🥦 채소 이미지 분류 - Pretrained 모델 비교 실험

이동재

2025년 3월 30일

목차

1. 서론

본 연구의 목적은 다양한 채소 이미지 데이터셋을 활용하여 대표 채소(예: 토마토, 오이, 당근 등)의 종류를 자동으로 분류하는 CNN 기반 모델을 개발하는 것이다. 전통적인 채소 분류 방식은 인력 소요와 시간이 많이 들기 때문에, 자동화된 분류 시스템을 통해 농가나 유통업체의 품질 관리 및 생산 효율성을 크게 향상시킬 수 있다. 본 보고서에서는 CNN 모델과 사전 훈련된 모델(전이 학습 기법)을 적용하여 채소 분류 정확도를 극대화하는 방법을 제시한다.

2. 데이터셋 설명

연구에 사용된 채소 이미지 데이터셋은 Kaggle 또는 기타 공개 데이터 소스에서 제공되는 채소 이미지 컬렉션을 기반으로 한다. 주요 특징은 다음과 같다.

- 이미지 해상도 및 형식**:** 모든 이미지는 고해상도(예: 300×300 픽셀)의 PNG 또는 JPEG 형식으로 제공되어, 채소의 세부 특징(색상, 질감 등)을 명확하게 확인할 수 있다.
- 클래스 구성**:** 데이터셋은 대표적인 채소 종류(예: 토마토, 오이, 당근 등)로 구성되어 있으며, 클래스 간 균형을 맞추어 데이터 불균형 문제를 최소화하였다.
- 촬영 조건**:** 다양한 각도, 배경 및 조명 조건에서 촬영된 이미지들이 포함되어 있어, 실제 환경에서의 분류 성능을 높일 수 있다.

이 데이터셋은 채소 분류뿐 아니라 객체 인식 및 품종 구분 연구에도 활용될 수 있으며, 농업 자동화 및 품질 관리 시스템 개발에 기여할 수 있다.

총 15000개의 이미지에서 클래스 수는 15개이다.

3. 모델 설명

1. Resnet18
    1. ResNet18은 18개의 층을 가진 잔차 네트워크(Residual Network)입니다.
        - 주요 특징**:**
            - 잔차 연결**(Residual Connections):** 깊은 네트워크에서 발생하는 기울기 소실 문제를 완화하기 위해, 입력 정보를 건너뛰어 다음 층으로 전달하는 skip connection을 사용합니다.
            - 구조**:** 비교적 간단한 구조로 이미지 분류, 객체 검출 등 다양한 비전 작업에서 기본 모델로 사용됩니다.

2. MobileNetV2
    1. MobileNetV2는 경량화된 모델로, 모바일 및 임베디드 디바이스에서 효율적으로 동작하도록 설계되었습니다.
        - 주요 특징:
            - 깊이별 분리 합성곱(**Depthwise Separable Convolutions**): 표준 합성곱보다 연산량과 파라미터 수를 크게 줄이면서도 성능을 유지합니다.
            - 인버티드 잔차 구조(**Inverted Residual Structure**): 낮은 차원에서 높은 차원으로 확장한 후 다시 낮은 차원으로 축소하는 구조를 사용하여 효율성을 높입니다.
3. EfficientNetB0
    1. EfficientNetB0는 EfficientNet 계열의 기본 모델로, 모델의 크기와 연산 효율성을 동시에 고려한 설계 방법을 사용합니다.
        - 주요 특징:
            - 컴파운드 스케일링(**Compound Scaling**): 네트워크의 깊이, 너비, 해상도를 동시에 균형 있게 확장하여 최적의 성능과 효율성을 달성합니다.
            - 효율성: 상대적으로 적은 파라미터와 연산량으로도 우수한 성능을 보여, 다양한 비전 애플리케이션에 적합합니다.
4. GoogleNet
    1. GoogleNet은 Inception 모듈을 도입한 CNN 모델로, 다양한 크기의 필터를 동시에 사용하여 여러 스케일의 특징을 추출합니다.
        - 주요 특징:
            - **Inception** 모듈: 여러 병렬 합성곱과 풀링 계층을 결합해, 서로 다른 크기의 특징을 동시에 학습할 수 있습니다.
            - 효율적인 계산: 깊이가 22층에 달하면서도 계산 비용을 줄인 구조로 이미지 분류 및 객체 인식 분야에서 큰 성과를 보였습니다.
5. Deep CNN
    1. Deep CNN은 여러 개의 합성곱 계층을 깊게 쌓은 구조의 신경망을 의미하며, 특정 모델 이름보다는 네트워크의 깊이에 초점을 맞춥니다.
        - 주요 특징:

- 계층적 특징 학습: 낮은 계층에서는 간단한 패턴을, 깊은 계층에서는 복잡한 패턴을 학습하여 이미지의 다양한 특성을 효과적으로 추출합니다.
- 다양한 응용 분야: 이미지 분류, 객체 검출, 시맨틱 세분화 등 다양한 컴퓨터 비전 작업에서 사용됩니다.

