



Fruits and Vegetables Classification Optimized with Fine-Tuned ResNet50 and Transfer Learning

[Date : 2025.03]

- 0. Introduction
- 1. Dataset
 - 1.1. Kaggle 'Fruits and Vegetables Image Recognition'
 - 1.2. ImageNet
 - 1.3. Data Pre-processing
- 2. ResNet50 Model
 - 2.1. Model Info
 - 2.2. Fine-Tuning
 - 2.3. Optimization
 - 2.4. Training
- 3. Visualization
 - 3.1. Training Accuracy & Loss Graph
 - 3.2. Confusion Matrix
 - 3.3. Classification Report
- 4. Additional Data Recognition
- 5. Report
 - 5.1. Evaluation and Interpretation
 - 5.2. Conclusions and Improvement Measures
 - 5.3. Scalability and Business Model
- 6. Connected DB
 - 6.1. File Load
 - 6.2. Connected SQLite
 - 6.3. Create Table
 - 6.4. Insert Data

- [6.5. Commit](#)
- [6.6. Check Table Info](#)
- [7. Using DB Browser for SQLite](#)
- [8. Google Colab](#)

0. Introduction

인공지능의 발달로 방대한 데이터가 하루에도 **2.5엑사바이트**씩 생성되는 데이터 시대에 오면서 데이터 활용에 대한 사람들의 관심도 커지고 있다. 그중의 하나가 이미지 분류 및 인식 분야이며, 많은 기업에서도 이미지를 분석한 다양한 서비스를 제공하고 있다.

본 프로젝트에서 사용한 데이터셋은 kaggle에서 제공하는 '**Fruits and Vegetables Image Recognition**'이며, **ResNet50 모델 기반의 다중 클래스 이미지 분류 모델을 구축하는 것을 목표로 한다.** 사전 학습된 CNN 모델을 기반으로 하여, Transfer Learning 기법을 적용하여 모델을 최적화하였다.

모델의 일반화 성능을 향상하기 위해 FC Layer를 현재 데이터셋에 맞게 재설계하고, 일부 Conv Layer에 대해서 Fine-Tuning을 수행하였다. 또한, 과적합 문제를 완화하기 위해 Dropout, Early Stopping, Learning Rate Scheduler 등의 다양한 Optimization 기법을 단계적으로 적용하였다.

본 보고서에는 데이터의 전처리 과정부터 모델 설계, 학습 과정, 성능 평가, 그리고 실험 결과 분석에 이르기까지 전체적인 프로젝트 과정을 기술하며, 다양한 사전 학습 모델 및 학습 전략에 따른 성능 비교도 함께 다룬다.

1. Dataset

1.1. Kaggle 'Fruits and Vegetables Image Recognition'

해당 프로젝트에서 주요 학습 및 평가 데이터로 사용하였다. 이 데이터셋은 **36개**의 과일 및 채소 클래스로 구성되어 있으며, 다양한 배경과 조도를 포함한 이미지로 이루어져 있다. 아래의 코드를 통해 데이터 구성을 살펴보자.

```
print(f"Train 데이터 개수: {len(train_dataset)}")      # 클래스 당 약 100장
print(f"Validation 데이터 개수: {len(valid_dataset)}") # 클래스 당 약 10장
print(f"Test 데이터 개수: {len(test_dataset)}")        # 클래스 당 약 10장
```

```

## [결과 확인]
## Train 데이터 개수: 3115
## Validation 데이터 개수: 351
## Test 데이터 개수: 359

# 클래스 라벨 확인
print(f'클래스 개수 : {len(test_dataset.classes)})')
print(f'클래스 목록 : {test_dataset.classes}')

## [결과 확인]
## 클래스 개수 : 36
## 클래스 목록 : ['apple', 'banana', 'beetroot', 'bell pepper', 'cabbage',
##                 'capsicum', 'carrot', 'cauliflower', 'chilli pepper', 'corn', 'cucumber',
##                 'eggplant', 'garlic', 'ginger', 'grapes', 'jalapeno', 'kiwi', 'lemon',
##                 'lettuce', 'mango', 'onion', 'orange', 'paprika', 'pear', 'peas',
##                 'pineapple', 'pomegranate', 'potato', 'radish', 'soy beans', 'spinach',
##                 'sweetcorn', 'sweetpotato', 'tomato', 'turnip', 'watermelon']

```

해당 데이터는 실생활 환경에 가까운 정물 이미지들이 다수 포함되어 있으며, 이는 모델의 일반화 성능을 평가하기에 적합하다.

이후 모델 학습에는 **ImageNet으로 사전 학습된 ResNet50을 사용**하였으며, 이는 전이 학습을 통해 소규모 데이터셋에서도 빠른 수렴과 높은 분류 정확도를 확보하는 데 도움을 주었다.

1.2. ImageNet

ImageNet은 사전 학습 데이터셋으로 약 1,400만 장의 이미지와 2만 개 이상의 객체 클래스로 구성된 대규모 이미지 분류 데이터셋이며 객체 인식 및 이미지 분류 분야에서 대표적인 벤치마크로 활용된다.

딥러닝 기반의 이미지 분류 모델 학습에 가장 널리 사용되며, ResNet, EfficientNet 등의 다양한 CNN 기반 모델들은 ImageNet을 기반으로 사전 학습된 가중치를 제공한다.

본 프로젝트에서는 ImageNet에서 사전 학습된 ResNet50 모델을 기반으로 전이 학습을 수행하였으며, 이는 적은 양의 훈련 데이터만으로도 빠르게 수렴하고 높은 성능을 낼 수 있도록 도와준다. 자세한 코드는 아래 전처리 과정에서 살펴보도록 하자.

1.3. Data Pre-processing

먼저 Colab에 Google Drive를 연결하여 데이터셋을 업로드해주었다.

```
# Dataset Path  
train_dir = '/content/drive/MyDrive/kaggle/Fruits_and_Vegetables_Image_Reco  
test_dir = '/content/drive/MyDrive/kaggle/Fruits_and_Vegetables_Image_Reco  
valid_dir = '/content/drive/MyDrive/kaggle/Fruits_and_Vegetables_Image_Reco
```

이후 사용할 ResNet50 모델에 맞게 이미지의 포맷을 변경해 주어야 한다. 그냥 원본 이미지 파일들을 가져다가 사용할 경우 **모델이 학습하면서 P모드의 이미지로 인한 경고 문구가 출력**되며 학습이 잘 이루어지지 않는다.

