='train'):
        self.root_dir = os.path.join(root_dir, split) # 데이터 분할 디렉터리 지정
        self.image_paths = [] # 이미지 경로 저장 리스트
        self.labels = [] # 레이블 저장 리스트
        self.class_to_idx = {} # 클래스 이름과 인덱스 매핑

        # 클래스별로 이미지 경로와 레이블 수집
        for idx, class_name in enumerate(sorted(os.listdir(self.root_dir))):
            self.class_to_idx[class_name] = idx
            class_dir = os.path.join(self.root_dir, class_name)
            if os.path.isdir(class_dir):
                for img_name in sorted(os.listdir(class_dir)):
                    if img_name.endswith('.jpg'): # JPG 파일만 처리
                        self.image_paths.append(os.path.join(class_dir, img_name))
                        self.labels.append(idx)

    def __len__(self):
        """데이터셋 크기 반환."""
        return len(self.image_paths)

    def __getitem__(self, idx):
        """특정 인덱스의 이미지와 레이블 반환."""
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB') # 이미지를 RGB로 변환
        image = image.resize((224, 224), Image.BILINEAR) # 이미지 크기 조정
        image = torch.from_numpy(np.array(image)).permute(2, 0, 1).float() / 255.0 # 정규화 및 Tensor
로 변환
        label = self.labels[idx] # 해당 이미지의 레이블
        return image, label

# 한 에폭 동안 모델 학습
def train_one_epoch(model, train_loader, optimizer, criterion, device):
    """한 에폭 동안 학습."""
    model.train() # 학습 모드로 전환
    total_loss = 0
    all_preds = []

```

```

all_labels = []

progress_bar = tqdm(train_loader, desc='Training') # 진행 상황 표시
for images, labels in progress_bar:
    images, labels = images.to(device), labels.to(device) # 데이터 로드 후 디바이스로 이동

    optimizer.zero_grad() # 그래디언트 초기화
    outputs = model(images) # 모델 예측값
    loss = criterion(outputs, labels) # 손실 계산

    loss.backward() # 역전파
    optimizer.step() # 가중치 업데이트

    total_loss += loss.item()

    preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값
    all_preds.extend(preds)
    all_labels.extend(labels.cpu().numpy()) # 실제값 저장

# 진행 상황 업데이트
progress_bar.set_postfix({'loss': loss.item()})

epoch_loss = total_loss / len(train_loader) # 평균 손실
epoch_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
return epoch_loss, epoch_acc

# 혼동 행렬 시각화 함수
def plot_confusion_matrix(cm, class_names, title='Confusion Matrix'):
    """혼동 행렬을 시각화."""
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)
    plt.title(title)
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.savefig(f'{title.lower().replace(" ", "_")}.png')
    plt.close()

# 검증 루프
def evaluate(model, val_loader, criterion, device):
    """검증 데이터셋에 대한 평가."""
    model.eval() # 평가 모드로 전환
    total_loss = 0

```

```

all_preds = []
all_labels = []

with torch.no_grad(): # 그래디언트 비활성화
    for images, labels in tqdm(val_loader, desc='Evaluating'):
        images, labels = images.to(device), labels.to(device)
        outputs = model(images) # 모델 예측값
        loss = criterion(outputs, labels) # 손실 계산

        total_loss += loss.item()

        preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy()) # 실제값 저장

val_loss = total_loss / len(val_loader) # 평균 손실
val_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
val_f1 = f1_score(all_labels, all_preds, average='weighted') # F1 Score 계산

return val_loss, val_acc, val_f1, all_preds, all_labels

# 테스트 데이터셋에 대한 평가
def test(model, test_loader, criterion, device, idx_to_class):
    """테스트 데이터셋에 대한 최종 평가."""
    model.eval() # 평가 모드 전환
    total_loss = 0
    all_preds = []
    all_labels = []
    misclassified_images = []
    misclassified_true = []
    misclassified_pred = []

    with torch.no_grad():
        for images, labels in tqdm(test_loader, desc='Testing'):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images) # 모델 예측값
            loss = criterion(outputs, labels) # 손실 계산

            total_loss += loss.item()
            preds = torch.argmax(outputs, dim=1)

            # 오분류된 이미지 수집
            misclassified_mask = preds != labels
            if misclassified_mask.any():
                misclassified_images.extend(images[misclassified_mask])

```

```

        misclassified_true.extend(labels[misclassified_mask].cpu().numpy())
        misclassified_pred.extend(preds[misclassified_mask].cpu().numpy())

    preds = preds.cpu().numpy()
    all_preds.extend(preds)
    all_labels.extend(labels.cpu().numpy())

test_loss = total_loss / len(test_loader) # 평균 손실
test_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
test_f1 = f1_score(all_labels, all_preds, average='weighted') # F1 Score 계산

# 혼동 행렬 및 결과 출력
class_names = [idx_to_class[i] for i in range(len(idx_to_class))]
report = classification_report(all_labels, all_preds, target_names=class_names)

# 혼동 행렬 시각화
cm = confusion_matrix(all_labels, all_preds)
plot_confusion_matrix(cm, class_names, title='Test Confusion Matrix')

return test_loss, test_acc, test_f1, report

# 메인 함수
def main():
    """MobileNet 모델을 학습 및 평가."""
    seed_everything(SEED) # 시드 고정
    print(f"Using device: {DEVICE}")

    # MobileNet 모델 로드
    model = timm.create_model(MODEL_NAME, pretrained=True,
num_classes=len(os.listdir(os.path.join(PREPROCESS_DIR, 'train'))))
    model = model.to(DEVICE)

    # 데이터셋 및 데이터 로더 생성
    train_dataset = CustomImageDataset(PREPROCESS_DIR, split='train')
    val_dataset = CustomImageDataset(PREPROCESS_DIR, split='val')
    test_dataset = CustomImageDataset(PREPROCESS_DIR, split='test')

    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)
    test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

    # 손실 함수 및 옵티마이저 설정
    criterion = nn.CrossEntropyLoss() # 손실 함수
    optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE) # Adam Optimizer
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,

```

```

patience=3, verbose=True)

# 학습 루프
for epoch in range(NUM_EPOCHS):
    print(f"\nEpoch {epoch+1}/{NUM_EPOCHS}")

    # 학습
    train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion, DEVICE)

    # 검증
    val_loss, val_acc, val_f1, val_preds, val_labels = evaluate(model, val_loader, criterion, DEVICE)

    # 스케줄러 업데이트
    scheduler.step(val_loss)

    print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
    print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}, Val F1: {val_f1:.4f}")