4. 실험 방법

1. 데이터 전처리 및 증강
    1. 이미지 전처리 224 x 224로 리사이즈하고, 픽셀 값 정규화 (0 ~ 1 범위)로 수정
2. 모델 학습
    1. Pre-trained 모델 불러오기.

```python
## 5. 모델 정의 및 학습 실행
model_list = {
    'ResNet18': models.resnet18(pretrained=True),
    'MobileNetV2': models.mobilenet_v2(pretrained=True),
    'EfficientNetB0': models.efficientnet_b0(pretrained=True),
    'GoogLeNet': models.googlenet(pretrained=True, aux_logits=True)
}

for name, model in model_list.items():
    if 'resnet' in name.lower():
        model.fc = nn.Linear(model.fc.in_features, NUM_CLASSES)
    elif 'mobilenet' in name.lower():
        model.classifier[1] = nn.Linear(model.classifier[1].in_features, NUM_CLASSES)
    elif 'efficientnet' in name.lower():
        model.classifier[1] = nn.Linear(model.classifier[1].in_features, NUM_CLASSES)
    elif 'googlenet' in name.lower():
        model.fc = nn.Linear(model.fc.in_features, NUM_CLASSES)

create_training_result_tables()
results = []
for name, model in model_list.items():
    result = train_model(model, name, epochs=10)
    results.append(result)
    print(results)
```

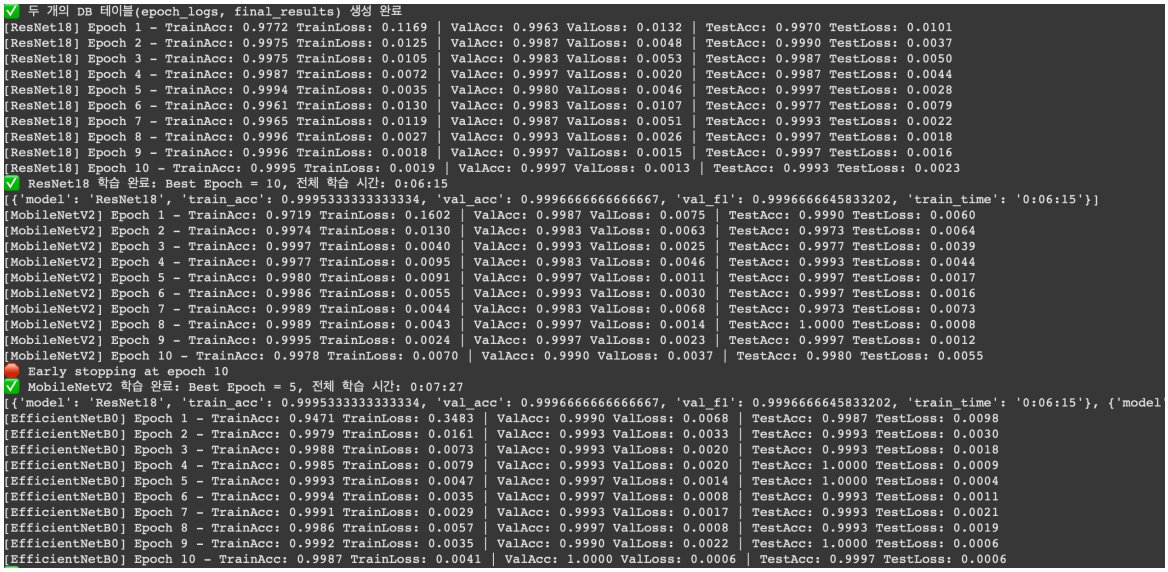
    2. Early stopping 걸어주기

```python
        # === Early Stopping 체크 ===
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_epoch = epoch + 1
            early_stop_counter = 0
            best_metrics = (train_acc, train_loss, val_acc, val_loss, test_acc, test_loss)

            # validation 성능이 개선될 때마다 모델 저장
            torch.save(model.state_dict(), f"best_model_{model_name}.pth")
        else:
            early_stop_counter += 1
            if early_stop_counter >= patience:
                print(f"🛑 Early stopping at epoch {epoch+1}")
                break

    # 최종 결과를 DB에 기록
```