확인해 보니 **.png** 파일의 이미지 형식이 P모드로 저장된 것이 문제였으며, .png 파일의 확장자를 **.jpg로 변경**하고 이미지의 형식을 **RGB로 변경**해 주는 작업을 진행하였다.

```
# png 파일의 이미지 형식이 P모드로 저장되어있어서 경고 메시지가 출력됨  
# -> png 파일의 확장자를 jpg로 변경  
def png_to_jpg(root_dir):  
    for class_folder in os.listdir(root_dir): # 클래스 폴더 순회  
        class_path = os.path.join(root_dir, class_folder)  
  
        # 폴더가 아니면 스킵  
        if not os.path.isdir(class_path):  
            continue  
  
        for img_name in os.listdir(class_path):  
            img_path = os.path.join(class_path, img_name)  
  
            # 확장자가 .png인지 확인  
            if img_path.lower().endswith(".png"):  
                # 이미지 열기  
                try:  
                    with Image.open(img_path) as img:  
                        # RGB로 변환 후 JPG로 저장  
                        img = img.convert("RGB")  
  
                        # 새로운 JPG 경로
```

```

new_img_path = img_path.rsplit(".", 1)[0] + ".jpg"

img.save(new_img_path, "JPEG", quality=95) # 95% 품질 유지
print(f"변환 완료: {img_path} → {new_img_path}")

# 기존 PNG 삭제
os.remove(img_path)

except Exception as e:
    print(f"변환 실패: {img_path} - {e}")

# 데이터셋에 변환 적용
png_to_jpg(train_dir)
png_to_jpg(valid_dir)
png_to_jpg(test_dir)

print("PNG → JPG 변환 성공")

## [결과 확인]
## PNG → JPG 변환 성공

```

사전 학습된 **ResNet50** 모델은 224x224 픽셀 크기와 앞서 살펴본 **ImageNet** 기준의 정규화 값을 입력으로 사용하므로, 본 프로젝트에서도 동일한 방식으로 이미지 전처리를 진행하였다.

```

# ResNet50에서 사용할 이미지로 변환
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),

    # ImageNet과 동일한 전처리 방식 사용 (동일한 정규화)
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# Dataset Load (ImageNet 전처리 적용)
train_dataset = datasets.ImageFolder(root=train_dir, transform=transform)

```

```
valid_dataset = datasets.ImageFolder(root=valid_dir, transform=transform)
test_dataset = datasets.ImageFolder(root=test_dir, transform=transform)
```

DataLoader를 설정할 때 GPU의 가속을 최적화하기 위해 batch_size와 num_workers의 값을 시스템의 환경에 맞게 조정하는 작업을 진행하였다. 초기 설정으로 batch_size=32, num_workers=4를 사용했으나, Colab 환경에서의 처리 안정성과 성능 향상을 고려하여 batch_size=16, num_workers=0으로 변경하였다.

결과적으로 더 낮은 메모리 사용량과 불필요한 DataLoader 병렬 처리 오류를 방지함으로써 오히려 더 나은 학습 결과를 도출할 수 있었다.

```
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True,
                         num_workers=0, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False,
                        num_workers=0, pin_memory=True)
valid_loader = DataLoader(valid_dataset, batch_size=16, shuffle=False,
                          num_workers=0, pin_memory=True)
```

여기서 batch_size와 num_workers의 값을 낮추었는데 성능이 더 좋은 이유를 분석해 보면, 먼저 작은 batch_size는 매 반복마다 다양한 gradient 방향을 경험하게 하므로 더 좋은 일반화 성능을 낼 수 있다.

또한, num_workers는 병렬화 작업을 진행하는데 일부 데이터가 꼬이면서 학습에 영향을 줄 수 있으므로 0일 때 더 안정적으로 작동했을 가능성이 높다.

마지막으로 Colab 환경에서는 VRAM 메모리를 더 많이 쓰기 때문에 32보다 16의 메모리 부담이 덜 해서 오히려 안정적으로 모든 연산이 수행되었다고 추측해 볼 수 있겠다.

2. ResNet50 Model

2.1. Model Info

ResNet은 Microsoft Research에서 2015년에 발표된 심층 신경망 모델로, 딥러닝 네트워크의 깊이가 깊어질수록 발생하는 성능 저하 문제를 해결하기 위해 개발되었다. 특히, 일반적인 CNN 구조에서는 층이 깊어질수록 학습이 어려워지고 성능이 오히려 감소하는 문제가 발생하는데, **ResNet은 이러한 문제를 Residual Learning이라는 구조적 접근으로 해결하였다.**

Residual Learning은 신경망의 잔차를 다음 층으로 직접 연결하는 **skip connection** 방식을 도입하여, 정보의 흐름을 원활하게 하고 기울기 소실 문제를 완화함으로써 깊은 네트워크에서도 안정적인 학습을 가능하게 한다.

본 프로젝트에서는 **50개의 레이어로 구성된 ResNet50을 사용**하였다. ImageNet 데이터셋으로 사전 학습된 가중치를 기반으로 하여 Transfer Learning을 수행한 모델이다. 이때, **FC Layer와 Conv Layer 일부를 부분적으로 Fine-Tuning 하여 36개의 클래스에 맞춰 Optimization을 진행**하였다.

2.2. Fine-Tuning

필요한 라이브러리는 가져왔다고 가정하고 진행한다. 먼저 사전 학습된 ResNet50 모델을 불러오자.

```
# ResNet50
model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
```

사전 학습된 ResNet50 모델을 기반으로 Fine-Tuning을 진행하였다. 이미 ImageNet의 데이터셋으로 사전 학습되어 일반적인 시각적 특징을 잘 추출할 수 있으므로, 이를 Feature Extractor로 활용하면서 특정 레이어만 학습할 수 있도록 변경하였다.

먼저, 원래 1,000개의 클래스 분류용으로 설계된 FC Layer를 클래스에 맞추어 **36개로 커스터마이징**을 진행하였다. 기존의 출력 크기인 2048을 **512차원으로 줄이고, ReLU 활성화 함수와 Dropout(0.3)**을 거쳐 클래스를 분류하도록 구조를 재정의하였다.

```
model.fc = nn.Sequential(
    nn.Linear(2048, 512), # 기존 출력 크기는 2048
    nn.ReLU(),           # 활성화 함수
    nn.Dropout(0.3),     # 과적합 방지로 추가 : 0.5에서 0.3으로 변경
    nn.Linear(512, 36)   # 최종 출력 개수를 36개로 변경
)
```

이후 모델 전체 파라미터를 **requires_grad=True**로 설정하여 모든 레이어를 학습할 수 있도록 설정하였다. 처음에는 **FC Layer만 학습**할 수 있도록 설정했지만, 모델의 학습 성능이 좋지 않아서 **Conv Layer의 Layer4도 학습**할 수 있도록 변경해 주었다.

