CNN 아키텍처 발전의 대표 모델 비교 고찰

- VGG16, ResNet-50, EfficientNet-B3 을 중심으로-

Milo.lee(이정택)/인공지능

목차

1. 서론

딥러닝 기술의 발전과 함께 이미지 분류 작업에서 합성곱 신경망(CNN)은 뛰어난 성능을 발휘하며 다양한 분야에서 널리 활용되고 있다. 특히, 사전 학습된(pretrained) CNN 모델들은 대규모 데이터셋에서 학습된 가중치를 활용하여 적은 데이터로도 높은 정확도를 제공하며, 이미지 분류 작업의 표준으로 자리 잡았다. 본 연구는 CNN 아키텍처의 발전 과정을 대표하는 세 가지 사전 학습 모델인 VGG16, ResNet-50, 그리고 EfficientNet-B3 을 비교 분석하는 데 중점을 둔다.

VGG16 은 단순하고 체계적인 네트워크 구조를 가진 전통적인 CNN 아키텍처로, 3x3 필터와 Max-Pooling 을 반복적으로 쌓아 깊은 네트워크를 구성한 모델이다. 이 모델은 단순한 구조 덕분에 디버깅과 수정이 용이하며, 여전히 다양한 연구와 실무에서 활용되고 있다. ResNet-50 은 잔차 연결(skip connection)을 도입하여 깊은 네트워크에서도 기울기 소실 문제를 해결한 혁신적인 모델로, 딥러닝 이미지 분류 작업의 표준으로 자리 잡았다. 반면, EfficientNet-B3 은 Compound Scaling 기법을 통해 네트워크의 깊이, 너비, 해상도를 균형 있게 조정하여 적은 연산량으로도 높은 성능을 제공하는 최신 아키텍처이다. 이 모델은 자원 제한 환경에서도 효율적으로 작동하며, 모바일 및 IoT 환경에 적합하다.

본 연구에서는 Kaggle 의 다양한 이미지 분류 데이터셋을 활용하여 이 세 가지 모델을 동일한 조건에서 학습 및 평가하고, 각 모델의 성능 차이를 분석한다. 이를 통해 각 아키텍처가 가진 강점과 약점을 규명하고, 특정 작업에 적합한 CNN 아키텍처를 선택하는 기준을 제시하고자 한다. 본 연구는 CNN 아키텍처 간 비교를 통해 딥러닝 모델 선택에 대한 실질적인 통찰을 제공할 수 있을 것으로 기대된다.

2. 데이터셋 설명

본 연구는 CNN 아키텍처(VGG16, ResNet-50, EfficientNet-B3)의 구조적 차이에 따른 분류 성능의 민감도 차이를 비교하기 위해, 시각적 복잡성과 작물군이 서로 다른 세 가지 공개 데이터셋을 활용하였다. 선택된 데이터셋은 토마토 잎 질병 분류, 벼 잎 질병 분류, 피스타치오 품질 분류로 구성되어 있으며, 선정된 데이터셋은 모두 Kaggle 플랫폼에서 제공되는 고품질 이미지 데이터셋이다. 본 연구는 작물의 종류와 질병 또는 품질 특성에 따라 CNN 모델이 얼마나 민감하게 반응하고, 어느 정도의 정확도를 보이는지를 분석하는 데 목적이 있다.

Dataset	Reason for selection
Tomato Disease Multiple Sources	- 많은 데이터 양 및 클래스 수 다양 (10 종 이상) - 병반 패턴이 복잡하고 시각적 다양성 높음 - 다양한 데이터 증강 기법 적용
Pistachio Image Dataset	- 이진 분류 (Good vs Bad) - 단순하지만 미세한 차이의 구별 필요 - 고해상도
Rice Plant Diseases	- 병 종류가 명확히 구분됨 - 잎의 색 변화, 반점 등 시각적 질병 특성이 잘 드러남 - 중간 정도의 데이터 난이도

왼쪽부터 Tomato, Rise, Pistachio 데이터셋 클래스별 데이터 수

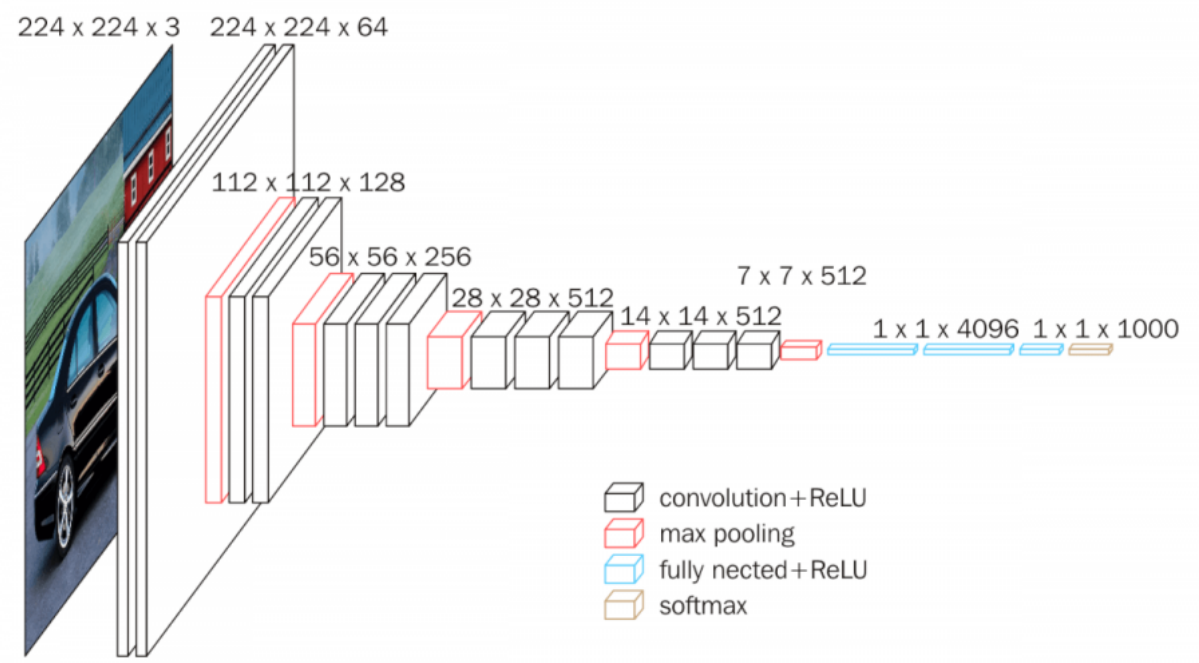
3. 모델 설명

모델 선정 과정에 앞서, CNN 아키텍처의 발전 과정을 살펴보면, CNN 은 LeNet-5 부터 시작해 AlexNet 으로 대중화되었고, VGGNet, GoogLeNet, ResNet, DenseNet, MobileNet 을 거치며 깊이와 효율성을 개선해왔다. 최근에는 EfficientNet 으로 효율성을 극대화했고, CNN 을 넘어 ViT(비전 트랜스포머)의 설계를 통합한 ConvNeXt 모델 같이 발전하는 추세이다. 본 연구에서는 딥러닝이 대중화 되면서부터 생성된 모델들 중 단순하고 직관적인 초기 모델 VGG16, 딥러닝 학습 안정성을 확보한 중기 모델 ResNet50, 성능과 효율을 모두 갖춘 최신 최적화 모델 EfficientNet-B3 를 선정하였다.

다양한 데이터셋을 활용하여 VGG16, ResNet-50, EfficientNet-B3 의 성능을 비교 분석한다. 각 모델은 CNN 아키텍처의 발전 과정을 대표하며, 서로 다른 설계 방식과 학습 효율성을 갖는다. 동일한 데이터셋과 학습 조건에서 이들 모델의 성능을 비교함으로써, 다양한 이미지 분류 작업에서 각 모델의 성능에 대한 비교 고찰을 해보고자 한다.