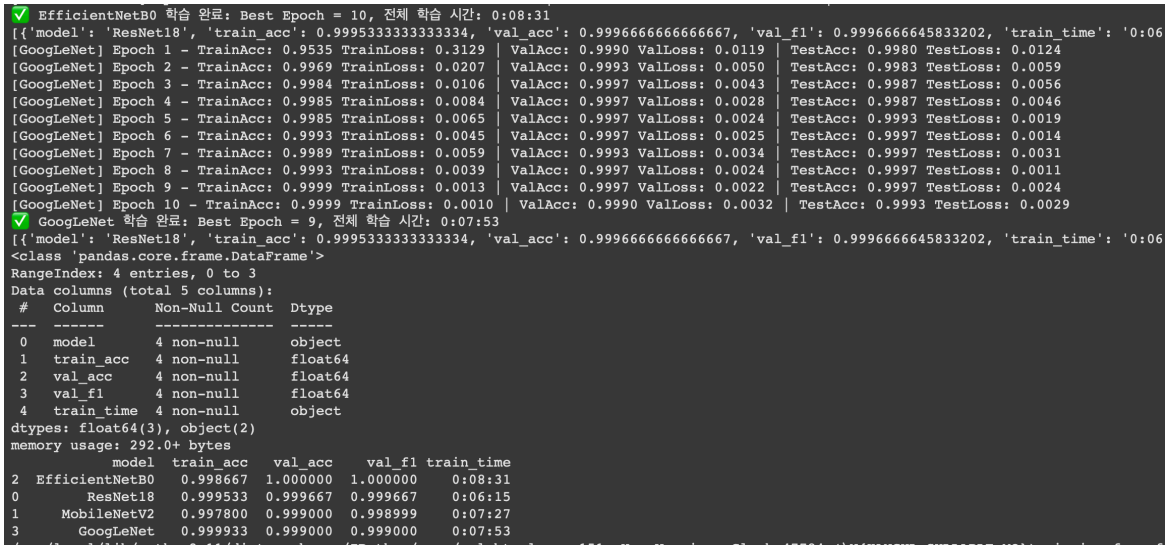
3. 모델 평가 및 분석

    1. 테스트 데이터셋을 통해 최종 모델의 분류 정확도와 손실(Loss)을 평가하였으며, 학습 과정 중 도출된 그래프를 분석하여 모델의 수렴 상태와 과적합 여부를 확인하였다.