Layer4는 가장 중요한 고차원적 특징을 가지고 있으며, False로 변경하면서 추가적인 학습이 가능해졌다. 기존 Feature Extractor 방식보다 더 강력한 **Fine-Tuning**을 진행하며 가중치가 늘어나고 성능 향상이 가능하다.

```

# 모든 레이어를 학습 가능하도록 설정
for param in model.parameters():
    param.requires_grad = True

# FC Layer랑 Conv Layer의 Layer4만 학습 가능하도록 설정
for name, param in model.named_parameters():
    if not name.startswith("layer4") and not name.startswith("fc"):
        param.requires_grad = False

# 모델 GPU로 이동
model = model.to(device)

```

결과적으로, 사전 학습된 ResNet50의 구조를 기반으로 FC Layer를 커스터마이징하고, Layer4만 학습할 수 있도록 설정하여 효율적인 Fine-Tuning을 수행하였다. 이러한 접근은 전체 모델을 학습시키는 방식보다 **연산 비용을 절감**하면서도, 중요한 **고차원 특징을 효과적으로 반영**하여 모델 성능 향상에 기여할 수 있다.

2.3. Optimization

앞서 Fine-Tuning된 ResNet50 모델을 기반으로, 다양한 최적화 전략을 결합하여 성능을 극대화하였다. 특히 Loss Function, Optimizer, Scheduler, AMP, Early Stopping 등을 활용하였으며 아래서 하나씩 살펴보자.

2.3.1. Loss Function

다중 클래스 분류 문제이므로 **CrossEntropyLoss()**를 사용하였다. 또한 과적합을 방지하기 위해서 **label_smoothing** 기법을 적용하여 라벨에 부드러움을 더해주었으며, 모델이 보다 일반화된 예측을 하도록 유도하였다.

```

# 손실 함수 설정
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

```

2.3.2. Optimizer

처음에 FC Layer만 학습하도록 튜닝을 진행했을 때는 Adam을 사용해 주었으나, Conv Layer의 Layer4를 추가 학습시키면서 성능 향상을 위해서 SGD로 변경하였다.

하지만 빠르고 안정적인 학습을 위해서 결과적으로 **AdamW**로 선택하여 사용하였다.

Learning Rate(Ir)은 0.001부터 0.0005, 0.0001로 줄이면서 모델의 진동을 억제하고 더욱 안정적인 수렴이 가능하도록 변경해 주었다.

```
# 옵티마이저 설정  
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0001)
```

2.3.3. Scheduler

Scheduler는 모델의 성능이 정체되면 자동으로 lr를 줄여주는 방식으로 동작한다. 처음에 사용하던 스케줄러는 ReduceLROnPlateau()였으나, 보다 부드럽게 학습률을 조절하면서 성능 향상을 유도하기 위해서 **CosineAnnealingWarmRestarts()**로 변경하였다.

CosineAnnealingWarmRestarts()를 사용하여 일정 주기마다 학습률을 리셋하며 진동을 조절하고 새로운 최적점을 탐색할 수 있도록 설정하였다.

```
# 스케줄러 설정  
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T_0=5, T_mult=2)
```

- T_0 = 5 : 5 epoch마다 주기를 리셋
- T_mult = 2 : 다음 주기는 2배씩 들어남 (5, 10, 20, ...)

2.3.4. AMP, Automatic Mixed Precision

torch.amp 기능을 활용하여 자동 혼합 정밀도 학습을 적용하였다. 이미지 분류와 같이 연산량이 큰 작업에서 연산 속도를 향상시키고 GPU 메모리 사용량을 감소시키는 데 효과적으로 작용한다.

```
# AMP : Mixed Precision 활성화 : 자동 스케일링 적용  
scaler = torch.amp.GradScaler("cuda")
```

AMP 적용시에 **GradScaler**를 활용하여 다음과 같이 역전파를 수행하였다. 진행 방식은 아래와 같다.

- torch.amp.autocast("cuda") : 자동으로 float16과 float32 혼합 연산을 수행
- scaler.scale(loss).backward() : 손실값을 스케일링하여 언더플로우를 방지

- `scaler.step(optimizer)` : 스케일된 값을 기반으로 파라미터 업데이트
- `scaler.update()` : 다음 스텝을 위한 스케일링 자동 조절

```
# 아래 모델 학습에서 진행되는 코드로 밑에서 전체 코드 확인 가능
with torch.amp.autocast("cuda"):
    outputs = model(images)
    loss = criterion(outputs, labels)

    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

2.3.5. Early Stopping

Validation Loss가 3회 연속 개선되지 않을 경우 학습을 조기 종료하여 불필요한 학습과 과적합을 방지하도록 설정하였다.

```
# Early Stopping 기준 변수
best_valid_acc = 0      # 최고의 Validation 정확도를 저장
best_valid_loss = float("inf")
early_stopping_counter = 0
early_stopping_patience = 3 # 3번 연속 성능이 개선되지 않으면 멈춤

# 모델 학습 안에서 나타남 -> 밑에 학습에서 전체 코드 확인 가능
    # Early Stopping
    if avg_valid_loss < best_valid_loss:
        best_valid_loss = avg_valid_loss
        early_stopping_counter = 0 # 성능 개선되면 카운터 초기화
        best_valid_acc = valid_acc

    # 모델 저장
    torch.save(model.state_dict(), "best_model.pth")
else:
    early_stopping_counter += 1 # 성능 개선 안 되면 카운트 증가
    if early_stopping_counter >= early_stopping_patience:
        print(f"Early Stopping. Epoch {epoch+1}에서 학습 종료")
        break
```

학습 도중 가장 높은 Validation 성능을 기록한 모델만 **best_model.pth**로 저장하여 최적의 모델을 사용한 평가만을 진행하였다.

2.4. Training

모델 학습은 총 20 epoch 동안 진행되었으며, **Train-Validation Split** 방식을 기반으로 Train 데이터와 Valid 데이터를 분리하여 진행하였다. 학습에는 위에서 보았던 Fine-Tuning과 Optimization이 주요하게 적용되었다.

2.4.1. 학습 과정

각 epoch 마다 아래의 다음과 같은 과정을 반복하였다.

- 모델을 학습 모드로 설정 → train data에 대해 loss 및 accuracy 계산
- 모델을 평가 모드로 설정 → valid data에 대해 loss 및 accuracy 계산
- 성능 기록 리스트에 저장
- Early Stopping 조건 확인 후 Best Model 저장

```
# 성능 기록을 위한 리스트 초기화 진행
train_accuracies = []
val_accuracies = []
train_losses = []
val_losses = []