3-1. VGG16

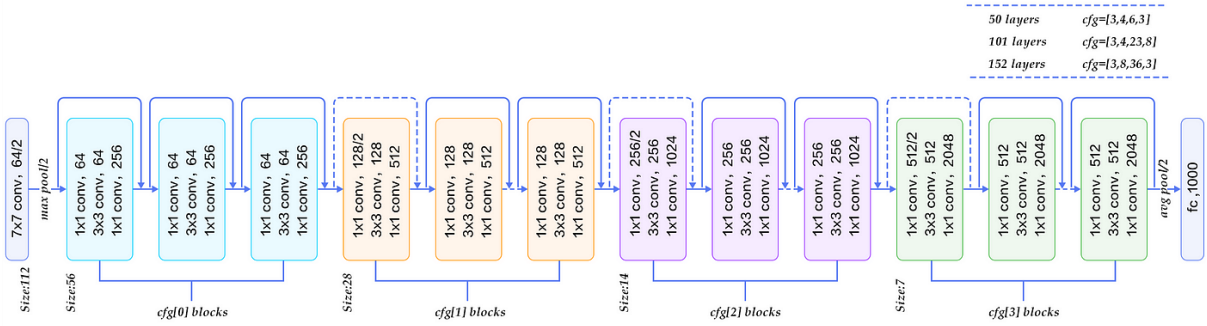
VGG16 은 가장 전통적인 CNN 모델 중 하나로, 간단하고 체계적인 구조를 갖추고 있다. 3x3 컨볼루션 필터와 2x2 맥스 풀링 계층을 반복적으로 쌓아 깊은 네트워크를 형성하며, 총 16 개의 계층으로 구성된다. 구조가 단순하기 때문에 모델의 해석이 용이하고, 다양한 연구 및 실무에서 여전히 사용되고 있다. 하지만 네트워크가 깊어질수록 연산량이 급격히 증가하며, 최신 모델들에 비해 효율성이 낮다는 단점이 있다.



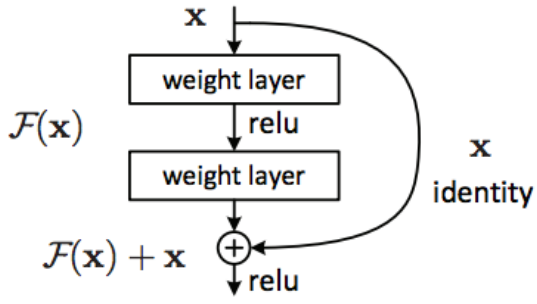
VGG16 Architecture

3-2. ResNet-50

ResNet-50 은 깊은 네트워크에서 발생하는 기울기 소실 문제를 해결하기 위해 잔차 연결(skip connection)을 도입한 모델이다. 50 개의 계층으로 구성된 이 모델은 입력과 출력을 직접 연결하는 구조를 통해 정보 손실을 방지하며, 깊은 네트워크에서도 효과적으로 학습할 수 있도록 설계되었다. 이러한 특징 덕분에 VGG16 보다 높은 정확도를 제공하며, 다양한 이미지 분류 및 객체 탐지 모델의 백본으로 활용되고 있다. 그러나 모델 크기가 상대적으로 크며, 학습과 추론 속도가 EfficientNet 에 비해 다소 느릴 수 있다.



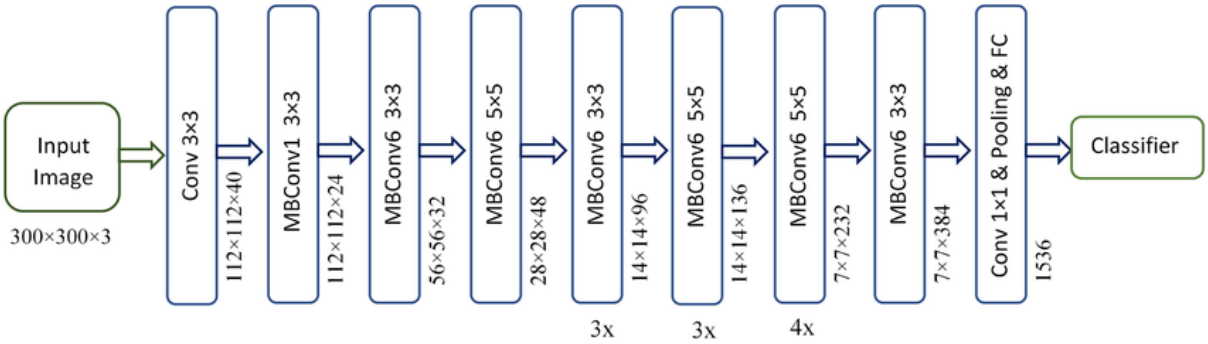
ResNet Architecture



Skip Connetion

3-3. EfficientNet-B3

EfficientNet-B3 은 CNN 모델의 크기와 성능 간의 균형을 최적화하기 위해 개발된 모델이다. 기존 CNN 모델들이 주로 네트워크의 깊이(depth)나 너비(width)를 키워가며 성을 높인 것과 달리, EfficientNet 은 모델의 depth(깊이), width(너비), resolution(입력 크기)를 동시에 조정하는 Compound Scaling 기법을 도입하였다. 이를 통해 적은 연산량으로도 높은 성능을 제공하며, 특히 연산 자원이 제한된 환경에서도 효과적으로 사용할 수 있다. EfficientNet-B3 은 ResNet-50 과 비슷한 수준의 정확도를 유지하면서도 모델 크기와 연산량을 크게 줄였다는 점에서 장점을 가진다.(부록 1)



EfficientNet-B3 Architecture

Characteristic	VGG16	ResNet50	EfficientNetB3
Depth	16 layers	50 layers	26 layers
Key Features	Simple structure with 3x3 filters	Residual blocks, Skip connections	Compound scaling, Depthwise separable convolutions
Number of Parameters	~138 million	~25 million	~12 million
Input Image Size	224x224	224x224	300x300
Main Layers	13 convolutional layers, 3 fully connected layers	Convolutional layers, Residual blocks, Batch normalization	MBConv blocks, Batch normalization
Activation Function	ReLU	ReLU	ReLU
Pooling	Max pooling	Max pooling, Average pooling	Average pooling
Main Advantages	Simple and easy to understand structure	Enables training of deep networks, Solves vanishing gradient problem	Efficient performance, High accuracy with fewer parameters
Key Applicatins	Image classification, Feature extraction	Image classification, Object detection	Image classification, Object detection, Segmentation

4. 연구 방법

본 연구에서는 VGG16, ResNet-50, EfficientNet-B3 세 가지 CNN 아키텍처를 대상으로, 작물 질병 및 품질 이미지에 대한 분류 성능 민감도 차이를 분석하였다. 실험은

PyTorch 프레임워크를 기반으로 수행되었으며, 모델 학습 및 검증은 동일한 조건에서 반복하여 비교 가능하도록 설계하였다.

4.1 데이터셋 구성 및 전처리

데이터셋은 Kaggle 에서 제공하는 Tomato Disease Multiple Sources, Pistachio Image Dataset, Rice Image Dataset 을 사용하였으며, 본 실험에서는 train 데이터셋을 기준으로 코드 실험을 진행하였다. 데이터 누수를 방지하기 위해, 전체 이미지 데이터셋을 기준으로 먼저 Stratified Sampling 을 적용하여 학습, 검증, 테스트 데이터를 분리하였다. 전체 데이터에서 70%는 학습용(train), 15%는 검증용(val), 15%는 테스트용(test)으로 분할하였다. 분할은 sklearn 의 train_test_split 함수를 사용하였고, 클래스 불균형 문제를 방지하기 위해 stratify 옵션을 적용하였다.