5. 결과 및 분석

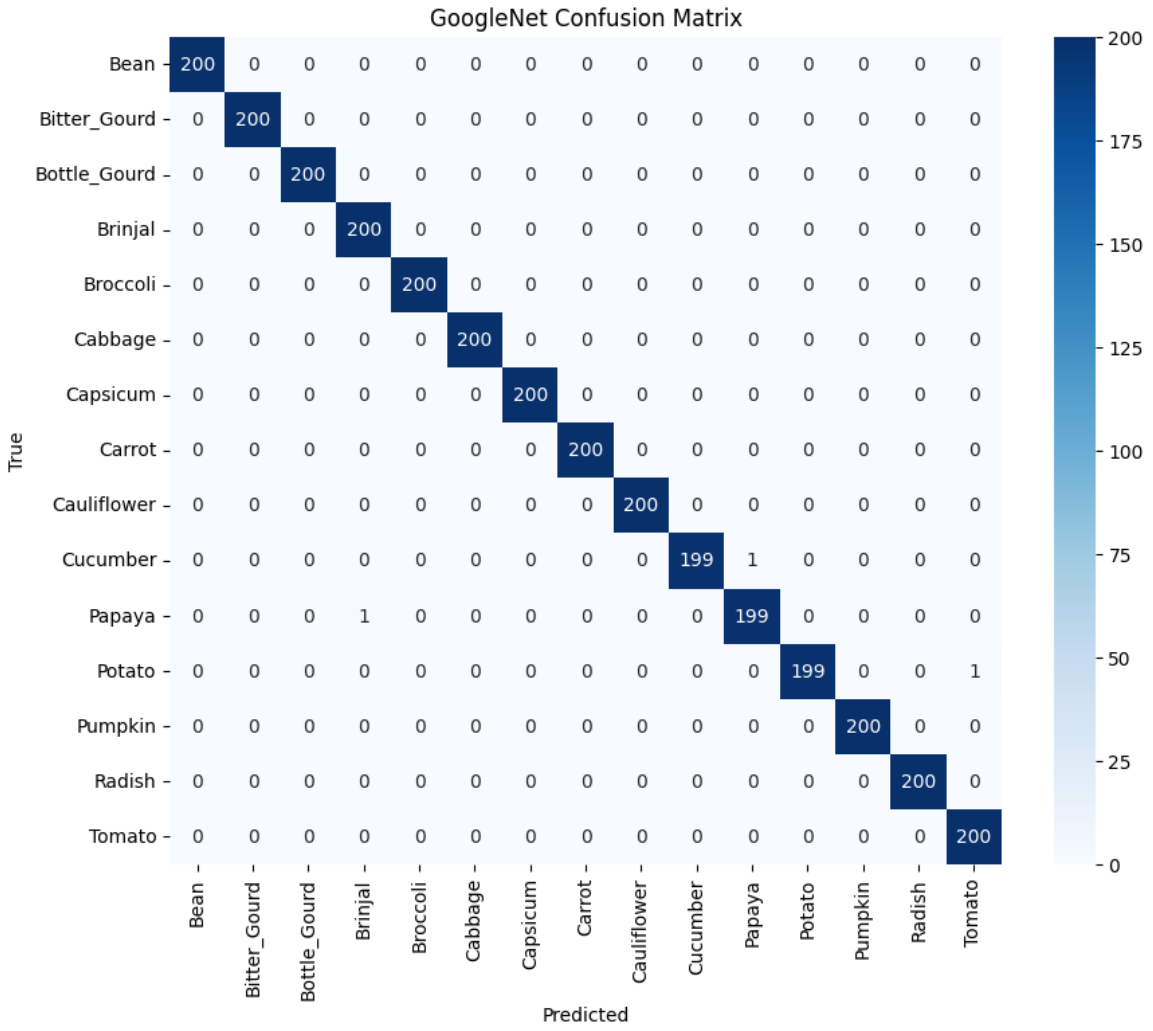


```
✅ 두 개의 DB 테이블(epoch_logs, final_results) 생성 완료
[ResNet18] Epoch 1 - TrainAcc: 0.9772 TrainLoss: 0.1169 | ValAcc: 0.9963 ValLoss: 0.0132 | TestAcc: 0.9970 TestLoss: 0.0101
[ResNet18] Epoch 2 - TrainAcc: 0.9975 TrainLoss: 0.0125 | ValAcc: 0.9987 ValLoss: 0.0048 | TestAcc: 0.9990 TestLoss: 0.0037
[ResNet18] Epoch 3 - TrainAcc: 0.9975 TrainLoss: 0.0105 | ValAcc: 0.9983 ValLoss: 0.0053 | TestAcc: 0.9987 TestLoss: 0.0050
[ResNet18] Epoch 4 - TrainAcc: 0.9987 TrainLoss: 0.0072 | ValAcc: 0.9997 ValLoss: 0.0020 | TestAcc: 0.9987 TestLoss: 0.0044
[ResNet18] Epoch 5 - TrainAcc: 0.9994 TrainLoss: 0.0035 | ValAcc: 0.9980 ValLoss: 0.0046 | TestAcc: 0.9997 TestLoss: 0.0028
[ResNet18] Epoch 6 - TrainAcc: 0.9961 TrainLoss: 0.0130 | ValAcc: 0.9983 ValLoss: 0.0107 | TestAcc: 0.9977 TestLoss: 0.0079
[ResNet18] Epoch 7 - TrainAcc: 0.9965 TrainLoss: 0.0119 | ValAcc: 0.9987 ValLoss: 0.0051 | TestAcc: 0.9993 TestLoss: 0.0022
[ResNet18] Epoch 8 - TrainAcc: 0.9996 TrainLoss: 0.0027 | ValAcc: 0.9993 ValLoss: 0.0026 | TestAcc: 0.9997 TestLoss: 0.0018
[ResNet18] Epoch 9 - TrainAcc: 0.9996 TrainLoss: 0.0018 | ValAcc: 0.9997 ValLoss: 0.0015 | TestAcc: 0.9997 TestLoss: 0.0016
[ResNet18] Epoch 10 - TrainAcc: 0.9995 TrainLoss: 0.0019 | ValAcc: 0.9997 ValLoss: 0.0013 | TestAcc: 0.9993 TestLoss: 0.0023
✅ ResNet18 학습 완료: Best Epoch = 10, 전체 학습 시간: 0:06:15
[{'model': 'ResNet18', 'train_acc': 0.9995333333333334, 'val_acc': 0.9996666666666667, 'val_f1': 0.9996666645833202, 'train_time': '0:06:15'}]
[MobileNetV2] Epoch 1 - TrainAcc: 0.9719 TrainLoss: 0.1602 | ValAcc: 0.9987 ValLoss: 0.0075 | TestAcc: 0.9990 TestLoss: 0.0060
[MobileNetV2] Epoch 2 - TrainAcc: 0.9974 TrainLoss: 0.0130 | ValAcc: 0.9983 ValLoss: 0.0063 | TestAcc: 0.9973 TestLoss: 0.0064
[MobileNetV2] Epoch 3 - TrainAcc: 0.9997 TrainLoss: 0.0040 | ValAcc: 0.9993 ValLoss: 0.0025 | TestAcc: 0.9977 TestLoss: 0.0039
[MobileNetV2] Epoch 4 - TrainAcc: 0.9977 TrainLoss: 0.0095 | ValAcc: 0.9983 ValLoss: 0.0046 | TestAcc: 0.9993 TestLoss: 0.0044
[MobileNetV2] Epoch 5 - TrainAcc: 0.9980 TrainLoss: 0.0091 | ValAcc: 0.9997 ValLoss: 0.0011 | TestAcc: 0.9997 TestLoss: 0.0017