# Early Stopping 기준 변수
best_valid_acc = 0      # 최고의 Validation 정확도를 저장
best_valid_loss = float("inf")
early_stopping_counter = 0
early_stopping_patience = 3 # 3번 연속 성능이 개선되지 않으면 멈춤

epochs = 20
scaler = torch.amp.GradScaler("cuda")

for epoch in range(epochs):
    start_time = time.time()
    model.train()
    train_loss, correct_train, total_train = 0, 0, 0
```

```

for images, labels in train_loader:
    images, labels = images.to(device), labels.to(device)
    optimizer.zero_grad()

    # AMP 적용 (Mixed Precision) : 속도 + 메모리 효율 높임
    with torch.amp.autocast("cuda"):
        outputs = model(images)
        loss = criterion(outputs, labels)

    # 역전파 수행
    # loss 값을 내부적으로 스케일해서 backward() 호출
    # : 작은 gradient들이 underflow되지 않고 안정적으로 역전파 가능
    scaler.scale(loss).backward()

    # 스케일된 gradient를 기준으로 optimizer step을 수행
    # : gradient를 스케일된 값으로 클리핑하고,
    # 다시 스케일을 원래대로 복원한 후 파라미터 업데이트를 진행함
    scaler.step(optimizer)
    scaler.update()

    train_loss += loss.item()
    _, predicted = torch.max(outputs, 1)
    correct_train += (predicted == labels).sum().item()
    total_train += labels.size(0)

    # Train Loss 계산
    avg_train_loss = train_loss / len(train_loader)
    train_acc = 100 * correct_train / total_train

    # Validation 성능 평가
    model.eval() # 모델을 평가 모드로 전환
    correct, total, val_loss = 0, 0, 0

    with torch.no_grad():
        for images, labels in valid_loader: # Validation Set 사용
            images, labels = images.to(device), labels.to(device)

```

```

        with torch.amp.autocast("cuda"):
            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        avg_valid_loss = val_loss / len(valid_loader)
        valid_acc = 100 * correct / total # Validation 정확도 계산

        # CosineAnnealingWarmRestarts에서 사용하는 스케줄러 방식 적용
        # -> Cosine Annealing은 metric 없이 step만 호출
        scheduler.step(epoch + 1)

        # 성능 기록
        train_losses.append(avg_train_loss)
        val_losses.append(avg_valid_loss)
        train_accuracies.append(train_acc)
        val_accuracies.append(valid_acc)

        # Early Stopping
        if avg_valid_loss < best_valid_loss:
            best_valid_loss = avg_valid_loss
            early_stopping_counter = 0 # 성능 개선되면 카운터 초기화
            best_valid_acc = valid_acc

        # 모델 저장
        torch.save(model.state_dict(), "best_model.pth")
else:
    early_stopping_counter += 1 # 성능 개선 안 되면 카운트 증가
    if early_stopping_counter >= early_stopping_patience:
        print(f"Early Stopping. Epoch {epoch+1}에서 학습 종료")
        break

print(f"Epoch [{epoch+1}/{epochs}], "
      f"Train Loss: {avg_train_loss:.4f}, "

```

```

f"Valid Loss: {avg_valid_loss:.4f}, "
f"Train Accuracy: {train_acc:.2f}%, "
f"Valid Accuracy: {valid_acc:.2f}%, "
f"Time: {time.time() - start_time:.2f}s")

```

>>> 결과

```

Epoch [1/20], Train Loss: 2.1518, Valid Loss: 1.0588, Train Accuracy: 54.48%, Valid Accuracy: 88.03%, Time: 958.30s
Epoch [2/20], Train Loss: 1.1991, Valid Loss: 0.9729, Train Accuracy: 82.12%, Valid Accuracy: 91.45%, Time: 131.61s
Epoch [3/20], Train Loss: 1.0047, Valid Loss: 0.8908, Train Accuracy: 89.66%, Valid Accuracy: 96.01%, Time: 131.31s
Epoch [4/20], Train Loss: 0.8850, Valid Loss: 0.8579, Train Accuracy: 94.35%, Valid Accuracy: 95.44%, Time: 132.06s
Epoch [5/20], Train Loss: 0.8336, Valid Loss: 0.8460, Train Accuracy: 96.37%, Valid Accuracy: 96.01%, Time: 132.22s
Epoch [6/20], Train Loss: 0.9086, Valid Loss: 0.8789, Train Accuracy: 93.26%, Valid Accuracy: 95.44%, Time: 132.04s
Epoch [7/20], Train Loss: 0.8857, Valid Loss: 0.8352, Train Accuracy: 94.83%, Valid Accuracy: 96.87%, Time: 131.99s
Epoch [8/20], Train Loss: 0.8230, Valid Loss: 0.8328, Train Accuracy: 96.73%, Valid Accuracy: 96.87%, Time: 132.09s
Epoch [9/20], Train Loss: 0.7917, Valid Loss: 0.7938, Train Accuracy: 97.88%, Valid Accuracy: 97.15%, Time: 133.20s
Epoch [10/20], Train Loss: 0.7679, Valid Loss: 0.8028, Train Accuracy: 98.30%, Valid Accuracy: 96.58%, Time: 132.59s
Epoch [11/20], Train Loss: 0.7492, Valid Loss: 0.7790, Train Accuracy: 98.72%, Valid Accuracy: 97.44%, Time: 133.27s
Epoch [12/20], Train Loss: 0.7320, Valid Loss: 0.7720, Train Accuracy: 99.13%, Valid Accuracy: 97.15%, Time: 132.33s
Epoch [13/20], Train Loss: 0.7290, Valid Loss: 0.7611, Train Accuracy: 99.13%, Valid Accuracy: 97.15%, Time: 132.03s
Epoch [14/20], Train Loss: 0.7200, Valid Loss: 0.7650, Train Accuracy: 99.29%, Valid Accuracy: 97.72%, Time: 132.56s
Epoch [15/20], Train Loss: 0.7165, Valid Loss: 0.7581, Train Accuracy: 99.26%, Valid Accuracy: 97.72%, Time: 131.90s
Epoch [16/20], Train Loss: 0.7662, Valid Loss: 0.8095, Train Accuracy: 98.04%, Valid Accuracy: 96.87%, Time: 131.76s
Epoch [17/20], Train Loss: 0.8017, Valid Loss: 0.8398, Train Accuracy: 97.37%, Valid Accuracy: 95.73%, Time: 132.03s
Early Stopping. Epoch 18에서 학습 종료

```

Validation Loss가 3번 연속 개선되지 않아서 **18번째 epoch**에서 **Early Stopping**으로 종료되었다. 전반적으로 Train Accuracy와 Valid Accuracy 모두 상승하며 과적합 없이 안정적인 학습이 진행되었음을 알 수 있다.

best_model.pth의 **Train Accuracy가 99.29%, Valid Accuracy가 97.72%** 정도로 매우 높다. best_model.pth는 가장 성능이 높은 시점인 epoch14 모델로 저장되어 있어서 약간의 흔들림이 있긴 했지만, 과적합 되지 않고 적절하게 일반화되었다고 할 수 있다.

2.4.2. Test

학습이 완료된 후 저장된 best_model.pth를 불러와 Test 데이터셋에 대해 최종 성능을 평가하였다. 모델은 eval() 모드로 전환되어 Dropout 등의 비활성화를 적용하고, torch.no_grad()를 통해 메모리 사용을 줄이며 예측을 수행하였다.

예측된 결과와 실제 정답을 비교하여 Accuracy를 계산하였으며, 이는 모델이 학습에 사용되지 않은 완전한 검증용 데이터에서 어느 정도 일반화 능력을 가지는지 판단하는 기준이 된다.