# 테스트
idx_to_class = {v: k for k, v in train_dataset.class_to_idx.items()}
test_loss, test_acc, test_f1, test_report = test(model, test_loader, criterion, DEVICE, idx_to_class)
print(f"Test Results:\n{test_report}")

if __name__ == "__main__":
    main()

```

## MobileNet 결과

Using device: cuda

model.safetensors:100%

22.1M/22.1M[00:00<00:00,203MB/s]

Epoch 1/30

Training: 100% ██████████ 117/117 [00:05<00:00, 20.92it/s, loss=1.19e-7]

Evaluating: 100% ██████████ 4/4 [00:00<00:00, 10.81it/s]

Train Loss: 0.8688, Train Acc: 0.8898

Val Loss: 1.3148, Val Acc: 0.8367, Val F1: 0.8537

Current LR: 0.001000


Epoch 2/30

Training: 100% ██████████ 117/117 [00:04<00:00, 24.23it/s, loss=2.59]

Evaluating: 100% ██████████ 4/4 [00:00<00:00, 13.59it/s]



Train Loss: 0.2309, Train Acc: 0.9715  
Val Loss: 1.0367, Val Acc: 0.9184, Val F1: 0.9176  
Current LR: 0.001000

Epoch 3/30

Training: 100% 117/117 [00:04<00:00, 24.07it/s, loss=0]  
Evaluating: 100% 4/4 [00:00<00:00, 13.28it/s]



Train Loss: 0.2356, Train Acc: 0.9677  
Val Loss: 0.2613, Val Acc: 0.9592, Val F1: 0.9530  
Current LR: 0.001000

Epoch 4/30

Training: 100% 117/117 [00:04<00:00, 24.38it/s, loss=0]  
Evaluating: 100% 4/4 [00:00<00:00, 12.46it/s]

Train Loss: 0.0861, Train Acc: 0.9860  
Val Loss: 0.5266, Val Acc: 0.9388, Val F1: 0.9214  
Current LR: 0.001000  
EarlyStopping counter: 1 out of 5

Epoch 5/30

Training: 100% 117/117 [00:04<00:00, 23.83it/s, loss=3.33]  
Evaluating: 100% 4/4 [00:00<00:00, 13.11it/s]

Train Loss: 0.2148, Train Acc: 0.9769  
Val Loss: 0.5507, Val Acc: 0.9184, Val F1: 0.9210  
Current LR: 0.001000  
EarlyStopping counter: 2 out of 5

Epoch 6/30

Training: 100% 117/117 [00:04<00:00, 24.50it/s, loss=0]  
Evaluating: 100% 4/4 [00:00<00:00, 13.66it/s]

Train Loss: 0.1230, Train Acc: 0.9817  
Val Loss: 0.5940, Val Acc: 0.9592, Val F1: 0.9592  
Current LR: 0.001000  
EarlyStopping counter: 3 out of 5

Epoch 7/30



Training: 100%|██████████████████| 117/117 [00:04<00:00, 23.51it/s, loss=1.19e-7]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 13.51it/s]

Train Loss: 0.1988, Train Acc: 0.9753

Val Loss: 0.6852, Val Acc: 0.9388, Val F1: 0.9382

Current LR: 0.000100

EarlyStopping counter: 4 out of 5

Epoch 8/30

Training: 100%|██████████████████| 117/117 [00:04<00:00, 23.47it/s, loss=2.13]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 13.34it/s]

Train Loss: 0.0817, Train Acc: 0.9941

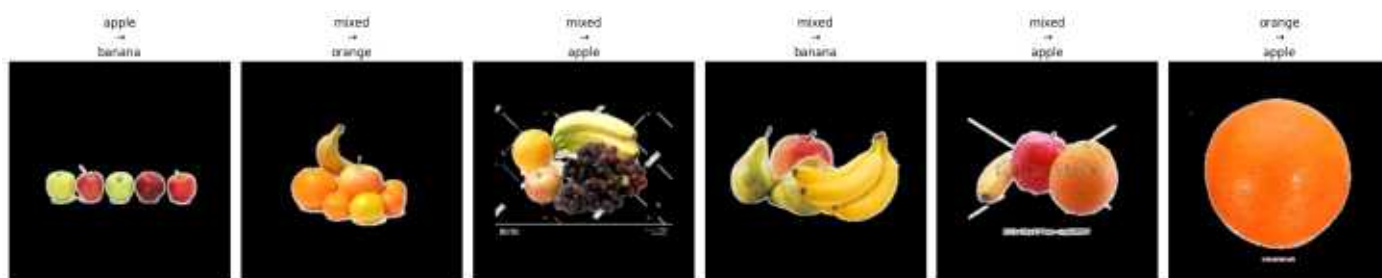
Val Loss: 0.6752, Val Acc: 0.9388, Val F1: 0.9382

Current LR: 0.000100

EarlyStopping counter: 5 out of 5

Early stopping triggered

Testing: 100%|██████████████████| 4/4 [00:00<00:00, 11.08it/s]



Test Results:

Test Loss: 1.0485

Test Accuracy: 0.9000

Test F1 Score: 0.8803

Test Classification Report:

	precision	recall	f1-score	support
apple	0.86	0.95	0.90	19
banana	0.90	1.00	0.95	18
mixed	1.00	0.20	0.33	5
orange	0.94	0.94	0.94	18
accuracy			0.90	60

macro avg	0.93	0.77	0.78	60
weighted avg	0.91	0.90	0.88	60

괜찮은 성능을 보이기는 하나 일부 클래스에서 오류를 보이고 있습니다.

## 코드 - EfficientNet Train

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import timm # PyTorch Image Models 라이브러리
import os
from PIL import Image
import numpy as np
from tqdm import tqdm
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
import random

# 기본 설정
BATCH_SIZE = 16 # 배치 크기
NUM_EPOCHS = 30 # 학습 에폭 수
LEARNING_RATE = 1e-4 # EfficientNet 학습률
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # GPU 사용 여부 확인
MODEL_NAME = "efficientnet_b0" # EfficientNet 모델 이름
PREPROCESS_DIR = 'data/preprocess_data' # 데이터 전처리 경로
PATIENCE = 5 # Early stopping patience
SEED = 42 # 랜덤 시드 값

# 시드 고정 함수
def seed_everything(seed):
    """재현성을 위해 모든 랜덤 시드를 고정."""
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# 커스텀 데이터셋 클래스
class CustomImageDataset(Dataset):
    """이미지 데이터셋 클래스."""
    def __init__(self, root_dir, split='train'):
        self.root_dir = os.path.join(root_dir, split) # 데이터 분할 디렉터리
```

```

self.image_paths = [] # 이미지 경로 리스트
self.labels = [] # 레이블 리스트
self.class_to_idx = {} # 클래스 이름과 인덱스 매핑

# 클래스별로 이미지 경로와 레이블 수집
for idx, class_name in enumerate(sorted(os.listdir(self.root_dir))):
    self.class_to_idx[class_name] = idx
    class_dir = os.path.join(self.root_dir, class_name)
    if os.path.isdir(class_dir):
        for img_name in sorted(os.listdir(class_dir)):
            if img_name.endswith('.jpg'): # JPG 파일만 처리
                self.image_paths.append(os.path.join(class_dir, img_name))
                self.labels.append(idx)

def __len__(self):
    """데이터셋 크기 반환."""
    return len(self.image_paths)

def __getitem__(self, idx):
    """특정 인덱스의 이미지와 레이블 반환."""
    img_path = self.image_paths[idx]
    image = Image.open(img_path).convert('RGB') # 이미지를 RGB로 변환
    image = image.resize((224, 224), Image.BILINEAR) # EfficientNet 입력 크기로 조정
    image = torch.from_numpy(np.array(image)).permute(2, 0, 1).float() / 255.0 # 정규화 및 Tensor
    변환

    # ImageNet 표준 정규화
    mean = torch.tensor([0.485, 0.456, 0.406]).view(-1, 1, 1)
    std = torch.tensor([0.229, 0.224, 0.225]).view(-1, 1, 1)
    image = (image - mean) / std
    label = self.labels[idx] # 이미지의 레이블
    return image, label

# 한 에폭 동안 학습
def train_one_epoch(model, train_loader, optimizer, criterion, device):
    """한 에폭 동안 모델을 학습."""
    model.train() # 학습 모드 전환
    total_loss = 0
    all_preds = []
    all_labels = []

    progress_bar = tqdm(train_loader, desc='Training') # 진행 상황 표시
    for images, labels in progress_bar:
        images, labels = images.to(device), labels.to(device) # 데이터 로드 및 디바이스로 이동

        optimizer.zero_grad() # 그래디언트 초기화