[MobileNetV2] Epoch 6 - TrainAcc: 0.9986 TrainLoss: 0.0055 | ValAcc: 0.9993 ValLoss: 0.0030 | TestAcc: 0.9997 TestLoss: 0.0016
[MobileNetV2] Epoch 7 - TrainAcc: 0.9989 TrainLoss: 0.0044 | ValAcc: 0.9983 ValLoss: 0.0068 | TestAcc: 0.9973 TestLoss: 0.0073
[MobileNetV2] Epoch 8 - TrainAcc: 0.9989 TrainLoss: 0.0043 | ValAcc: 0.9997 ValLoss: 0.0014 | TestAcc: 1.0000 TestLoss: 0.0008
[MobileNetV2] Epoch 9 - TrainAcc: 0.9995 TrainLoss: 0.0024 | ValAcc: 0.9997 ValLoss: 0.0023 | TestAcc: 0.9997 TestLoss: 0.0012
[MobileNetV2] Epoch 10 - TrainAcc: 0.9978 TrainLoss: 0.0070 | ValAcc: 0.9990 ValLoss: 0.0037 | TestAcc: 0.9980 TestLoss: 0.0055
🛑 Early stopping at epoch 10
✅ MobileNetV2 학습 완료: Best Epoch = 5, 전체 학습 시간: 0:07:27
[{'model': 'ResNet18', 'train_acc': 0.9995333333333334, 'val_acc': 0.9996666666666667, 'val_f1': 0.9996666645833202, 'train_time': '0:06:15'}, {'model'
[EfficientNetB0] Epoch 1 - TrainAcc: 0.9471 TrainLoss: 0.3483 | ValAcc: 0.9990 ValLoss: 0.0068 | TestAcc: 0.9987 TestLoss: 0.0098
[EfficientNetB0] Epoch 2 - TrainAcc: 0.9979 TrainLoss: 0.0161 | ValAcc: 0.9993 ValLoss: 0.0033 | TestAcc: 0.9993 TestLoss: 0.0030
[EfficientNetB0] Epoch 3 - TrainAcc: 0.9988 TrainLoss: 0.0073 | ValAcc: 0.9993 ValLoss: 0.0020 | TestAcc: 0.9993 TestLoss: 0.0018
[EfficientNetB0] Epoch 4 - TrainAcc: 0.9985 TrainLoss: 0.0079 | ValAcc: 0.9993 ValLoss: 0.0020 | TestAcc: 1.0000 TestLoss: 0.0009
[EfficientNetB0] Epoch 5 - TrainAcc: 0.9993 TrainLoss: 0.0047 | ValAcc: 0.9997 ValLoss: 0.0014 | TestAcc: 1.0000 TestLoss: 0.0004
[EfficientNetB0] Epoch 6 - TrainAcc: 0.9994 TrainLoss: 0.0035 | ValAcc: 0.9997 ValLoss: 0.0008 | TestAcc: 0.9993 TestLoss: 0.0011
[EfficientNetB0] Epoch 7 - TrainAcc: 0.9991 TrainLoss: 0.0029 | ValAcc: 0.9993 ValLoss: 0.0017 | TestAcc: 0.9993 TestLoss: 0.0021
[EfficientNetB0] Epoch 8 - TrainAcc: 0.9986 TrainLoss: 0.0057 | ValAcc: 0.9997 ValLoss: 0.0008 | TestAcc: 0.9993 TestLoss: 0.0019
[EfficientNetB0] Epoch 9 - TrainAcc: 0.9992 TrainLoss: 0.0035 | ValAcc: 0.9990 ValLoss: 0.0022 | TestAcc: 1.0000 TestLoss: 0.0006
[EfficientNetB0] Epoch 10 - TrainAcc: 0.9987 TrainLoss: 0.0041 | ValAcc: 1.0000 ValLoss: 0.0006 | TestAcc: 0.9997 TestLoss: 0.0006
```