```

# 저장된 모델 불러오기
model.load_state_dict(torch.load("best_model.pth"))

# 모델 GPU로 이동
model.to(device)

```

```

# 모델 평가
model.eval()

# 예측 결과 저장
all_preds = []
all_labels = []

correct, total = 0, 0

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        # 리스트에 예측 결과 저장
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

print(f"테스트 정확도: {100 * correct / total:.2f}%")

```

>>> 결과

테스트 정확도: 97.49%

Image Classification에서 모델 학습의 정확도가 95~97% 이면 상위권 정확도로 분류하며 Valid와 Test의 정확도가 비슷한 것이 중요하다.

Test Accuracy는 97.49%로 Valid Accuracy(97.72%)와 거의 동일하게 나타났으며, 정확도도 최상위권으로 분류가 된다. 이는 모델 훈련이 과적합 되지 않고 적절하게 일반화되었음을 의미한다.

3. Visualization

3.1. Training Accuracy & Loss Graph

Train 데이터와 valid 데이터를 학습시킨 모델의 결과를 시각화한 그래프를 통해서 확인해 보자. 아래 그래프는 epoch 별로 Train 및 Valid 데이터에 대한 Accuracy와 Loss의 변화를 시각화한 것이다. 최대 Accuracy를 살펴보기 위해서 Accuracy 그래프에 마커 표시를 넣어주었다.

```
# best 모델이 저장된 epoch index (Validation Accuracy 기준)
best_epoch = val_accuracies.index(max(val_accuracies))
best_val_acc = val_accuracies[best_epoch]

# 가로 창 띄우기
plt.figure(figsize=(12, 6))

# Accuracy Graph
plt.subplot(1, 2, 1)
plt.plot(train_accuracies, label='Train Accuracy', color='green')
plt.plot(val_accuracies, label='Validation Accuracy', color='deeppink')

# 최고 성능 epoch의 점을 plot으로 표시 (marker 사용)
plt.plot(best_epoch, best_val_acc, marker='o', color='blue')

# 텍스트 추가
plt.text(best_epoch, best_val_acc - 2, f'{best_val_acc:.2f}%', color='blue', ha='center', fontsize=10)

plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.grid(True)

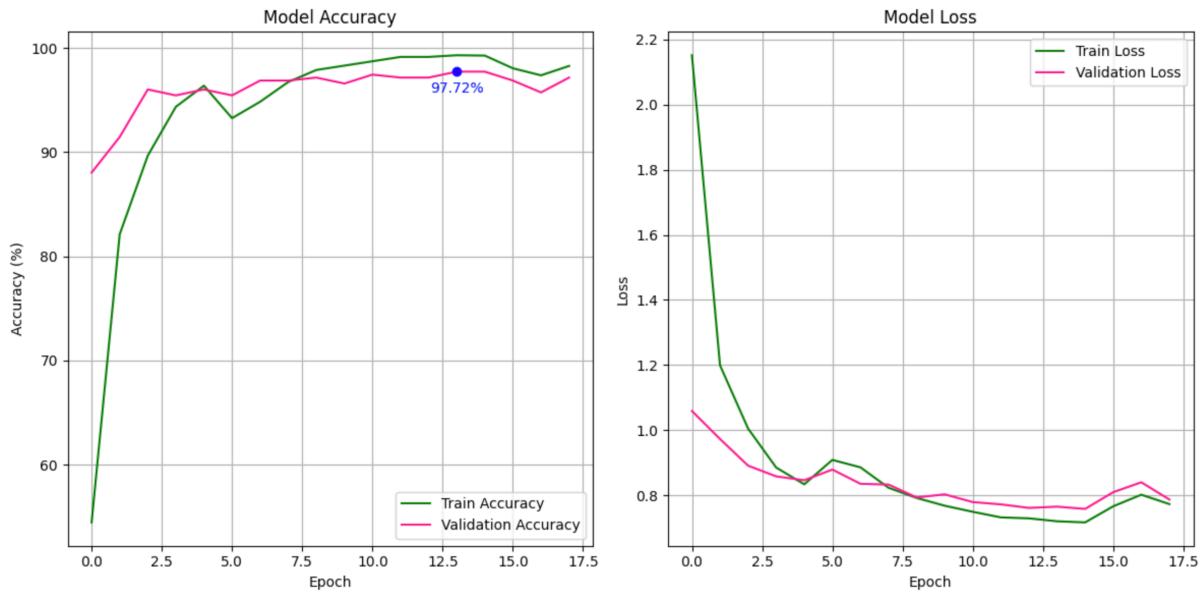
# Loss Graph
plt.subplot(1, 2, 2)
plt.plot(train_losses, label='Train Loss', color='green')
plt.plot(val_losses, label='Validation Loss', color='deeppink')
plt.title('Model Loss')
```

```

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

```



▼ 앞선 결과와 함께 보기 위해서 모델 학습 결과를 다시 가져와 분석을 진행해보자.