```

```

    outputs = model(images) # 모델 출력
    loss = criterion(outputs, labels) # 손실 계산

    loss.backward() # 역전파
    optimizer.step() # 가중치 업데이트

    total_loss += loss.item()
    preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값 계산
    all_preds.extend(preds)
    all_labels.extend(labels.cpu().numpy()) # 실제값 저장

    # 진행 상황 업데이트
    progress_bar.set_postfix({'loss': loss.item()})

    epoch_loss = total_loss / len(train_loader) # 평균 손실
    epoch_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
    return epoch_loss, epoch_acc

# 검증 함수
def evaluate(model, val_loader, criterion, device):
    """검증 데이터셋에 대한 평가."""
    model.eval() # 평가 모드 전환
    total_loss = 0
    all_preds = []
    all_labels = []

    with torch.no_grad(): # 그래디언트 비활성화
        for images, labels in tqdm(val_loader, desc='Evaluating'):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images) # 모델 출력
            loss = criterion(outputs, labels) # 손실 계산

            total_loss += loss.item()
            preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값
            all_preds.extend(preds)
            all_labels.extend(labels.cpu().numpy()) # 실제값 저장

    val_loss = total_loss / len(val_loader) # 평균 손실
    val_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
    val_f1 = f1_score(all_labels, all_preds, average='weighted') # F1 Score 계산

    return val_loss, val_acc, val_f1, all_preds, all_labels

# 테스트 데이터 평가
def test(model, test_loader, criterion, device, idx_to_class):

```

```

"""테스트 데이터에 대한 최종 평가."""
model.eval() # 평가 모드 전환
total_loss = 0
all_preds = []
all_labels = []

with torch.no_grad(): # 그래디언트 비활성화
    for images, labels in tqdm(test_loader, desc='Testing'):
        images, labels = images.to(device), labels.to(device)
        outputs = model(images) # 모델 출력
        loss = criterion(outputs, labels) # 손실 계산

        total_loss += loss.item()
        preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy()) # 실제값 저장

test_loss = total_loss / len(test_loader) # 평균 손실
test_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
test_f1 = f1_score(all_labels, all_preds, average='weighted') # F1 Score 계산

# 분류 리포트 생성
class_names = [idx_to_class[i] for i in range(len(idx_to_class))]
report = classification_report(all_labels, all_preds, target_names=class_names)

return test_loss, test_acc, test_f1, report

# 메인 함수
def main():
    """EfficientNet 모델 학습 및 평가."""
    seed_everything(SEED) # 시드 고정
    print(f"Using device: {DEVICE}")

    # EfficientNet 모델 로드
    model = timm.create_model(MODEL_NAME, pretrained=True,
num_classes=len(os.listdir(os.path.join(PREPROCESS_DIR, 'train'))))
    model = model.to(DEVICE)

    # 데이터셋 및 데이터 로더 생성
    train_dataset = CustomImageDataset(PREPROCESS_DIR, split='train')
    val_dataset = CustomImageDataset(PREPROCESS_DIR, split='val')
    test_dataset = CustomImageDataset(PREPROCESS_DIR, split='test')

    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

```

```

test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

# 손실 함수 및 옵티마이저 설정
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=LEARNING_RATE)

# 학습 루프
for epoch in range(NUM_EPOCHS):
    print(f"\nEpoch {epoch+1}/{NUM_EPOCHS}")

    # 학습
    train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion, DEVICE)

    # 검증
    val_loss, val_acc, val_f1, _, _ = evaluate(model, val_loader, criterion, DEVICE)

    print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
    print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}, Val F1: {val_f1:.4f}")

if __name__ == "__main__":
    main()

```

## EfficientNet Train 결과

Using device: cuda

Epoch 1/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.68it/s, loss=0.311]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 11.72it/s]

Train Loss: 0.3715, Train Acc: 0.8935

Val Loss: 0.2271, Val Acc: 0.8776, Val F1: 0.8836

Current LR: 0.000100

Epoch 2/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.44it/s, loss=0.135]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.45it/s]

Train Loss: 0.0389, Train Acc: 0.9860

Val Loss: 0.2207, Val Acc: 0.8980, Val F1: 0.9021

Current LR: 0.000100

Epoch 3/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.37it/s, loss=0.0107]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 11.95it/s]

Train Loss: 0.0221, Train Acc: 0.9925

Val Loss: 0.1286, Val Acc: 0.8980, Val F1: 0.9002

Current LR: 0.000100

Epoch 4/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.53it/s, loss=0.0069]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.70it/s]

Train Loss: 0.0221, Train Acc: 0.9903

Val Loss: 0.1484, Val Acc: 0.9184, Val F1: 0.9167

Current LR: 0.000100

EarlyStopping counter: 1 out of 5

Epoch 5/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.20it/s, loss=0.0039]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.54it/s]