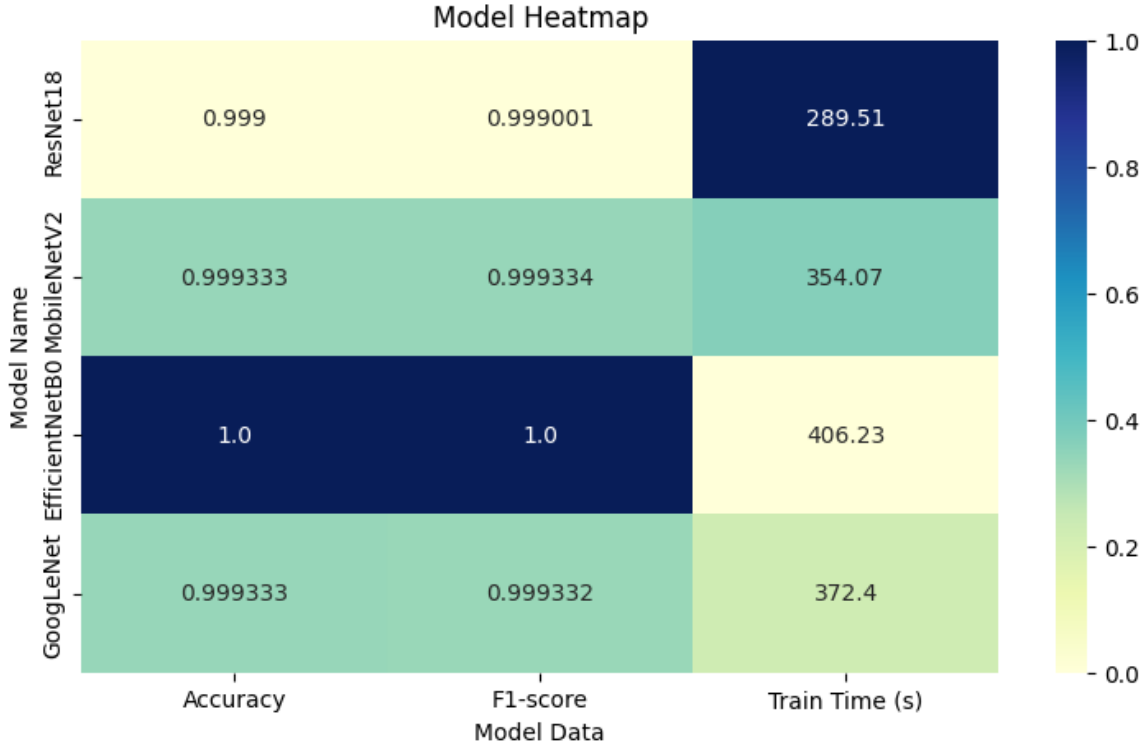


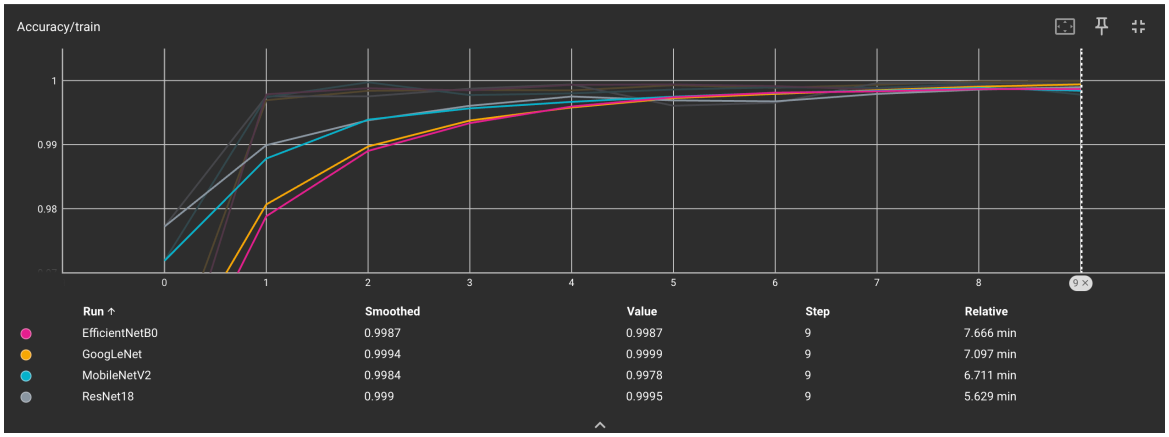
```
✅ EfficientNetB0 학습 완료: Best Epoch = 10, 전체 학습 시간: 0:08:31
[{'model': 'ResNet18', 'train_acc': 0.9995333333333334, 'val_acc': 0.9996666666666667, 'val_f1': 0.9996666645833202, 'train_time': '0:06
[GoogLeNet] Epoch 1 - TrainAcc: 0.9535 TrainLoss: 0.3129 | ValAcc: 0.9990 ValLoss: 0.0119 | TestAcc: 0.9980 TestLoss: 0.0124
[GoogLeNet] Epoch 2 - TrainAcc: 0.9969 TrainLoss: 0.0207 | ValAcc: 0.9993 ValLoss: 0.0050 | TestAcc: 0.9983 TestLoss: 0.0059
[GoogLeNet] Epoch 3 - TrainAcc: 0.9984 TrainLoss: 0.0106 | ValAcc: 0.9997 ValLoss: 0.0043 | TestAcc: 0.9987 TestLoss: 0.0056
[GoogLeNet] Epoch 4 - TrainAcc: 0.9985 TrainLoss: 0.0084 | ValAcc: 0.9997 ValLoss: 0.0028 | TestAcc: 0.9987 TestLoss: 0.0046
[GoogLeNet] Epoch 5 - TrainAcc: 0.9985 TrainLoss: 0.0065 | ValAcc: 0.9997 ValLoss: 0.0024 | TestAcc: 0.9993 TestLoss: 0.0019
[GoogLeNet] Epoch 6 - TrainAcc: 0.9993 TrainLoss: 0.0045 | ValAcc: 0.9997 ValLoss: 0.0025 | TestAcc: 0.9997 TestLoss: 0.0014
[GoogLeNet] Epoch 7 - TrainAcc: 0.9989 TrainLoss: 0.0059 | ValAcc: 0.9993 ValLoss: 0.0034 | TestAcc: 0.9997 TestLoss: 0.0031
[GoogLeNet] Epoch 8 - TrainAcc: 0.9993 TrainLoss: 0.0039 | ValAcc: 0.9997 ValLoss: 0.0024 | TestAcc: 0.9997 TestLoss: 0.0011
[GoogLeNet] Epoch 9 - TrainAcc: 0.9999 TrainLoss: 0.0013 | ValAcc: 0.9997 ValLoss: 0.0022 | TestAcc: 0.9997 TestLoss: 0.0024
[GoogLeNet] Epoch 10 - TrainAcc: 0.9999 TrainLoss: 0.0010 | ValAcc: 0.9990 ValLoss: 0.0032 | TestAcc: 0.9993 TestLoss: 0.0029
✅ GoogLeNet 학습 완료: Best Epoch = 9, 전체 학습 시간: 0:07:53
[{'model': 'ResNet18', 'train_acc': 0.9995333333333334, 'val_acc': 0.9996666666666667, 'val_f1': 0.9996666645833202, 'train_time': '0:06
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   model       4 non-null      object
 1   train_acc   4 non-null      float64
 2   val_acc     4 non-null      float64
 3   val_f1      4 non-null      float64
 4   train_time  4 non-null      object
dtypes: float64(3), object(2)
memory usage: 292.0+ bytes
           model  train_acc   val_acc    val_f1 train_time
2  EfficientNetB0   0.998667  1.000000  1.000000    0:08:31
0        ResNet18   0.999533  0.999667  0.999667    0:06:15
1     MobileNetV2   0.997800  0.999000  0.998999    0:07:27
3       GoogLeNet   0.999933  0.999000  0.999000    0:07:53
```

- 학습 하는 epoch당 Acc, Loss 값.