전처리 과정은 학습용 데이터에는 데이터 증강을 위해 Resize, RandomHorizontalFlip, ToTensor, Normalize 를 적용하였고, 검증 및 테스트용 데이터에는 증강 없이 Resize, ToTensor, Normalize 만 적용하였다. 학습용과 검증/테스트용 데이터셋은 transform 을 다르게 적용하기 위해, ImageFolder 를 서로 다른 transform 으로 세 번 생성한 뒤, 미리 분리한 인덱스를 기준으로 Subset 을 각각 생성하여 사용하였다.

4.2 모델 설계

본 연구에서는 세 가지 주요 CNN 아키텍처(EfficientNetB3, ResNet50, VGG16)를 기반으로 이미지 분류 모델을 설계하였다. 사전 학습 모델의 기본 구조를 유지하면서 분류기 부분을 커스터마이징하였다. 모델 설계 방식은 공통적으로 다음과 같다:

- 사전 학습된 기본 모델을 기반으로 사용
- 기존 분류기 또는 기존 완전 연결 레이어를 제거하고 새로운 레이어 추가
- 새로운 분류기 구조:

1. 배치 정규화 (BatchNorm1d)
2. 선형 레이어 (in_features → 256)
3. ReLU 활성화 함수
4. 드롭아웃 (rate: 0.45)
5. 최종 선형 레이어 (256 → class_count)

VGG16, ResNet50 의 경우 마지막 분류 레이어의 경우 1000 개의 출력 노드를 가지고 있고, EfficientNetB3 의 경우 1536 개의 출력 노드를 가지고 있다. 따라서 분류기 부분의 커스터마이징을 진행하면서, 바로 데이터셋에 맞추어 노드 수를 크게 줄이면 특징 표현력 손실 및 차원 축소로 인한 정보 손실이 일어날 수 있고, 학습이 불안정하게 된다. 따라서 완화를 위한 레이어를 추가하여 기본적인 성능이 나올 수 있도록 하였다.

4.3 학습

본 실험에서는 VGG16, ResNet-50, EfficientNet-B3 모델을 사용하여 학습을 진행하였다. 각 모델의 학습 과정은 다음과 같다.

학습 설정

- 배치 크기: 32
- 에폭: 50
- 옵티마이저: Adam
- 학습률: 0.001
- 손실 함수: Categorical Cross-Entropy

학습 과정

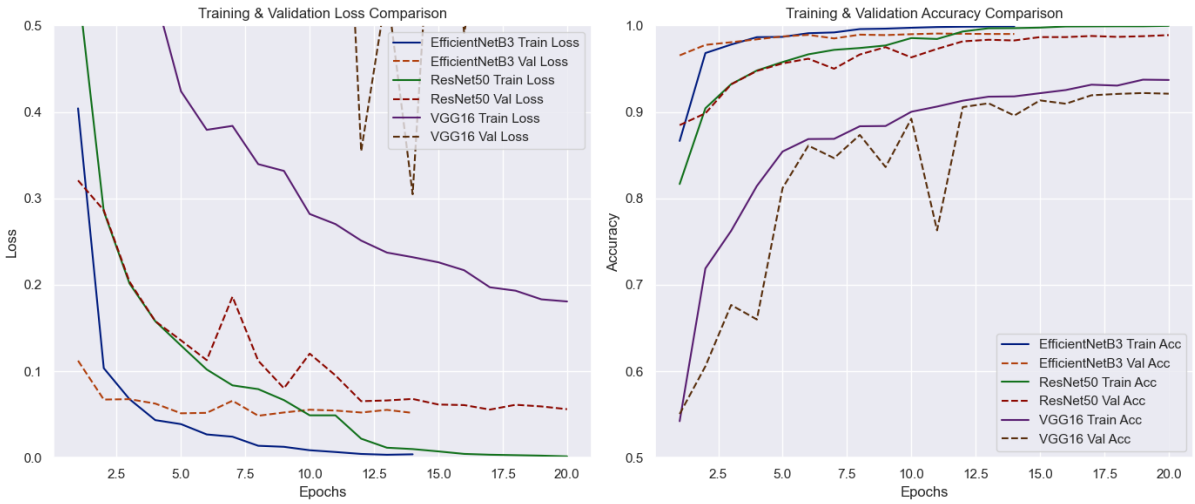
1. 각 모델의 사전 훈련된 가중치를 로드
2. 모델의 최상위 층을 데이터셋에 맞게 수정
3. 전체 모델을 미세 조정 (fine-tuning)
4. 검증 세트를 사용하여 과적합 모니터링
5. 조기 종료 (Early Stopping) 기법 적용하여 최적의 모델 저장

학습 모니터링

- 훈련 및 검증 정확도
- 훈련 및 검증 손실
- 학습 곡선을 통한 모델 성능 추이 관찰

4.4 결과 분석

4.4.1 Tomato Disease Multiple Sources



Loss 그래프 해석 (좌측):

- 모든 모델에서 train loss 는 빠르게 감소함
- EfficientNetB3 와 ResNet50 은 val loss 도 안정적으로 낮아짐 → 좋은 일반화
- VGG16 은 val loss 가 초기에는 낮지만 이후 불안정하거나 수렴이 느림

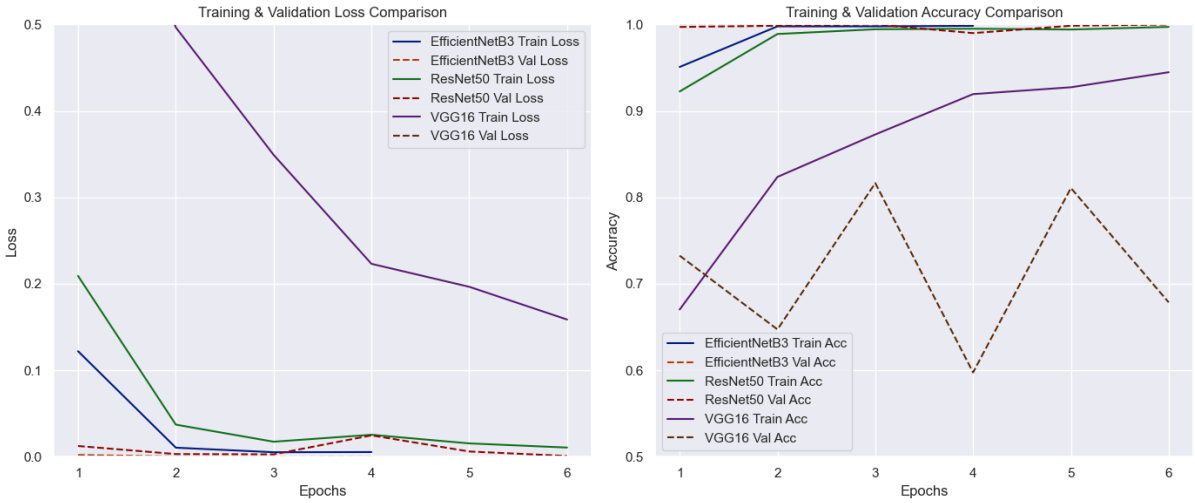
Accuracy 그래프 해석 (우측):

- EfficientNetB3 는 10epoch 이전에 빠르게 90% 이상에 도달 → 빠른 수렴

- ResNet50 도 비슷하게 안정적 상승
- VGG16 은 초기에 낮았다가 점진적으로 증가하지만 가장 낮은 성능

결론: Tomato 처럼 복잡하고 클래스 많은 고해상도 데이터셋에서는 EfficientNetB3 가 가장 우수. ResNet50 도 좋은 대안이지만 VGG16 은 확실히 부족함.