Train Loss: 0.0060, Train Acc: 0.9978

Val Loss: 0.0593, Val Acc: 0.9592, Val F1: 0.9600

Current LR: 0.000100

Epoch 6/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.51it/s, loss=1.19e-7]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.21it/s]

Train Loss: 0.0085, Train Acc: 0.9962

Val Loss: 0.0799, Val Acc: 0.9592, Val F1: 0.9592

Current LR: 0.000100

EarlyStopping counter: 1 out of 5

Epoch 7/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.25it/s, loss=0.184]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.24it/s]


Train Loss: 0.0271, Train Acc: 0.9930

Val Loss: 0.1786, Val Acc: 0.9388, Val F1: 0.9386

Current LR: 0.000100

EarlyStopping counter: 2 out of 5

Epoch 8/30

Training: 100% 117/117 [00:06<00:00, 17.30it/s, loss=0.00108]

Evaluating: 100% 4/4 [00:00<00:00, 11.64it/s]


Train Loss: 0.0278, Train Acc: 0.9887

Val Loss: 0.1351, Val Acc: 0.9388, Val F1: 0.9394

Current LR: 0.000100

EarlyStopping counter: 3 out of 5

Epoch 9/30

Training: 100% 117/117 [00:06<00:00, 17.40it/s, loss=0.00768]


Evaluating: 100% 4/4 [00:00<00:00, 11.26it/s]

Train Loss: 0.0116, Train Acc: 0.9973

Val Loss: 0.0379, Val Acc: 0.9796, Val F1: 0.9796

Current LR: 0.000100

Epoch 10/30

Training: 100% 117/117 [00:06<00:00, 17.70it/s, loss=0.000379]


Evaluating: 100% 4/4 [00:00<00:00, 11.57it/s]

Train Loss: 0.0080, Train Acc: 0.9962

Val Loss: 0.0085, Val Acc: 1.0000, Val F1: 1.0000

Current LR: 0.000100

Epoch 11/30

Training: 100% 117/117 [00:06<00:00, 17.52it/s, loss=0.0247]

Evaluating: 100% 4/4 [00:00<00:00, 12.39it/s]


Train Loss: 0.0073, Train Acc: 0.9978

Val Loss: 0.0371, Val Acc: 1.0000, Val F1: 1.0000

Current LR: 0.000100

EarlyStopping counter: 1 out of 5

Epoch 12/30

Training: 100% 117/117 [00:06<00:00, 17.50it/s, loss=2.1e-5]

Evaluating: 100% 4/4 [00:00<00:00, 11.41it/s]

Train Loss: 0.0032, Train Acc: 0.9984



Val Loss: 0.0537, Val Acc: 0.9592, Val F1: 0.9581

Current LR: 0.000100

EarlyStopping counter: 2 out of 5

Epoch 13/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.98it/s, loss=5.22e-6]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.19it/s]

Train Loss: 0.0025, Train Acc: 0.9995

Val Loss: 0.0163, Val Acc: 1.0000, Val F1: 1.0000

Current LR: 0.000100

EarlyStopping counter: 3 out of 5

Epoch 14/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.61it/s, loss=0.00868]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.45it/s]

Train Loss: 0.0047, Train Acc: 0.9978

Val Loss: 0.0948, Val Acc: 0.9592, Val F1: 0.9592

Current LR: 0.000010

EarlyStopping counter: 4 out of 5

Epoch 15/30

Training: 100%|██████████████████| 117/117 [00:06<00:00, 17.58it/s, loss=0.00103]

Evaluating: 100%|██████████████████| 4/4 [00:00<00:00, 12.30it/s]

Train Loss: 0.0011, Train Acc: 1.0000

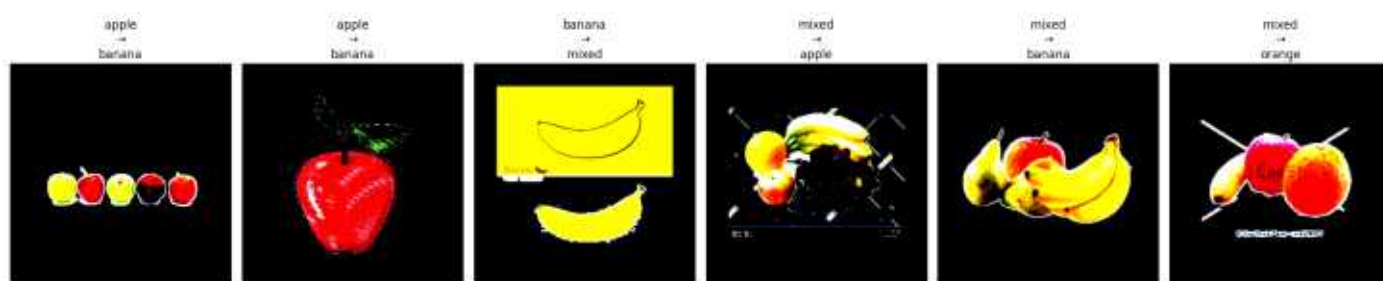
Val Loss: 0.1028, Val Acc: 0.9796, Val F1: 0.9804

Current LR: 0.000010

EarlyStopping counter: 5 out of 5

Early stopping triggered

Testing: 100%|██████████████████| 4/4 [00:00<00:00, 11.20it/s]



Test Results:

Test Loss: 0.4458

Test Accuracy: 0.9000

Test F1 Score: 0.8930

Test Classification Report:

	precision	recall	f1-score	support
apple	0.94	0.89	0.92	19
banana	0.85	0.94	0.89	18
mixed	0.67	0.40	0.50	5
orange	0.95	1.00	0.97	18
accuracy			0.90	60
macro avg	0.85	0.81	0.82	60
weighted avg	0.89	0.90	0.89	60

#혼합 클래스의 경우 혼합 중 하나의 클래스로 분류하는 경향을 보이고 있습니다.