```

Epoch [1/20], Train Loss: 2.1518, Valid Loss: 1.0588, Train Accuracy: 54.48%, Valid Accuracy: 88.03%, Time: 958.30s
Epoch [2/20], Train Loss: 1.1991, Valid Loss: 0.9729, Train Accuracy: 82.12%, Valid Accuracy: 91.45%, Time: 131.61s
Epoch [3/20], Train Loss: 1.0047, Valid Loss: 0.8908, Train Accuracy: 89.66%, Valid Accuracy: 96.01%, Time: 131.31s
Epoch [4/20], Train Loss: 0.8850, Valid Loss: 0.8579, Train Accuracy: 94.35%, Valid Accuracy: 95.44%, Time: 132.06s
Epoch [5/20], Train Loss: 0.8336, Valid Loss: 0.8460, Train Accuracy: 96.37%, Valid Accuracy: 96.01%, Time: 132.22s
Epoch [6/20], Train Loss: 0.9086, Valid Loss: 0.8789, Train Accuracy: 93.26%, Valid Accuracy: 95.44%, Time: 132.04s
Epoch [7/20], Train Loss: 0.8857, Valid Loss: 0.8352, Train Accuracy: 94.83%, Valid Accuracy: 96.87%, Time: 131.99s
Epoch [8/20], Train Loss: 0.8230, Valid Loss: 0.8328, Train Accuracy: 96.73%, Valid Accuracy: 96.87%, Time: 132.09s
Epoch [9/20], Train Loss: 0.7917, Valid Loss: 0.7933, Train Accuracy: 97.88%, Valid Accuracy: 97.15%, Time: 133.20s
Epoch [10/20], Train Loss: 0.7679, Valid Loss: 0.8028, Train Accuracy: 98.30%, Valid Accuracy: 96.58%, Time: 132.59s
Epoch [11/20], Train Loss: 0.7492, Valid Loss: 0.7790, Train Accuracy: 98.72%, Valid Accuracy: 97.44%, Time: 133.27s
Epoch [12/20], Train Loss: 0.7320, Valid Loss: 0.7720, Train Accuracy: 99.13%, Valid Accuracy: 97.15%, Time: 132.33s
Epoch [13/20], Train Loss: 0.7290, Valid Loss: 0.7611, Train Accuracy: 99.13%, Valid Accuracy: 97.15%, Time: 132.03s
Epoch [14/20], Train Loss: 0.7200, Valid Loss: 0.7650, Train Accuracy: 99.29%, Valid Accuracy: 97.72%, Time: 132.56s
Epoch [15/20], Train Loss: 0.7165, Valid Loss: 0.7581, Train Accuracy: 99.26%, Valid Accuracy: 97.72%, Time: 131.90s
Epoch [16/20], Train Loss: 0.7662, Valid Loss: 0.8095, Train Accuracy: 98.04%, Valid Accuracy: 96.87%, Time: 131.76s
Epoch [17/20], Train Loss: 0.8017, Valid Loss: 0.8398, Train Accuracy: 97.37%, Valid Accuracy: 95.73%, Time: 132.03s
Early Stopping. Epoch 18에서 학습 종료

```

3.1.1. Model Accuracy Graph (좌측)

- Train Accuracy : 초반엔 낮다가 빠르게 상승하여 99% 이상까지 도달
- Validation Accuracy : 초기부터 높았고, 점진적으로 상승하여 97% 이상까지 도달

- 중간중간에 약간의 요동은 있지만, 전반적으로 train과 valid 간에 차이가 크지 않고 유사한 추세를 보임
 - 중간에 Validation이 소폭 하락하는 구간은 있었지만 빠르게 회복된 상태
 - validation이 더 높게 나오는 구간도 있어서 과적합 없이 일관된 학습 상태로 보임

3.1.2. Model Loss Graph (우측)

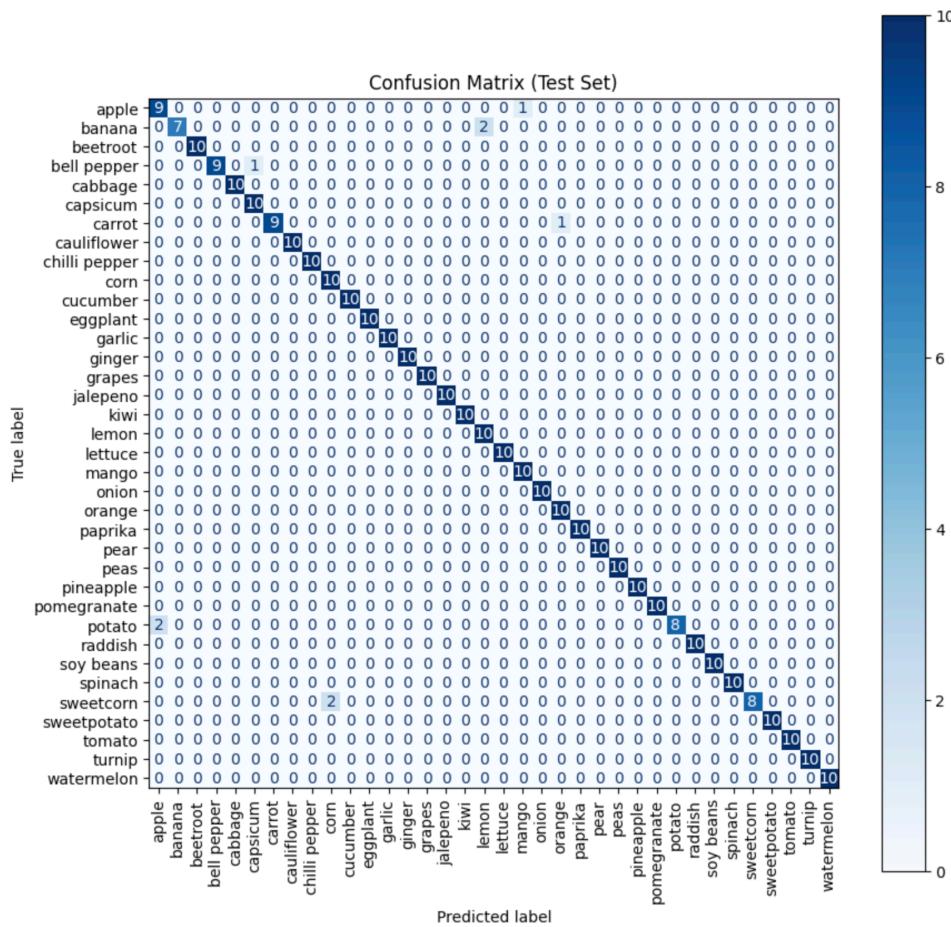
- Train Loss : 안정적으로 감소하고 있고, 마지막에 0.7 근처까지 떨어짐
- Validation Loss : 전체적으로 낮고, 중간에 살짝 요동친 부분도 금방 회복됨
 - Loss가 과도하게 분리되지 않고, 오히려 valid loss가 더 낮은 구간도 있음 → 일반화가 잘 되었다는 증거
 - 중간에 train loss는 줄었는데 valid loss가 조금 튀었다가 다시 떨어지는 건 흔한 현상 → 회복되며 큰 이탈 없이 수렴
 - Train Loss와 Validation Loss가 과도하게 벌어지지 않고 거의 평행한 패턴을 유지, 이는 모델이 과적합 없이 잘 학습되었고, 일반화에도 성공했음을 시사

3.2. Confusion Matrix

모델의 성능을 보다 정밀하게 분석하기 위해 Test Dataset에 대해 Confusion Matrix를 시각화하였다. Confusion Matrix는 각 클래스별 예측 결과를 행렬 형태로 나타내어, 모델이 어떤 클래스를 잘 분류했는지, 어떤 클래스에서 오류가 발생했는지를 직관적으로 파악할 수 있게 한다. 아래에서 코드를 통해 결과를 살펴보자.