4.4.2 Rice Plant Diseases



Loss 그래프 해석 (좌측):

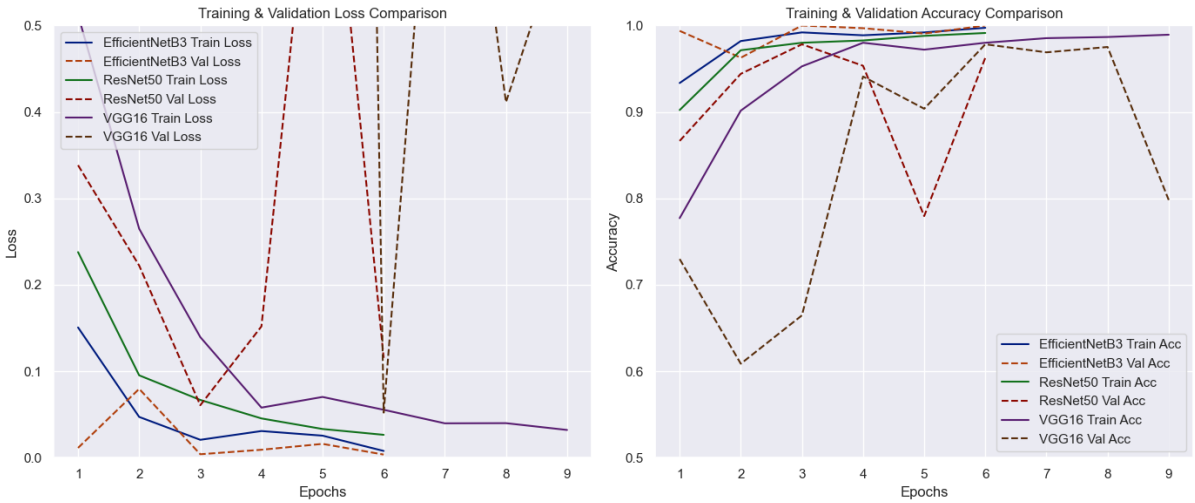
- 전체적으로 모델 간 큰 차이 없음
- VGG16 은 val loss 가 확실히 가장 높은 편 → 불안정
- EfficientNetB3 와 ResNet50 은 val loss 가 빠르게 낮아지며 안정적으로 수렴

Accuracy 그래프 해석 (우측):

- EfficientNetB3 는 val accuracy 100%에 수렴
- ResNet50 도 거의 유사한 수준
- VGG16 은 epoch 별로 큰 진폭을 보이며 불안정 (성능 흔들림)

결론: 모든 모델이 일정 수준 이상 잘 작동하지만, 효율성과 안정성 면에서는 EfficientNetB3 > ResNet50 > VGG16 단순하고 잘 정리된 데이터셋에서 EfficientNet 의 빠른 수렴력 돋보임.

4.4.3 Pistachio Image Dataset



Loss 그래프 해석 (좌측):

- VGG16 과 ResNet50 은 val loss 가 요동치며 불안정
- EfficientNetB3 는 초기부터 꾸준히 낮은 loss 유지
- VGG16 은 학습은 잘 됐지만 val 에서 심각한 과적합 경향

Accuracy 그래프 해석 (우측):

- EfficientNetB3 는 약간의 진폭 있지만 95~100% 정확도 범위에서 안정적 유지
- ResNet50, VGG16 은 크게 흔들리며 일관성 없는 예측 결과

결론: 'Pistachio 처럼 미묘한 시각적 차이를 분류하는 고해상도 이진 분류 문제에서는 EfficientNetB3 의 미세 특성 감지 능력과 해상도 대응력이 강하게 발휘되는 것을 확인할 수 있다. ResNet50 은 그나마 따라가지만, VGG16 은 확실히 약하다.

4.5 정리

VGG16 은 단순한 구조와 많은 파라미터 수로 인해 데이터셋의 복잡도에 비해 과도한 표현력을 가지며, 이로 인해 모든 실험에서 훈련 성능은 높지만 검증 성능은 낮은 과적합 경향을 보였다. 반면 ResNet50 은 복잡한 데이터셋에서는 안정적인 학습 성능을 보였지만, 클래스 수가 적거나 미세한 차이를 구분해야 하는 경우에는 학습 진폭이 커지며 불안정한 결과를 보이기도 했다. 이는 ResNet 의 깊은 구조와 표현력의 이점이 특정 상황에서는 과도하거나 부적절하게 작용할 수 있음을 시사한다.

4.6 추가 연구

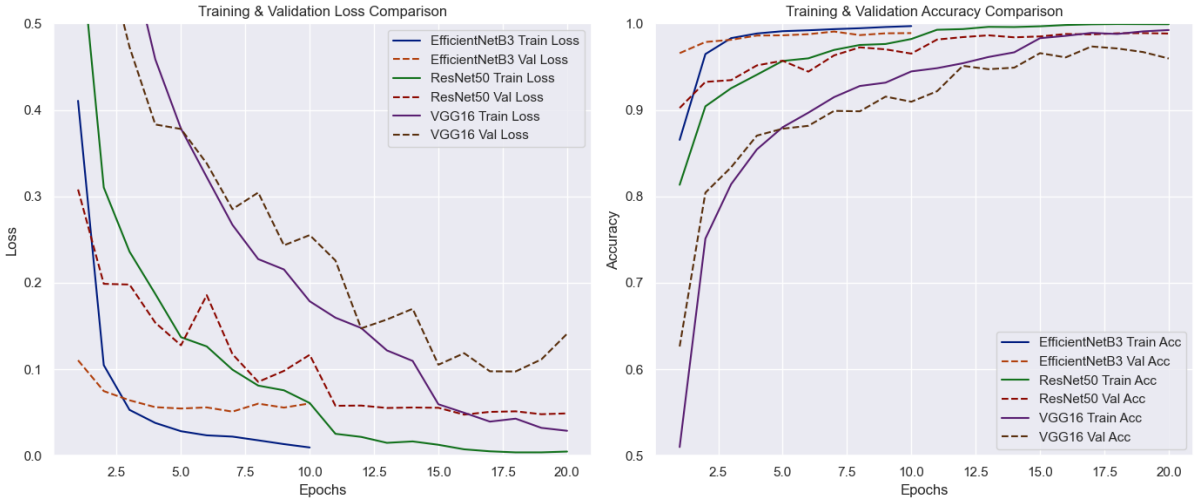
모델마다 최적의 레이어를 쌓아 커스텀하여 성능을 최대한 끌어올려 보았다.

```
################################################
# 3. 모델 3 개 정의
################################################
# 모델 예시 1: EfficientNetB3 기반 모델
```

```
model1 = models.efficientnet_b3(pretrained=True)
in_features1 = model1.classifier[1].in_features
model1.classifier = nn.Sequential(
    nn.Linear(in_features1, 512),
    nn.SiLU(),                    # EfficientNet 기본 활성화 함수
    nn.Dropout(0.3),
    nn.Linear(512, class_count)
)
model1 = model1.to(device)

# 모델 예시 2: ResNet50 기반 모델
model2 = models.resnet50(pretrained=True)
in_features2 = model2.fc.in_features
model2.fc = nn.Sequential(
    nn.Linear(in_features2, 512),
    nn.BatchNorm1d(512),
    nn.ReLU(),
    nn.Dropout(0.5),

    nn.Linear(512, 256),
    nn.BatchNorm1d(256),
    nn.ReLU(),
    nn.Dropout(0.3),

    nn.Linear(256, class_count)
)
model2 = model2.to(device)

# 모델 예시 3: VGG16 기반 모델
model3 = models.vgg16(pretrained=True)
in_features3 = model3.classifier[
    0
].in_features  # VGG의 첫 번째 Fully Connected Layer 입력 크기
model3.classifier = nn.Sequential(
    nn.Linear(in_features3, 512),
    nn.BatchNorm1d(512),
    nn.ReLU(),
    nn.Dropout(0.6),

    nn.Linear(512, 128),
    nn.BatchNorm1d(128),
    nn.ReLU(),
    nn.Dropout(0.4),

    nn.Linear(128, class_count)
)
model3 = model3.to(device)
```

EfficientNet 은 이미 깊고 복잡한 네트워크이고, SiLU 활성화 + DropConnect 가 내장되어 있다. 따라서 head 는 간단하고 가볍게, BatchNorm 은 피하고, SiLU 와 Dropout 위주로 구성되었다. ResNet 은 skip connection 으로 깊은 네트워크의 학습 안정성이 뛰어나고, BatchNorm → ReLU 조합이 매우 효과적이기 때문에 이를 활용한 head 를 구성했다. VGG 는 구조가 단순하고 파라미터가 많아서 과적합에 민감한 특성이 있어 강한 Dropout, BatchNorm, 그리고 차원을 줄이는 방식이 중요하다.