## 코드 - ViT Train

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from transformers import ViTForImageClassification, ViTFeatureExtractor
import os
from PIL import Image
import numpy as np
from tqdm import tqdm
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
import random

# 기본 설정
BATCH_SIZE = 16 # 배치 크기
NUM_EPOCHS = 5 # 학습 에폭 수
LEARNING_RATE = 2e-5 # 학습률
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # 디바이스 설정
MODEL_NAME = "google/vit-base-patch16-224" # 사전학습된 ViT 모델 이름
PREPROCESS_DIR = 'data/preprocess_data' # 데이터 경로
PATIENCE = 5 # Early stopping patience
SEED = 42 # 시드 값 (재현성을 위해 설정)

# 재현성을 위한 시드 고정 함수
```

```

def seed_everything(seed):
    """시드 값을 고정하여 코드 실행의 재현성을 보장."""
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# 커스텀 이미지 데이터셋 정의
class CustomImageDataset(Dataset):
    """이미지 데이터셋 클래스."""
    def __init__(self, root_dir, feature_extractor, split='train'):
        self.root_dir = os.path.join(root_dir, split) # 데이터 분할 디렉터리 (train/val/test)
        self.feature_extractor = feature_extractor # ViT Feature Extractor
        self.image_paths = [] # 이미지 경로 리스트
        self.labels = [] # 레이블 리스트
        self.class_to_idx = {} # 클래스 이름과 인덱스 매핑

        # 데이터 디렉터리에서 클래스와 이미지 경로 수집
        for idx, class_name in enumerate(sorted(os.listdir(self.root_dir))):
            self.class_to_idx[class_name] = idx
            class_dir = os.path.join(self.root_dir, class_name)
            if os.path.isdir(class_dir):
                for img_name in sorted(os.listdir(class_dir)):
                    if img_name.endswith('.jpg'): # 이미지 파일 필터
                        self.image_paths.append(os.path.join(class_dir, img_name))
                        self.labels.append(idx)

    def __len__(self):
        """데이터셋 크기 반환."""
        return len(self.image_paths)

    def __getitem__(self, idx):
        """특정 인덱스의 데이터 반환."""
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB') # RGB 이미지로 로드
        inputs = self.feature_extractor(images=image, return_tensors="pt") # ViT Feature Extractor 적용
        pixel_values = inputs.pixel_values.squeeze() # Tensor로 변환
        label = self.labels[idx]
        return pixel_values, label

# 한 에폭 동안 학습

```

```

def train_one_epoch(model, train_loader, optimizer, criterion, device):
    """한 에폭 동안 모델을 학습."""
    model.train() # 학습 모드 설정
    total_loss = 0
    all_preds = []
    all_labels = []

    progress_bar = tqdm(train_loader, desc='Training') # 진행 상황 표시
    for pixel_values, labels in progress_bar:
        pixel_values, labels = pixel_values.to(device), labels.to(device)

        optimizer.zero_grad() # 이전 그래디언트 초기화
        outputs = model(pixel_values=pixel_values).logits # 모델 출력
        loss = criterion(outputs, labels) # 손실 계산

        loss.backward() # 역전파
        optimizer.step() # 가중치 업데이트

        total_loss += loss.item()

        preds = torch.argmax(outputs, dim=1).cpu().numpy()
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy())

    # 진행 상황 업데이트
    progress_bar.set_postfix({'loss': loss.item()})

    epoch_loss = total_loss / len(train_loader) # 평균 손실
    epoch_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
    return epoch_loss, epoch_acc

# 혼동 행렬 시각화 함수
def plot_confusion_matrix(cm, class_names, title='Confusion Matrix'):
    """혼동 행렬을 시각화."""
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)
    plt.title(title)
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.savefig(f'{title.lower().replace(" ", "_")}.png')
    plt.close()

```

```

# Early Stopping 구현
class EarlyStopping:
    """검증 손실 개선 여부에 따라 학습을 조기에 중단."""
    def __init__(self, patience=7, min_delta=0, path='checkpoint.pth'):
        self.patience = patience
        self.min_delta = min_delta
        self.path = path
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss, model, optimizer, epoch, val_acc):
        """검증 손실을 기준으로 Early Stopping 여부 결정."""
        if self.best_loss is None or val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.save_checkpoint(val_loss, model, optimizer, epoch, val_acc)
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True

    def save_checkpoint(self, val_loss, model, optimizer, epoch, val_acc):
        """현재 모델 상태를 체크포인트로 저장."""
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'val_loss': val_loss,
            'val_acc': val_acc
        }, self.path)

# 평가 루프
def evaluate(model, val_loader, criterion, device):
    """검증 데이터에 대한 평가."""
    model.eval() # 평가 모드 설정
    total_loss = 0
    all_preds = []
    all_labels = []

    with torch.no_grad(): # 그래디언트 비활성화
        for pixel_values, labels in tqdm(val_loader, desc='Evaluating'):
            pixel_values, labels = pixel_values.to(device), labels.to(device)
            outputs = model(pixel_values=pixel_values).logits
            loss = criterion(outputs, labels)

```

```

        total_loss += loss.item()
        preds = torch.argmax(outputs, dim=1).cpu().numpy()
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy())

val_loss = total_loss / len(val_loader) # 평균 손실
val_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
val_f1 = f1_score(all_labels, all_preds, average='weighted') # F1 Score 계산

return val_loss, val_acc, val_f1, all_preds, all_labels

# 메인 함수
def main():
    """모델 학습 및 평가."""
    seed_everything(SEED) # 시드 고정
    print(f"Using device: {DEVICE}")

    # Feature Extractor 및 모델 초기화
    feature_extractor = ViTFeatureExtractor.from_pretrained(MODEL_NAME)
    model = ViTForImageClassification.from_pretrained(
        MODEL_NAME,
        num_labels=len(os.listdir(os.path.join(PREPROCESS_DIR, 'train'))),
        ignore_mismatched_sizes=True
    )
    model = model.to(DEVICE)

    # 데이터셋 및 데이터 로더 생성
    train_dataset = CustomImageDataset(PREPROCESS_DIR, feature_extractor, split='train')
    val_dataset = CustomImageDataset(PREPROCESS_DIR, feature_extractor, split='val')
    test_dataset = CustomImageDataset(PREPROCESS_DIR, feature_extractor, split='test')

    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

    # 손실 함수, 옵티마이저, 스케줄러 정의
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.AdamW(model.parameters(), lr=LEARNING_RATE)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=NUM_EPOCHS)

    # Early Stopping 초기화
    early_stopping = EarlyStopping(patience=PATIENCE, path='best_model.pth')

    # 학습 루프
    for epoch in range(NUM_EPOCHS):

```

```

print(f"\nEpoch {epoch+1}/{NUM_EPOCHS}")

# 학습
train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion, DEVICE)

# 검증
val_loss, val_acc, val_f1, val_preds, val_labels = evaluate(model, val_loader, criterion, DEVICE)

# 스케줄러 업데이트
scheduler.step()

print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}, Val F1: {val_f1:.4f}")