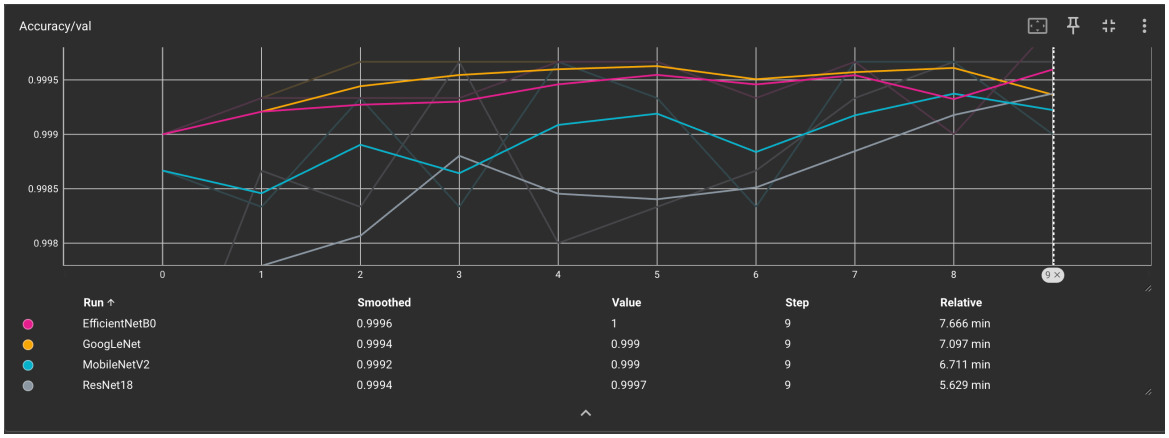


- GoogleNet Confusion Matrix, 모델 별 Heapmap





- TensorBoard Train Acc



- TensorBoard Val Acc

- SQLite를 이용한 DB연결.

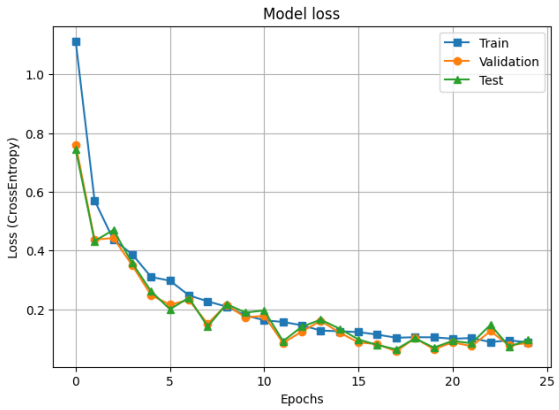
```
SQLite DB에 성공적으로 연결되었습니다: training_logs.db
데이터베이스 내 테이블 목록:
 - epoch_logs
 - sqlite_sequence
 - final_results

테이블 'epoch_logs'의 스키마:
(0, 'id', 'INTEGER', 0, None, 1)
(1, 'model_name', 'TEXT', 0, None, 0)
(2, 'epoch', 'INTEGER', 0, None, 0)
(3, 'train_acc', 'REAL', 0, None, 0)
(4, 'train_loss', 'REAL', 0, None, 0)
(5, 'val_acc', 'REAL', 0, None, 0)
(6, 'val_loss', 'REAL', 0, None, 0)
(7, 'test_acc', 'REAL', 0, None, 0)
(8, 'test_loss', 'REAL', 0, None, 0)
(9, 'timestamp', 'TEXT', 0, None, 0)

테이블 'sqlite_sequence'의 스키마:
(0, 'name', '', 0, None, 0)
(1, 'seq', '', 0, None, 0)

테이블 'final_results'의 스키마:
(0, 'id', 'INTEGER', 0, None, 1)
(1, 'model_name', 'TEXT', 0, None, 0)
(2, 'best_epoch', 'INTEGER', 0, None, 0)
(3, 'train_acc', 'REAL', 0, None, 0)
(4, 'train_loss', 'REAL', 0, None, 0)
(5, 'val_acc', 'REAL', 0, None, 0)
(6, 'val_loss', 'REAL', 0, None, 0)
(7, 'test_acc', 'REAL', 0, None, 0)
(8, 'test_loss', 'REAL', 0, None, 0)
(9, 'timestamp', 'TEXT', 0, None, 0)

조회할 테이블 이름을 입력하세요: sqlite_sequence

테이블 'sqlite_sequence'의 데이터:
('epoch_logs', 40)
('final_results', 4)
```

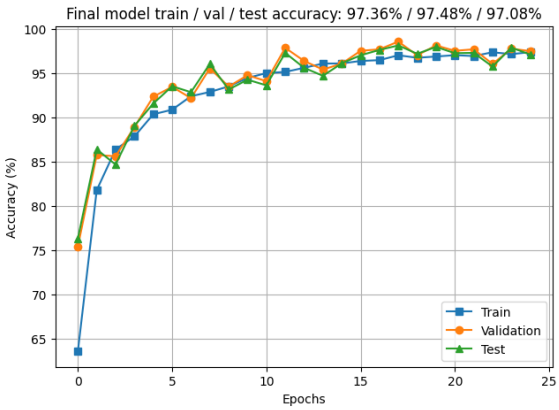
	id	model_name	epoch	start_time	end_time	loss	accuracy	f1_score
0	1	ResNet18	1	2025-03-25T05:30:31.974598	2025-03-25T05:30:31.974600	0.1234	0.9876	0.9821
1	2	ResNet18	2	2025-03-25T05:30:31.984205	2025-03-25T05:30:31.984207	0.0987	0.9901	0.9844

- DB DataFrame으로 확인하기.