```
# Test Dataset Confusion Matrix 시각화
cm = confusion_matrix(all_labels, all_preds)
class_names = test_loader.dataset.classes

# 시각화
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                               display_labels=class_names)
fig, ax = plt.subplots(figsize=(10, 10))
disp.plot(ax=ax, xticks_rotation=90, cmap='Blues')
plt.title("Confusion Matrix (Test Set)")
plt.show()
```



대부분의 클래스에서 정확하게 예측이 이루어졌으며 다수의 클래스는 100%의 정확도를 나타냈다. 오분류가 발생한 경우는 소수에 불과하며 apple, potato, sweetcorn 등이 있다.

이러한 오분류는 주로 시각적으로 유사한 채소 또는 과일 간에 발생하거나 여러 채소 종류가 한 사진에 모두 들어가 있는 경우, 캐릭터나 로고 사진의 경우에 발생한다. 이는 이미지 분류 모델에서 자주 나타나는 일반적인 경향이다.

결과적으로, Confusion Matrix는 전체적으로 모델이 클래스 전반에 걸쳐 안정적이고 높은 분류 성능을 유지하고 있음을 시각적으로 확인할 수 있게 해준다.

3.3. Classification Report

다중 클래스 분류 문제에서 **Precision**(정밀도), **Recall**(재현율), **F1-score**, **Support**(지원수) 등의 성능 지표를 클래스별로 상세히 제공하는 평가 도구이다. 이는 단순한 정확도만으로는 알 수 없는 각 클래스별 분류의 성능을 분석하는데 유용하다.

```
# classification report 출력  
print(classification_report(all_labels, all_preds, target_names=class_names))
```

- **Precision**, 정밀도 : 예측한 Positive 중에서 실제 Positive인 비율
→ 이 클래스라고 예측한 것 중에서 맞는 비율
- **Recall**, 재현율 : 실제 Positive 중에서 맞게 예측한 비율
→ 이 클래스를 맞춘 비율
- **F1-score** : Precision과 Recall의 조화 평균
→ Precision과 Recall의 균형을 고려한 점수
- **Support** : 각 클래스의 실제 샘플 수

	precision	recall	f1-score	support
apple	0.82	0.90	0.86	10
banana	1.00	0.78	0.88	9
beetroot	1.00	1.00	1.00	10
bell pepper	1.00	0.90	0.95	10
cabbage	1.00	1.00	1.00	10
capsicum	0.91	1.00	0.95	10
carrot	1.00	0.90	0.95	10
cauliflower	1.00	1.00	1.00	10
chilli pepper	1.00	1.00	1.00	10
corn	0.83	1.00	0.91	10
cucumber	1.00	1.00	1.00	10
eggplant	1.00	1.00	1.00	10
garlic	1.00	1.00	1.00	10
ginger	1.00	1.00	1.00	10
grapes	1.00	1.00	1.00	10
jalepeno	1.00	1.00	1.00	10
kiwi	1.00	1.00	1.00	10
lemon	0.83	1.00	0.91	10
lettuce	1.00	1.00	1.00	10
mango	0.91	1.00	0.95	10
onion	1.00	1.00	1.00	10
orange	0.91	1.00	0.95	10
paprika	1.00	1.00	1.00	10
pear	1.00	1.00	1.00	10
peas	1.00	1.00	1.00	10
pineapple	1.00	1.00	1.00	10
pomegranate	1.00	1.00	1.00	10
potato	1.00	0.80	0.89	10
raddish	1.00	1.00	1.00	10
soy beans	1.00	1.00	1.00	10
spinach	1.00	1.00	1.00	10
sweetcorn	1.00	0.80	0.89	10
sweetpotato	1.00	1.00	1.00	10
tomato	1.00	1.00	1.00	10
turnip	1.00	1.00	1.00	10
watermelon	1.00	1.00	1.00	10
accuracy			0.97	359
macro avg	0.98	0.97	0.97	359
weighted avg	0.98	0.97	0.97	359

전체 accuracy는 97%로, 모델이 전반적으로 높은 예측 성능을 달성하였음을 보여준다. 대부분의 클래스에서 Precision, Recall, F1-score가 모두 1.00에 일치하거나 근접한다. 또한, macro avg, weighted avg가 각각 0.97~0.98로 클래스 간 불균형 없이 전반적으로 고른 성능을 나타낸다.

이러한 결과는 모델이 대다수 클래스에 대해 안정적인 분류 성능을 달성하고 있으며, 특히 과일 및 채소 간 시각적으로 유사한 항목들에서도 높은 수준의 분별력을 보여준다는 점에서 매우 긍정적으로 평가가 가능하다.

4. Additional Data Recognition

모델의 분류 성능을 직관적으로 평가하기 위해서 추가로 임의의 실제 데이터 5개를 입력하여 분류 결과 예측을 진행하였다. 예측에 사용된 이미지는 sweetcorn, mango, cabbage, cucumber, capsicum으로 구성되며, 다른 채소들과 비슷하여 분류하기 헷갈리는 이미지도 일부 들어가 있다.

```
import torch.nn.functional as F

# 이미지 경로 리스트
img_paths = [
    "/content/drive/MyDrive/kaggle/sweetcorn.png",
    "/content/drive/MyDrive/kaggle/mango.png",
    "/content/drive/MyDrive/kaggle/cabbage.png",
    "/content/drive/MyDrive/kaggle/cucumber.png",
    "/content/drive/MyDrive/kaggle/capsicum.png"
]

# 모델 예측 및 시각화
model.eval()

class_names = test_loader.dataset.classes

fig, axes = plt.subplots(1, 5, figsize=(15, 4))

for i, img_path in enumerate(img_paths):
    true_label = os.path.basename(img_path).split('.')[0].lower()
```

```

# 이미지 불러오기 및 resize : 시각화용 이미지도 224x224로 맞춤
img = Image.open(img_path).convert("RGB").resize((224, 224))
input_tensor = transform(img).unsqueeze(0).to(device)

with torch.no_grad():
    output = model(input_tensor)
    probs = F.softmax(output, dim=1)
    pred_idx = probs.argmax(dim=1).item()

predicted_label = class_names[pred_idx]
title_color = 'deeppink' if true_label == predicted_label else 'royalblue'