# Early Stopping 체크
early_stopping(val_loss, model, optimizer, epoch, val_acc)

if early_stopping.early_stop:
    print("Early stopping triggered")
    break


# 최적 모델 로드 및 테스트
checkpoint = torch.load('best_model.pth')
model.load_state_dict(checkpoint['model_state_dict'])

if __name__ == "__main__":
    main()

```

## ViT 결과

Epoch 1/5


Training: 100% 117/117 [00:30<00:00, 3.86it/s, loss=0.0153]

Evaluating: 100% 4/4 [00:00<00:00, 8.32it/s]

Train Loss: 0.2032, Train Acc: 0.9570

Val Loss: 0.0625, Val Acc: 1.0000, Val F1: 1.0000

Epoch 2/5

Training: 100% 117/117 [00:29<00:00, 3.92it/s, loss=0.00676]

Evaluating: 100% 4/4 [00:00<00:00, 8.10it/s]

Train Loss: 0.0082, Train Acc: 1.0000

Val Loss: 0.0456, Val Acc: 1.0000, Val F1: 1.0000

Epoch 3/5

Training: 100% ██████████ 117/117 [00:29<00:00, 3.95it/s, loss=0.00406]

Evaluating: 100% ██████████ 4/4 [00:00<00:00, 8.11it/s]

Train Loss: 0.0036, Train Acc: 1.0000

Val Loss: 0.0387, Val Acc: 1.0000, Val F1: 1.0000

Epoch 4/5

Training: 100% ██████████ 117/117 [00:30<00:00, 3.90it/s, loss=0.00251]

Evaluating: 100% ██████████ 4/4 [00:00<00:00, 8.16it/s]

Train Loss: 0.0025, Train Acc: 1.0000

Val Loss: 0.0364, Val Acc: 1.0000, Val F1: 1.0000

Epoch 5/5

Training: 100% ██████████ 117/117 [00:29<00:00, 3.91it/s, loss=0.00182]

Evaluating: 100% ██████████ 4/4 [00:00<00:00, 8.05it/s]

Train Loss: 0.0022, Train Acc: 1.0000

Val Loss: 0.0357, Val Acc: 1.0000, Val F1: 1.0000

checkpoint = torch.load('best\_model.pth')

Testing: 100% ██████████ 4/4 [00:00<00:00, 7.38it/s]

apple  
→  
mixed





Test Results:

Test Loss: 0.0552

Test Accuracy: 0.9833

Test F1 Score: 0.9839

Test Classification Report:

	precision	recall	f1-score	support
apple	1.00	0.95	0.97	19
banana	1.00	1.00	1.00	18
mixed	0.83	1.00	0.91	5
orange	1.00	1.00	1.00	18
accuracy			0.98	60
macro avg	0.96	0.99	0.97	60
weighted avg	0.99	0.98	0.98	60

가장 강력한 성능을 보이고 있습니다. 초록색 사과가 섞여 혼합클래스로 오분류 하는 경향이 있는 것 같습니다.

## 코드 - Res Net 50 Train

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import timm # PyTorch Image Models 라이브러리
import os
from PIL import Image
import numpy as np
from tqdm import tqdm
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
import random

# 기본 설정
BATCH_SIZE = 16 # 배치 크기
NUM_EPOCHS = 3 # 학습 에폭 수
LEARNING_RATE = 1e-4 # ResNet50에 맞는 학습률
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # GPU 사용 여부 확인
MODEL_NAME = "resnet50" # ResNet50 모델 이름
PREPROCESS_DIR = 'data/preprocess_data' # 데이터 전처리 경로
PATIENCE = 5 # Early stopping patience
SEED = 42 # 랜덤 시드 값
```

```

# 시드 고정 함수
def seed_everything(seed):
    """재현성을 위해 모든 랜덤 시드를 고정."""
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# 커스텀 데이터셋 클래스
class CustomImageDataset(Dataset):
    """이미지 데이터셋 클래스."""
    def __init__(self, root_dir, split='train'):
        self.root_dir = os.path.join(root_dir, split) # 데이터 분할 디렉터리
        self.image_paths = [] # 이미지 경로 리스트
        self.labels = [] # 레이블 리스트
        self.class_to_idx = {} # 클래스 이름과 인덱스 매핑

        # 클래스별로 이미지 경로와 레이블 수집
        for idx, class_name in enumerate(sorted(os.listdir(self.root_dir))):
            self.class_to_idx[class_name] = idx
            class_dir = os.path.join(self.root_dir, class_name)
            if os.path.isdir(class_dir):
                for img_name in sorted(os.listdir(class_dir)):
                    if img_name.endswith('.jpg'): # JPG 파일만 처리
                        self.image_paths.append(os.path.join(class_dir, img_name))
                        self.labels.append(idx)

    def __len__(self):
        """데이터셋 크기 반환."""
        return len(self.image_paths)

    def __getitem__(self, idx):
        """특정 인덱스의 이미지와 레이블 반환."""
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB') # 이미지를 RGB로 변환
        image = image.resize((224, 224), Image.BILINEAR) # ResNet50 입력 크기로 조정
        image = torch.from_numpy(np.array(image)).permute(2, 0, 1).float() / 255.0 # 정규화 및 Tensor
        # 변환
        image = self.normalize(image) # ResNet 표준 정규화
        label = self.labels[idx] # 이미지의 레이블
        return image, label

```

```

def normalize(self, image):
    """ResNet 표준 정규화."""
    mean = torch.tensor([0.485, 0.456, 0.406]).view(-1, 1, 1)
    std = torch.tensor([0.229, 0.224, 0.225]).view(-1, 1, 1)
    return (image - mean) / std

# 학습 함수
def train_one_epoch(model, train_loader, optimizer, criterion, device):
    """한 에폭 동안 모델을 학습."""
    model.train() # 학습 모드로 전환
    total_loss = 0
    all_preds = []
    all_labels = []

    progress_bar = tqdm(train_loader, desc='Training') # 진행 상황 표시
    for images, labels in progress_bar:
        images, labels = images.to(device), labels.to(device) # 데이터 로드 및 디바이스로 이동

        optimizer.zero_grad() # 그래디언트 초기화
        outputs = model(images) # 모델 출력
        loss = criterion(outputs, labels) # 손실 계산

        loss.backward() # 역전파
        optimizer.step() # 가중치 업데이트

        total_loss += loss.item()
        preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값 계산
        all_preds.extend(preds)
        all_labels.extend(labels.cpu().numpy()) # 실제값 저장

    # 진행 상황 업데이트
    progress_bar.set_postfix({'loss': loss.item()})

    epoch_loss = total_loss / len(train_loader) # 평균 손실
    epoch_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
    return epoch_loss, epoch_acc

# 혼동 행렬 시각화
def plot_confusion_matrix(cm, class_names, title='Confusion Matrix'):
    """혼동 행렬을 시각화."""
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)