Tomato Disease Multiple Sources



Loss 그래프

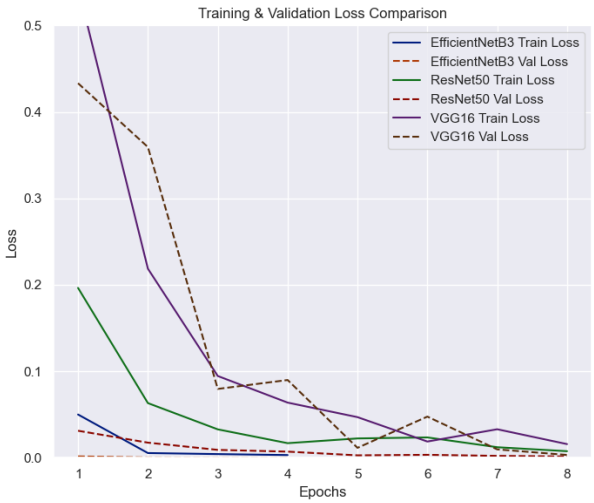
- EfficientNetB3 와 ResNet50 이 빠르게 안정적인 val loss 에 도달
- VGG16 은 여전히 들쭉날쭉한 과적합 경향 보이지만, 이전보다는 훨씬 완만해짐
- 전체적으로 커스텀 이후 VGG 의 val loss 개선됨
- EfficientNetB3 는 거의 초반부터 가장 낮은 loss 유지
-

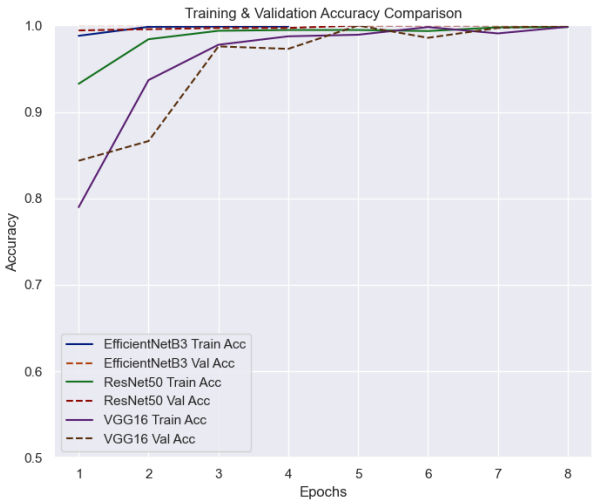
Accuracy 그래프

- 모든 모델이 약 90% 이상에서 수렴
- EfficientNetB3 가 가장 빠르게 95%에 도달 후 안정 유지
- ResNet50 도 비슷하지만 1~2 epoch 더 필요함
- VGG16 은 개선되긴 했지만 여전히 다른 모델보다 느림

결과: 커스텀 이후 VGG 도 조금 개선되었지만, 복잡한 다중 클래스 문제에서는 EfficientNet-B3 > ResNet50 > VGG16 순으로 여전히 성능 차이 존재 EfficientNet 은 여전히 가장 빠르고 안정적인 수렴을 보임

Rice Plant Diseases





Loss 그래프

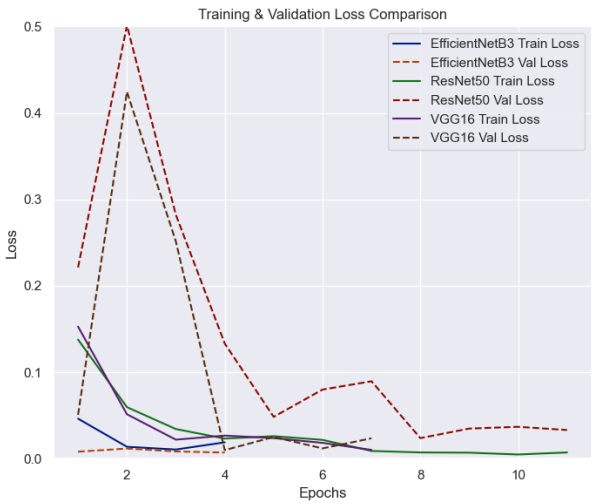
- 모든 모델이 빠르게 낮은 loss 로 수렴
- EfficientNetB3 와 ResNet50 은 거의 동일한 수준의 안정적 수렴
- VGG16 도 과적합 없이 안정적인 loss 감소 보임 → 커스텀 효과 확실히 있음

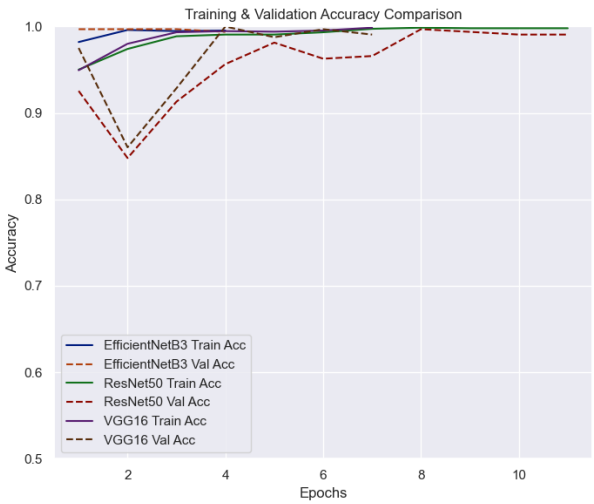
Accuracy 그래프

- 모든 모델이 약 95~100% 사이에서 수렴
- ResNet50 과 EfficientNetB3 는 거의 차이 없음
- VGG16 도 비슷한 수준까지 따라옴

결과: 클래스 수가 적고 구조가 단순한 데이터셋에서는 세 모델 모두 성능 상향 평준화되었다. 특히 커스텀 이후 VGG 도 과적합 없이 좋은 성능을 보이는 것으로 나타났다. 하지만 여전히 EfficientNetB3 가 가장 안정적인 수렴 패턴 보인다.

Pistachio Image Dataset





Loss 그래프

- EfficientNetB3 는 초반부터 val loss 가 낮고 빠르게 수렴
- ResNet50 도 안정적으로 감소하지만 EfficientNet 보단 느림
- VGG16 은 다른 모델에 비해 loss 가 조금 높지만, 이전보다 훨씬 덜 불안정

Accuracy 그래프
- EfficientNetB3 는 거의 100% 근처에서 안정됨
- ResNet50 도 거의 근접
- VGG16 은 90% 초중반에서 머무름, 추세는 좋지만 상한선이 존재

결과: Pistachio 처럼 미세한 차이 구분이 필요한 이진 분류에서는 여전히 EfficientNet-B3 가 미세 패턴 학습에 강한 구조적 장점을 보인다. ResNet50 도 잘 작동하지만 일부에서 변동성 있다.VGG 는 커스텀으로 과적합은 줄었지만, 표현력 한계가 여전하다.