- Deep CNN의 구조.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 128, 128]           1,792
       BatchNorm2d-2           [-1, 64, 64, 64]             128
            Conv2d-3          [-1, 128, 64, 64]          73,856
       BatchNorm2d-4          [-1, 128, 32, 32]             256
            Conv2d-5          [-1, 256, 32, 32]         295,168
       BatchNorm2d-6          [-1, 256, 16, 16]             512
            Conv2d-7          [-1, 512, 16, 16]       1,180,160
       BatchNorm2d-8            [-1, 512, 8, 8]           1,024
            Conv2d-9            [-1, 512, 8, 8]       2,359,808
      BatchNorm2d-10            [-1, 512, 4, 4]           1,024
           Conv2d-11            [-1, 512, 2, 2]       2,359,808
      BatchNorm2d-12            [-1, 512, 1, 1]           1,024
           Linear-13                 [-1, 1024]         525,312
           Linear-14                   [-1, 15]          15,375
================================================================
Total params: 6,815,247
Trainable params: 6,815,247
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 19.09
Params size (MB): 26.00
Estimated Total Size (MB): 45.28
```





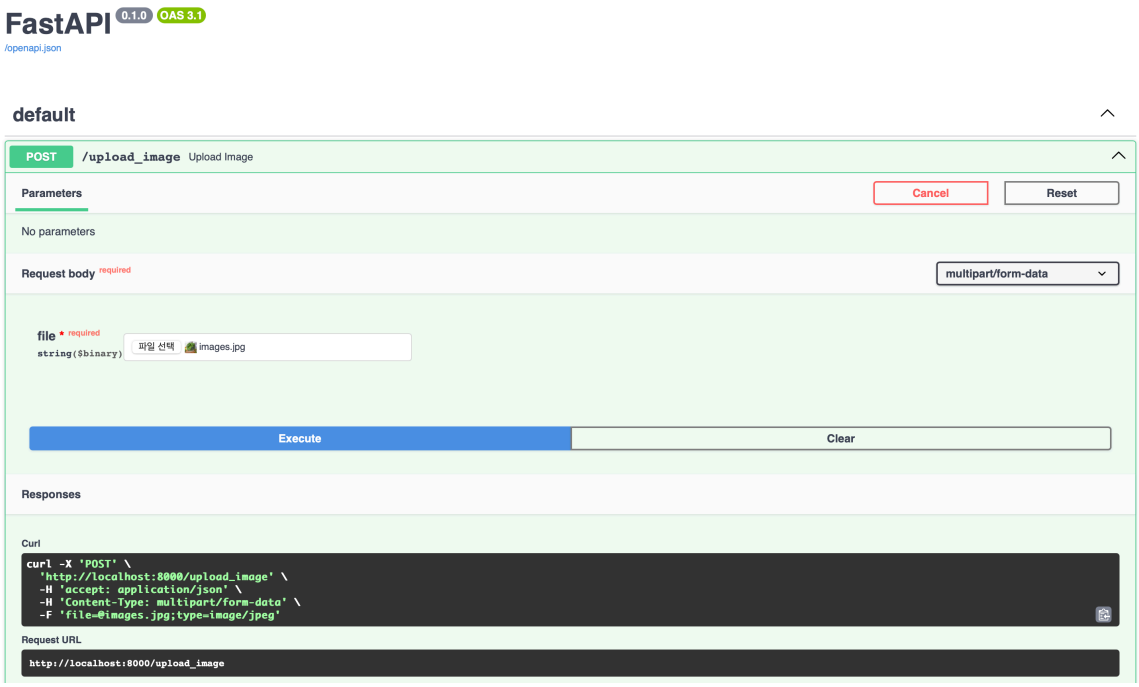
- Deep CNN Loss와 Train / Val/ Test accuracy

6. 결론

본 연구에서는 Pre-trained된 모델을 사용하여, CNN 기반 모델을 다뤄보는 경험을 하게 되었다. 사전 훈련된 CNN 모델을 도입함으로써 전이 학습의 장점을 효과적으로 활용하였다.

기본 모델에 비해 전이 학습 모델은 높은 분류 정확도와 빠른 수렴 속도를 보였으나, 일부 과적합 문제와 연산 비용 이슈가 존재하였다.

향후 Fastapi로 배포해보는 과정을 진행하면서, 백엔드 AI 엔지니어로서 필요한 역량을 채워나갈 예정이다.



이미지 업로드 부분까진 완료가 되었지만, 모델을 불러와서, 학습을 진행하는 부분은 완성하지 못하였다. 프로젝트가 완료되어도 이 부분을 완료시켜 git에 푸시할 예정이다.

7. 참고 문헌 및 부록

1. https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset/code
2. https://www.tensorflow.org/api_docs
3. https://www.tensorflow.org/tutorials?hl=ko