```

```

plt.title(title)
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
plt.savefig(f'{title.lower().replace(" ", "_")}.png')
plt.close()

# 검증 함수
def evaluate(model, val_loader, criterion, device):
    """검증 데이터셋에 대한 평가."""
    model.eval() # 평가 모드 전환
    total_loss = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in tqdm(val_loader, desc='Evaluating'):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images) # 모델 예측값
            loss = criterion(outputs, labels) # 손실 계산

            total_loss += loss.item()
            preds = torch.argmax(outputs, dim=1).cpu().numpy() # 예측값
            all_preds.extend(preds)
            all_labels.extend(labels.cpu().numpy()) # 실제값 저장

    val_loss = total_loss / len(val_loader) # 평균 손실
    val_acc = accuracy_score(all_labels, all_preds) # 정확도 계산
    val_f1 = f1_score(all_labels, all_preds, average='weighted') # F1 Score 계산

    return val_loss, val_acc, val_f1, all_preds, all_labels

# 메인 함수
def main():
    """ResNet50 모델 학습 및 평가."""
    seed_everything(SEED) # 시드 고정
    print(f"Using device: {DEVICE}")

    # ResNet50 모델 로드
    model = timm.create_model(MODEL_NAME, pretrained=True,
num_classes=len(os.listdir(os.path.join(PREPROCESS_DIR, 'train'))))
    model = model.to(DEVICE)

    # 데이터셋 및 데이터로더 생성
    train_dataset = CustomImageDataset(PREPROCESS_DIR, split='train')

```

```

val_dataset = CustomImageDataset(PREPROCESS_DIR, split='val')
test_dataset = CustomImageDataset(PREPROCESS_DIR, split='test')

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

# 손실 함수 및 옵티마이저 설정
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=LEARNING_RATE)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,
patience=3, verbose=True)

# 학습 루프
for epoch in range(NUM_EPOCHS):
    print(f"\nEpoch {epoch+1}/{NUM_EPOCHS}")

    # 학습
    train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion, DEVICE)

    # 검증
    val_loss, val_acc, val_f1, val_preds, val_labels = evaluate(model, val_loader, criterion, DEVICE)

    # 스케줄러 업데이트
    scheduler.step(val_loss)

    print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
    print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}, Val F1: {val_f1:.4f}")


if __name__ == "__main__":
    main()

```

## Res Net 50 결과

Using device: cuda

Epoch 1/30

Training: 100% 117/117 [00:09<00:00, 12.67it/s, loss=0.717]


Evaluating: 100% 4/4 [00:00<00:00, 12.86it/s]

Train Loss: 1.0984, Train Acc: 0.6231

Val Loss: 0.7956, Val Acc: 0.8776, Val F1: 0.8799

Current LR: 0.000100

Epoch 2/30

Training: 100% 117/117 [00:09<00:00, 12.72it/s, loss=0.442]


Evaluating: 100% 4/4 [00:00<00:00, 12.21it/s]

Train Loss: 0.2949, Train Acc: 0.9656

Val Loss: 0.1892, Val Acc: 0.9184, Val F1: 0.9190

Current LR: 0.000100

Epoch 3/30

Training: 100% 117/117 [00:09<00:00, 12.63it/s, loss=0.145]


Evaluating: 100% 4/4 [00:00<00:00, 11.94it/s]

Train Loss: 0.0733, Train Acc: 0.9909

Val Loss: 0.1791, Val Acc: 0.8980, Val F1: 0.8986

Current LR: 0.000100

Epoch 4/30

Training: 100% 117/117 [00:09<00:00, 12.60it/s, loss=0.00499]


Evaluating: 100% 4/4 [00:00<00:00, 12.15it/s]

Train Loss: 0.0421, Train Acc: 0.9925

Val Loss: 0.1627, Val Acc: 0.8980, Val F1: 0.8986

Current LR: 0.000100

Epoch 5/30

Training: 100% 117/117 [00:09<00:00, 12.74it/s, loss=0.0472]


Evaluating: 100% 4/4 [00:00<00:00, 12.40it/s]

Train Loss: 0.0235, Train Acc: 0.9968

Val Loss: 0.1593, Val Acc: 0.9388, Val F1: 0.9382

Current LR: 0.000100

Epoch 6/30

Training: 100% 117/117 [00:09<00:00, 12.76it/s, loss=0.00129]



Evaluating: 100% 4/4 [00:00<00:00, 12.51it/s]

Train Loss: 0.0153, Train Acc: 0.9989

Val Loss: 0.1568, Val Acc: 0.9388, Val F1: 0.9382



Current LR: 0.000100

Epoch 7/30

Training: 100% 117/117 [00:09<00:00, 12.71it/s, loss=0.0348]  
Evaluating: 100% 4/4 [00:00<00:00, 11.67it/s]



Train Loss: 0.0117, Train Acc: 0.9989  
Val Loss: 0.1719, Val Acc: 0.9388, Val F1: 0.9382  
Current LR: 0.000100  
EarlyStopping counter: 1 out of 5

Epoch 8/30

Training: 100% 117/117 [00:09<00:00, 12.77it/s, loss=0.0401]  
Evaluating: 100% 4/4 [00:00<00:00, 12.28it/s]



Train Loss: 0.0142, Train Acc: 0.9968  
Val Loss: 0.2489, Val Acc: 0.9184, Val F1: 0.9182  
Current LR: 0.000100  
EarlyStopping counter: 2 out of 5

Epoch 9/30

Training: 100% 117/117 [00:09<00:00, 12.73it/s, loss=0.00749]  
Evaluating: 100% 4/4 [00:00<00:00, 12.39it/s]



Train Loss: 0.0069, Train Acc: 0.9989  
Val Loss: 0.2135, Val Acc: 0.9388, Val F1: 0.9382  
Current LR: 0.000100  
EarlyStopping counter: 3 out of 5

Epoch 10/30

Training: 100% 117/117 [00:09<00:00, 12.75it/s, loss=0.00623]  
Evaluating: 100% 4/4 [00:00<00:00, 10.64it/s]

Train Loss: 0.0136, Train Acc: 0.9973  
Val Loss: 0.2110, Val Acc: 0.9388, Val F1: 0.9382  
Current LR: 0.000010  
EarlyStopping counter: 4 out of 5

Epoch 11/30

Training: 100% 117/117 [00:09<00:00, 12.75it/s, loss=0.18]  
Evaluating: 100% 4/4 [00:00<00:00, 10.95it/s]

Train Loss: 0.0083, Train Acc: 0.9995

Val Loss: 0.2269, Val Acc: 0.9388, Val F1: 0.9382

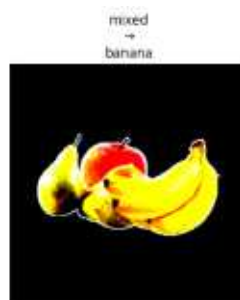
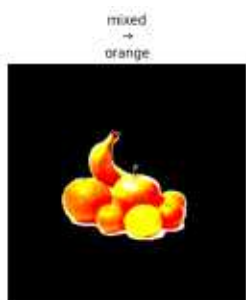
Current LR: 0.000010

EarlyStopping counter: 5 out of 5

Early stopping triggered

checkpoint = torch.load('best\_model.pth')

Testing: 100%  4/4 [00:00<00:00, 11.69it/s]



Test Results:

Test Loss: 0.2301

Test Accuracy: 0.9333

Test F1 Score: 0.9124

Test Classification Report:

	precision	recall	f1-score	support
apple	0.95	1.00	0.97	19
banana	0.90	1.00	0.95	18
mixed	1.00	0.20	0.33	5
orange	0.95	1.00	0.97	18
accuracy			0.93	60
macro avg	0.95	0.80	0.81	60
weighted avg	0.94	0.93	0.91	60

앞선 모델들과 다르게 mixed클래스만 4개를 오분류 하는 모습을 보이고 있습니다.