5. 결과 및 분석

본 연구에서는 CNN 아키텍처 발전의 대표적인 세 가지 모델인 EfficientNet-B3, ResNet-50, VGG16 을 대상으로 토마토 잎 질병(Tomato Disease Multiple Sources), 벼 잎 질병(Rice Plant Diseases), 피스타치오 품질(Pistachio Image Dataset)의 서로 다른 이미지 데이터셋에서 성능을 비교 분석하였다. 특히, 각 모델의 아키텍처적 특징을 고려한 커스텀 설계를 통해 최적의 성능을 발휘할 수 있도록 하였다.

먼저, 복잡하고 클래스가 많은 Tomato Disease Multiple Sources 데이터셋의 경우 EfficientNet-B3 가 빠르게 높은 정확도에 도달하며 가장 안정적인 학습 곡선을 보였다. ResNet-50 도 높은 성능을 보였으나 초기 수렴 속도가 EfficientNet-B3 에 비해 상대적으로 느렸다. VGG16 의 경우 과적합 현상이 심각하게 나타났지만, 모델 구조를 최적화한 이후에는 검증 데이터의 손실이 완화되어 과적합이 어느 정도 개선되었다. 다만, 여전히 다른 두 모델에 비해 성능이 낮게 유지되는 한계를 보였다.

두 번째로, 비교적 간단한 구조의 Rice Plant Diseases 데이터셋에서는 모든 모델이 우수한 성능을 나타냈다. EfficientNet-B3 와 ResNet-50 은 모두 빠르고 안정적으로 95% 이상의 정확도를 기록하였으며, VGG16 또한 구조 커스터마이징 후 과적합 없이 안정적인 학습 성과를 보였다. 이 결과는 데이터셋의 복잡도가 낮고 클래스 구분이 명확한 상황에서는 대부분의 CNN 모델들이 안정적인 성능을 나타낼 수 있음을 시사한다.

마지막으로, 미세한 시각적 차이를 구분해야 하는 Pistachio Image Dataset 에서는 EfficientNet-B3 가 뚜렷한 우위를 보이며 거의 100%에 근접하는 검증 정확도를 나타냈다. ResNet-50 역시 준수한 성능을 보였지만 약간의 불안정성을 보였으며, VGG16 은 과적합이 개선되었음에도 불구하고 여전히 최상위 모델과는 유의미한 성능 격차가 나타났다. 이 결과는 EfficientNet-B3 가 구조적 특성상 미세한 시각적 특징 감지에 뛰어난 능력을 가지고 있음을 나타내며, 반대로 VGG16 과 ResNet-50 이 미세한 시각적 차이에서는 성능이 제한될 수 있음을 시사한다.

종합적으로, 본 실험에서 EfficientNet-B3 는 모든 데이터셋에서 우수하거나 가장 뛰어난 성능을 보였으며, ResNet-50 은 데이터셋 특성에 따라 성능이 다소 변화했다. VGG16 의 경우는 과적합이 빈번히 발생하였지만, 모델 커스터마이징을 통해 어느 정도 성능 개선이 가능함을 확인할 수 있었다.

6. 결론

본 연구를 통해 CNN 아키텍처의 발전이 실제 이미지 분류 문제에서의 성능 향상에 어떠한 영향을 주는지 확인할 수 있었다. 특히 EfficientNet-B3 의 뛰어난 성능은 네트워크의 깊이, 너비, 이미지 해상도를 동시에 조정하는 Compound Scaling 기법이 우수한 일반화 성능과 안정적인 학습 결과를 가져온다는 것을 입증하였다. 이는 최신 CNN 모델 설계가 단순히 파라미터 수 증가나 깊이 증가가 아니라, 아키텍처 요소들의 균형 잡힌 조정을 통해 성능과 효율성을 모두 극대화할 수 있음을 보여준다.

반면, VGG16 은 과도하게 많은 파라미터와 단순한 구조로 인해 과적합이 반복적으로 발생했다. 특히, 복잡한 데이터셋이나 미세한 시각적 특징을 요구하는 데이터셋에서 이러한 현상이 두드러졌다. 모델의 구조 최적화를 통해 과적합이 일부 완화되었으나, 근본적인 한계를 해결하기에는 어려움이 있었다. 이는 CNN 모델 선택 시 반드시 데이터셋의 복잡도와 클래스 특성을 고려해야 함을 시사한다.

ResNet-50 의 경우 안정적이고 깊은 네트워크 구조 덕분에 복잡한 데이터셋에서는 좋은 성능을 보였으나, 비교적 간단하거나 미세한 특징이 중요한 문제에서는 다소 불안정한 성능을 보였다. 이 결과는 깊고 복잡한 구조의 CNN 이 항상 최선의 선택이 아닐 수도 있으며, 데이터의 특성과 과업의 목적에 따라 적절한 모델 구조 선택과 커스터마이징이 필요함을 의미한다.

본 연구의 결과를 종합적으로 고려할 때, CNN 모델의 선택과 설계 과정에서 가장 중요한 것은 데이터셋의 특성(클래스 수, 데이터의 복잡성, 미세한 특징의 중요도)을 명확히 분석한 뒤 이에 가장 적합한 아키텍처를 선정하고 세부적인 구조를 조정하는 것이다. EfficientNet-B3 가 일반적으로 추천되지만, 데이터와 목적에 따라 ResNet 또는 VGG 계열 모델의 활용 및 개선도 여전히 의미가 있을 것으로 판단된다.

향후 연구에서는 더 다양한 데이터셋과 모델 구조를 활용하여 보다 세부적인 성능 분석을 진행할 필요가 있다. 또한 모델의 연산 효율성 및 실제 배포 환경에서의 성능 평가를 추가로 고려한다면 보다 종합적인 아키텍처 선정 기준을 마련할 수 있을 것으로 기대된다.