## 5.1 결과 요약

모델	Train Loss	Train Acc	Val Loss	Val Acc	Val F1	Test Loss	Test Acc	Test F1
MoE & Multimodal	0.0983	95.13%	0.0932	95.92%	90.10%	0.2323	90.00%	84.64%
Simple CNN	0.0330	99.68%	0.6182	81.63%	82.19%	0.6926	75.00%	74.34%
MobileNet	0.1230	98.17%	0.5940	95.92%	95.92%	1.0485	90.00%	88.03%
EfficientNet	0.0278	98.87%	0.1351	93.88%	93.94%	0.4458	90.00%	89.30%
ViT (Vision Transformer)	0.0032	99.95%	0.0357	100.0%	100.0%	0.0552	98.33%	98.39%
ResNet50	0.0069	99.89%	0.2135	93.88%	93.82%	0.2301	93.33%	91.24%

### 결과 해석:

- ViT: 가장 높은 성능(Test F1: 약 98.39%)을 달성하며, 모든 클래스에서 우수한 정확도 및 재현율을 보입니다.
- ResNet50: 전통적 아키텍처 중 가장 성능이 뛰어나며(Test F1: 약 91.24%), 안정적인 분류 성능을 보이나 Mixed 클래스에서 혼동이 발생(F1 33%)하고 있습니다.
- EfficientNet, MobileNet: 전반적으로 높은 정확도(90%)를 유지하며, 경량화 모델인 MobileNet도 준수한 성능(Test F1: 약 88.03%)을 확보. EfficientNet은 Test F1 약 89.30%로 고른 성능을 보입니다.
- MoE & Multimodal: 멀티모달 활용에도 Mixed 클래스에서 성능 저하(F1 60%)가 눈에 띈. 그러나 다른 클래스에는 준수한 성능을 확보하고 있습니다.
- Simple CNN: 가장 단순한 구조로 Test F1 74.34%에 머무르며, 성능 측면에서 한계가 존재하나 기본 베이스라인으로 활용이 가능합니다.

## 5.2 방법론 비교 및 한계

### 모델별 특징:

#### 1. MoE & Multimodal

- 색상, 형상 등 다양한 특성을 융합한 멀티모달 접근으로 검증 세트에서는 높은 성능(Val F1: 90%) 확보했습니다.
- Mixed 클래스에서 낮은 F1(60%)로 성능 저하. 추가 데이터 증강, 클래스별 전문가 추가 튜닝 필요할 것으로 예상됩니다.

#### Simple CNN

- 단순 구조로 빠른 구현 및 학습이 가능합니다.
- Test 성능(F1 약 74%)이 낮아 복잡한 클래스 분류에는 부적합합니다.
- 베이스라인 혹은 사전 실험용으로 활용이 가능합니다.

#### MobileNet

- 경량 모델로 빠른 추론 속도와 낮은 연산량 확보할 수 있습니다.
- Test F1 약 88%로 준수한 성능을 보입니다.
- Mixed 클래스 Recall 저하로 일부 한계를 보입니다.

#### EfficientNet

- 균형 잡힌 확장으로 효율적 특징 추출을 할 수 있습니다.
- Test F1 약 89%로 높은 성능 유지했습니다.
- 특정 클래스(특히 Mixed)에서 약간의 성능이 저하됩니다.

#### ViT (Vision Transformer)

- 전반적으로 가장 뛰어난 성능(Test F1 약 98.39%)을 가지고 있습니다.
- 모든 클래스에서 안정적이고 높은 재현율/정확도를 보입니다.
- 다만 높은 연산량 및 학습 시간, 자원 소모가 큼니다.

#### ResNet50

- 안정적인 전이학습 모델로 여전히 경쟁력 있는 성능(Test F1 약 91%)입니다.
- Mixed 클래스에서 극단적 오분류가 발생(F1: 33%)합니다.

- 클래스별 특성 강화, 데이터 균형화가 필요합니다.

#### 한계사항:

1. Mixed 클래스 성능 저하
  - 대부분 모델이 Mixed 클래스(여러 과일 특성이 결합된 경우)에서 성능이 떨어지고 있습니다.
  - 데이터 증강, 클래스 정의 개선, 추가적인 모달 정보 활용 필요할 것 같습니다.
2. 데이터셋 구조 문제
  - Mixed 클래스 데이터 부족, 클래스 간 불균형 등의 문제가 있습니다.
  - 추가 데이터 수집 또는 클래스 전략 재구성이 필요합니다.
3. 모델 자원 및 시간
  - ViT나 EfficientNet처럼 강력한 모델은 훈련 시간 및 자원 소요가 발생합니다.
  - 실용적 배치를 위해서는 경량 모델(MobileNet)과 성능·효율 균형화 필요합니다.
4. 오분류 원인 분석 필요
  - 비슷한 특성의 과일 간 혼동, Mixed 클래스 모호성 해결을 위한 Attention 기법, Feature Visualization, Data Augmentation 등 다양한 전략이 필요합니다.

#### 결론:

- ViT는 가장 높은 성능을 보였으나, 실용성(학습 자원, 시간) 측면에서 제한이 있을 수 있습니다.
- MobileNet, EfficientNet은 자원 효율성 대비 높은 성능으로 실용적 선택이 가능합니다.
- Mixed 클래스 성능 개선을 위해 추가적인 데이터 수집, 멀티모달 정보 강화, 클래스 재정의 등의 전략이 필요합니다.
- MoE & Multimodal 접근은 가능성을 보였지만 Mixed 클래스 성능 개선 및 전문가 모듈 최적화가 필요합니다.

종합하자면, ViT가 최종 성능에서는 압도적 우위를 보였고, EfficientNet, MobileNet 등이 안정적이며 빠른 추론이 가능한 대안으로 떠오르고 있습니다. 향후 Mixed 클래스 개선 및 효율성 확보 전략 마련이 핵심 과제로 남습니다.

## 참고 사항

A Survey on Mixture of Experts

Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, Jiayi Huang

Chat gpt o1, 4o

Claude 3.5 sonet