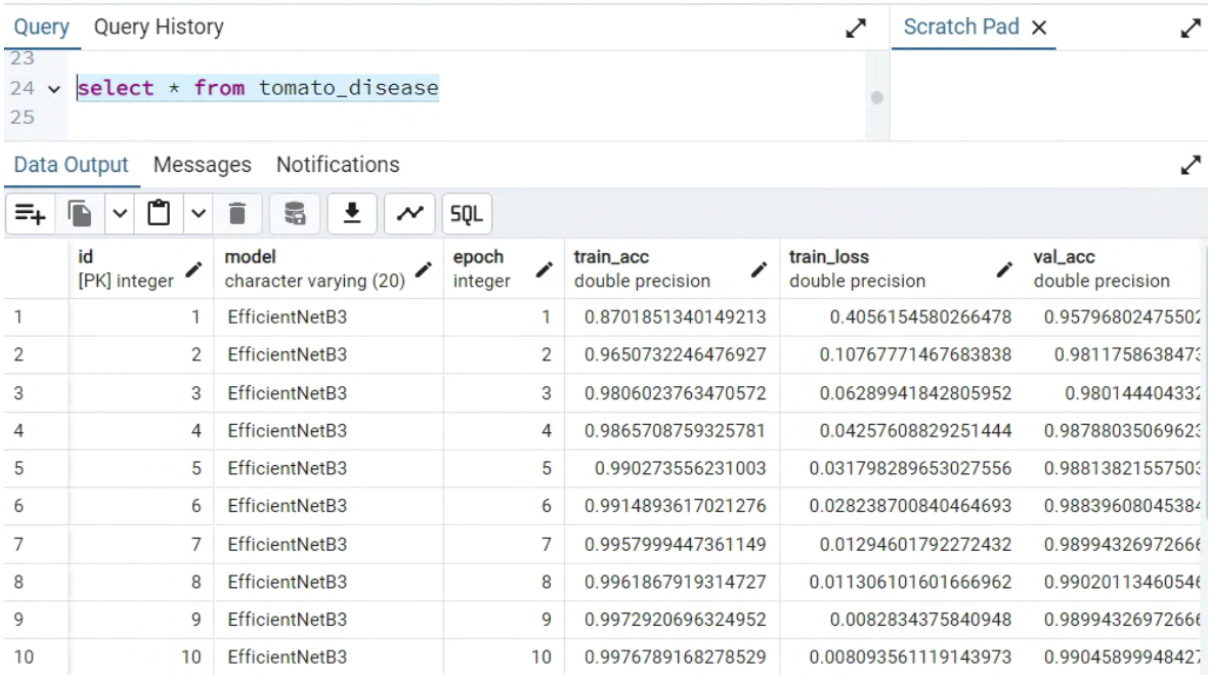
7. DB 저장

데이터베이스에 모델 별 성능평가 저장을 하기위해 PostgreSQL 을 활용하였다. pgAdmin 을 통해 ktb_project 데이터베이스를 생성하고 tomato_disease, pistachio, rice_disease 테이블을 생성했다. 파이썬의 psycopg2 라이브러리를 통해 로컬 DB 와 연결하여 값을 저장 할 수 있었다.

Tomato_disease, Rice_disease, pistachio

	Column_name	Data_type	Property	Is_nullable
1	Id	Serial	Primary key	No
2	Model	Varchar(20)		Yes
3	epoch	int		Yes
4	Train_acc	float		Yes
5	Train_loss	Float		Yes
6	Val_acc	Float		Yes
7	Val_loss	float		Yes

아래 사진으로 값이 잘 저장되었음을 확인할 수 있다.



	id [PK] integer	model character varying (20)	epoch integer	train_acc double precision	train_loss double precision	val_acc double precision
1	1	EfficientNetB3	1	0.8701851340149213	0.4056154580266478	0.9579680247550
2	2	EfficientNetB3	2	0.9650732246476927	0.10767771467683838	0.981175863847
3	3	EfficientNetB3	3	0.9806023763470572	0.06289941842805952	0.98014440433
4	4	EfficientNetB3	4	0.9865708759325781	0.04257608829251444	0.9878803506962
5	5	EfficientNetB3	5	0.990273556231003	0.031798289653027556	0.9881382155750
6	6	EfficientNetB3	6	0.9914893617021276	0.028238700840464693	0.9883960804538
7	7	EfficientNetB3	7	0.9957999447361149	0.01294601792272432	0.9899432697266
8	8	EfficientNetB3	8	0.9961867919314727	0.011306101601666962	0.9902011346054
9	9	EfficientNetB3	9	0.9972920696324952	0.0082834375840948	0.9899432697266
10	10	EfficientNetB3	10	0.9976789168278529	0.008093561119143973	0.9904589994842

부록.

1. EfficientNet-B3 아키텍처 상세 구조

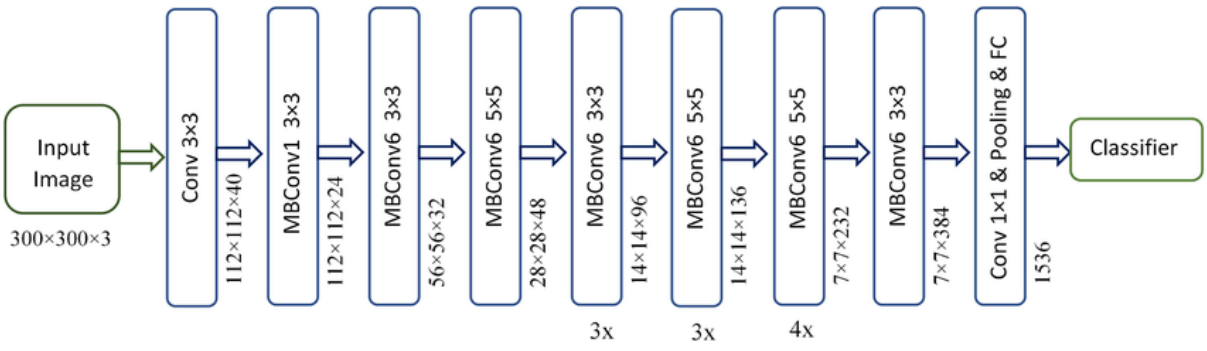
EfficientNet 은 MobileNetV2 의 inverted bottleneck 구조를 기반으로 다수의 MBConv 블록(Mobile Inverted Bottleneck Convolution)을 쌓은 구조이다.

기본 구성 블록 (MBConv)

- 1×1 expand → depthwise conv → 1×1 project
- Swish (SiLU) 활성화 함수 사용
- SE (Squeeze-and-Excitation) block 내장 → 채널 중요도 반영
- DropConnect 적용 → regularization
- 총 ≈ 12M 파라미터
- 입력 해상도 기본 300×300
- 출력: 최종 nn.Linear(1536, class_count)

EfficientNetB3 Archicheture Table

Stage	Operator	Output Size	Channels	Layers
Input	Conv3x3	150x150	40	1
1	MBConv1	150x150	24	1
2	MBconv6	75x75	32	2
3	MBconv6	38x38	48	2
4	MBconv6	19x19	96	3
5	MBconv6	10x10	136	3
6	MBconv6	10x10	232	4
7	MBconv6	5x5	384	1
Head	Conv1x1 + Pool + FC	1x1	1536	1



1-1 Compound Scaling (컴파운드 스케일링)

Compound Scaling 은 CNN 모델의 성능을 높이기 위해 네트워크의 깊이(Depth), 너비(Width), 입력 해상도(Resolution)를 균형 있게 함께 증가시키는 방법이다. 기존 CNN 모델은 주로 깊이 또는 너비 중 한 가지 요소만 늘려왔지만, Compound Scaling 은 이 세 가지 요소를 일정한 비율(Scale Factor)에 따라 균형 있게 동시에 확장하여 효율성을 극대화한다.

1-2 SiLU (Sigmoid Linear Unit, Swish)

입력 x 를 시그모이드 함수 σ(x) 값과 곱한 형태로, Swish 라고도 불린다. ReLU 의 변형이며, 부드러운 곡선 형태를 가지고 있다. ReLU 에 비해 더 부드럽고 연속적인 미분 가능성을 갖춰서 학습 안정성을 높임. 특히 깊은 신경망에서 gradient flow 가 더 원활하게 이루어짐. EfficientNet, MobileNetV3 같은 최신 아키텍처에서 ReLU 보다 더 우수한 성능으로 널리 쓰임.

1-3 DropConnect (드롭커넥트)

DropConnect 는 일반적인 Dropout 과 유사한 정규화(regularization) 기법이지만, 노드가 아니라 가중치(weight)를 랜덤하게 drop 하여 학습 중 네트워크를 랜덤하게 변화시키는 방식이다. 일반적인 dropout 보다 강력한 정규화 효과 네트워크의 과적합(overfitting)을 줄이고 일반화 성능(generalization)을 향상시킴

학습 코드

```
import time, itertools, copy, warnings
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import Adamax
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

warnings.filterwarnings('ignore')

############################################
# 1. 데이터셋 준비
############################################
```

```
# 데이터셋 경로
train_dir = "D:/kaggle_dataset/tomato_disease/train"
# train_dir = "D:/kaggle_dataset/rice/train"
# train_dir = "D:/kaggle_dataset/pistachio/train"

# 전처리 정의
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

val_test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# 1. 라벨만 얻기 위한 dataset (transform=None)
base_dataset = datasets.ImageFolder(root=train_dir)
targets = [s[1] for s in base_dataset.samples]
indices = list(range(len(base_dataset)))

# Stratified split: Train 70%, Val 15%, Test 15%
train_idx, temp_idx = train_test_split(indices, train_size=0.7,
stratify=targets, random_state=123)
temp_targets = [targets[i] for i in temp_idx]
val_idx, test_idx = train_test_split(temp_idx, test_size=0.5,
stratify=temp_targets, random_state=123)

# 2. transform 이 적용된 버전 3 개 불러오기
full_train = datasets.ImageFolder(root=train_dir,
transform=train_transform)
full_val   = datasets.ImageFolder(root=train_dir,
transform=val_test_transform)
full_test  = datasets.ImageFolder(root=train_dir,
transform=val_test_transform)

# 3. Subset 적용
train_dataset = Subset(full_train, train_idx)
valid_dataset = Subset(full_val, val_idx)
test_dataset  = Subset(full_test, test_idx)

# 4. DataLoader 생성
```

```
batch_size = 40
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False, num_workers=2)

# 테스트 배치 사이즈 자동 설정 (최대 80 이하)
ts_length = len(test_dataset)
possible_bs = [ts_length // n for n in range(1, ts_length+1) if
ts_length % n == 0 and (ts_length / n) <= 80]
test_batch_size = max(possible_bs) if possible_bs else batch_size
test_loader = DataLoader(test_dataset, batch_size=test_batch_size,
shuffle=False, num_workers=2)

# 정리
dataloaders = {'train': train_loader, 'val': valid_loader}
dataset_sizes = {'train': len(train_dataset), 'val':
len(valid_dataset)}
class_count = len(full_train.classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

##############################################
# 2. 학습 함수 정의
##############################################
def train_model(model, dataloaders, criterion, optimizer, scheduler,
num_epochs=5, device=device):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [],
'val_acc': []}

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 10)
        # 각 epoch 마다 train 과 validation 단계 진행
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()
            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
```

```
                optimizer.zero_grad()
                with torch.set_grad_enabled(phase=='train'):
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    _, preds = torch.max(outputs, 1)
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() /
dataset_sizes[phase]
            print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')
            if phase == 'train':
                history['train_loss'].append(epoch_loss)
                history['train_acc'].append(epoch_acc.item())
            else:
                history['val_loss'].append(epoch_loss)
                history['val_acc'].append(epoch_acc.item())
                scheduler.step(epoch_loss)
                # best model 저장
                if epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_model_wts = copy.deepcopy(model.state_dict())
        print()
    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed//60:.0f}m
{time_elapsed%60:.0f}s')
    print(f'Best val Acc: {best_acc:.4f}')
    model.load_state_dict(best_model_wts)
    return model, history
############################################
# 3. 모델 3개 정의
############################################
# 모델 예시 1: EfficientNetB3 기반 모델
model1 = models.efficientnet_b3(pretrained=True)
in_features1 = model1.classifier[1].in_features
model1.classifier = nn.Sequential(
    nn.BatchNorm1d(in_features1),
    nn.Linear(in_features1, 256),
    nn.ReLU(),
    nn.Dropout(0.45),
    nn.Linear(256, class_count)
)
model1 = model1.to(device)
```

```
# 모델 예시 2: ResNet50 기반 모델
model2 = models.resnet50(pretrained=True)
in_features2 = model2.fc.in_features
model2.fc = nn.Sequential(
    nn.BatchNorm1d(in_features2),
    nn.Linear(in_features2, 256),
    nn.ReLU(),
    nn.Dropout(0.45),
    nn.Linear(256, class_count)
)
model2 = model2.to(device)

# 모델 예시 3: VGG16 기반 모델
model3 = models.vgg16(pretrained=True)
in_features3 = model3.classifier[0].in_features  # VGG의 첫 번째 Fully Connected Layer 입력 크기
model3.classifier = nn.Sequential(
    nn.BatchNorm1d(in_features3),
    nn.Linear(in_features3, 256),
    nn.ReLU(),
    nn.Dropout(0.45),
    nn.Linear(256, class_count) # 최종 레이어: 클래스 개수로 출력 조정
)
model3 = model3.to(device)
##############################################
# 4. 모델 학습 및 결과 저장
##############################################
import gc

num_epochs = 5
criterion = nn.CrossEntropyLoss()
histories = {}

# EfficientNetB3 학습
optimizer1 = Adamax(model1.parameters(), lr=0.001)
scheduler1 = ReduceLROnPlateau(optimizer1, mode='min', factor=0.5,
patience=1, verbose=True)
print("Training EfficientNetB3 Model")
model1, history1 = train_model(model1, dataloaders, criterion,
optimizer1, scheduler1, num_epochs)
histories['EfficientNetB3'] = history1

# 로컬 그래픽카드 메모리 부족 아슈..
gc.collect()
torch.cuda.empty_cache()
```

```
# ResNet50 학습
optimizer2 = Adamax(model2.parameters(), lr=0.001)
scheduler2 = ReduceLROnPlateau(optimizer2, mode='min', factor=0.5,
patience=1, verbose=True)
print("Training ResNet50 Model")
model2, history2 = train_model(model2, dataloaders, criterion,
optimizer2, scheduler2, num_epochs)
histories['ResNet50'] = history2

gc.collect()
torch.cuda.empty_cache()

# vgg16 학습
optimizer3 = Adamax(model3.parameters(), lr=0.001)
scheduler3 = ReduceLROnPlateau(optimizer3, mode='min', factor=0.5,
patience=1, verbose=True)
print("Training VGG16 Model")
model3, history3 = train_model(model3, dataloaders, criterion,
optimizer3, scheduler3, num_epochs)
histories['VGG16'] = history3

gc.collect()
torch.cuda.empty_cache()
```

```
##############################################
# 결과 시각화
##############################################

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 6))
sns.set_theme(style="darkgrid", palette="dark", context="notebook")

# Loss 비교
plt.subplot(1, 2, 1)
for model_name, hist in histories1.items():
    epochs = range(1, len(hist['train_loss']) + 1)
    plt.plot(epochs, hist['train_loss'], label=f'{model_name} Train
Loss')
    plt.plot(epochs, hist['val_loss'], '--', label=f'{model_name} Val
Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.ylim(0, 0.5)
plt.title('Training & Validation Loss Comparison')
```

```
plt.legend()

# Accuracy 비교
plt.subplot(1, 2, 2)
for model_name, hist in histories1.items():
    epochs = range(1, len(hist['train_acc']) + 1)
    plt.plot(epochs, hist['train_acc'], label=f'{model_name} Train
Acc')
    plt.plot(epochs, hist['val_acc'], '--', label=f'{model_name} Val
Acc')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training & Validation Accuracy Comparison')
plt.ylim(0.5, 1)
plt.legend()

plt.tight_layout()
plt.show()
```

```
##############################################
# DB 연동 및 저장
##############################################

import psycopg2
from psycopg2 import sql

# 데이터베이스 연결 설정
conn = psycopg2.connect(
    dbname="ktb_project",
    user="postgres",
    password="postgres",
    host="localhost",
    port="5432"
)

# 커서 생성
cur = conn.cursor()


# 각 모델에 대해 데이터 삽입
for model, values in histories1.items():
    for epoch in range(len(values['train_loss'])):
        cur.execute(
            sql.SQL("""
            INSERT INTO rice_disease (model, epoch, train_acc,
train_loss, val_acc, val_loss)
```

```
            VALUES (%s, %s, %s, %s, %s, %s)
            """),
            (
                model,
                epoch + 1,
                values['train_acc'][epoch],
                values['train_loss'][epoch],
                values['val_acc'][epoch],
                values['val_loss'][epoch]
            )
        )

# 변경사항 커밋
conn.commit()

# 연결 종료
cur.close()
conn.close()

print("데이터가 성공적으로 삽입되었습니다.")
```