# 이미지 출력
axes[i].imshow(img)
axes[i].axis('off')
axes[i].set_title(f"True: {true_label}\nPred: {predicted_label}",
                  fontsize=10, color=title_color)

plt.tight_layout()
plt.show()

```



모든 예시 이미지에서 모델은 실제 클래스와 동일한 클래스를 정확히 예측하였으며, 특히 sweetcorn, cabbage, capsicum처럼 유사한 색상과 질감을 가진 클래스도 정확하게 분류하였다. 이는 모델이 단순히 색상이나 모양에만 의존하는 것이 아니라, 학습된 고차원적인 특성을 바탕으로 객체를 정확하게 인식하고 있다는 것을 보여준다.

5. Report

5.1. Evaluation and Interpretation

본 모델은 ResNet50을 기반으로 Fine-Tuning을 적용하여 Fruits and Vegetables 이미지를 클래스별로 분류하는 문제를 해결하였다. 학습 과정에서 AMP, Scheduler 등의 Optimization 기법을 사용하여 학습 효율과 안정성을 확보하였다.

학습 결과, Train Accuracy는 99.29%까지 상승하였고, **Validation Accuracy는 최고 97.72%**를 기록하였다. Train과 Validation 간의 성능 차이가 크지 않았으며, Validation Loss 역시 안정적인 하향 추세를 유지하였다. 이는 과적합 없이 모델이 잘 일반화되었음을 의미한다.

Test Accuracy도 97.49%로 Validation Accuracy와 비슷하게 나왔다. 추가로 Confusion Matrix와 Classification Report를 분석한 결과, 대부분의 클래스에서 높은 Precision과 Recall을 나타냈으며 전체 Accuracy도 97% 이상, F1-score 또한 매우 우수하게 나타났다. 이는 복잡한 클래스 간 구분도 잘 수행하고 있음을 시사한다.

또한, 임의 이미지 데이터 5장에 대한 예측에서도 비슷한 외형의 클래스 간의 구분 능력이 높게 나타났으며, 모델이 다양한 실제 이미지에 대해서도 잘 동작함을 확인할 수 있다.

5.2. Conclusions and Improvement Measures

이번 프로젝트를 통해 ResNet50 기반의 전이 학습과 Fine-Tuning 전략을 활용해 고성능 이미지 분류 모델을 구축하였다. 과적합 없이 높은 정확도를 달성하였으며, 실제 이미지에 대한 예측 결과 또한 일관성 있게 잘 수행되었다.

향후 개선 방향은 아래와 같이 생각해 볼 수 있다.

- 데이터 확장
 - 현재의 데이터셋은 train : valid : test = 10 : 1 : 1인데, test 이미지의 경우 클래스당 10장의 test 이미지로 제한되어 있다.
 - 더 다양한 배경, 조도, 각도의 이미지를 포함하면 일반화 성능의 향상이 가능할 것 같다.
- 모델 경량화 진행
 - 실시간 환경을 고려하여 최적화를 진행할 수 있다.
 - Knowledge Distillation, Quantization 등의 기법을 도입해 볼 수 있다.

5.3. Scalability and Business Model

본 모델은 **식품 공장, 대형 유통 센터, 스마트 팜** 등에서 **채소와 과일을 자동으로 분류하는** 시스템에 활용될 수 있다. 컨베이너벨트를 따라 움직이는 물체를 CCTV나 고정형 카메라로 실시간 촬영을 진행한 다음에 분류를 진행한다. 이후 자동 패킹이나 선별 로봇 등으로 연계하는 시스템을 구축해 볼 수 있을 듯하다. 이러한 확장으로 다음과 같은 효과를 기대해 볼 수 있다.

- **작업 자동화** : 반복적인 수작업을 줄여 전체적인 작업 속도 향상
- **인건비 절감** : 단순 분류 작업의 자동화로 인력 효율성 증대
- **품질 일관성** : 사람보다 일관된 기준으로 분류하여 품질 안정성 확보 가능

6. Connected DB

모델 학습의 결과를 테이블로 관리하면 편리할 것이므로 DB와 연결하는 추가적인 학습을 진행하였다. Google Colab 환경에서 진행하므로 코드를 작성하기 용이한 **SQLite**를 사용하였다. SQLite에서 직접 MySQL로 마이그레이션 하는 방식을 택했다.

6.1. File Load

앞서서 모델 학습이 끝난 뒤에 성능을 기록한 `train_accuracies`, `val_accuracies`, `train_losses`, `val_losses` 리스트를 `performance_data.pkl` 파일로 구글 드라이브에 저장해두었다. (구글 드라이브 마운트는 완료되었다고 가정한다.)

```
# pickle 파일 로드
with open("/content/drive/MyDrive/kaggle/performance_data.pkl", 'rb') as f:
    performance_data = pickle.load(f)
```

6.2. Connected SQLite

Colab 환경에서 `sqlite3`을 이용하여 바로 DB 연결을 수행하였다. 이는 Google Drive에 `.db` 파일로 저장된다. `model_metrics.db` 파일로 저장하였으며, 파일이 존재하면 연결 후 덮어쓸 수 있는 상태로 연결되고, 존재하지 않을 경우 자동으로 생성된다.

연결 후 `.cursor()`를 통해서 SQL 실행에 사용할 커서를 생성한다.

```
# SQLite DB 연결 (없으면 자동 생성됨)
conn = sqlite3.connect('/content/drive/MyDrive/kaggle/model_metrics.db')
cursor = conn.cursor()
```

6.3. Create Table

Table은 MySQL과 유사한 SQL 문법을 따르며, 테이블 명은 performance로 생성해주었다. PK는 epoch으로 지정하였으며, 나머지는 값들은 모두 실수형 REAL 데이터 타입으로 선언하였다.

```
# 테이블 생성
cursor.execute("""
    CREATE TABLE IF NOT EXISTS performance (
        epoch INTEGER PRIMARY KEY,
        train_accuracy REAL,
        val_accuracy REAL,
        train_loss REAL,
        val_loss REAL
    )
""")
```

6.4. Insert Data

생성된 performance 테이블에 performance_data.pkl 파일의 값들을 넣는 작업을 수행하였다.

```
# 데이터 삽입
for epoch in range(len(performance_data['train_accuracies'])):
    cursor.execute("""
        INSERT INTO performance (epoch, train_accuracy, val_accuracy, train_loss)
        VALUES (?, ?, ?, ?, ?)
    """, (
        epoch + 1,
        performance_data['train_accuracies'][epoch],
        performance_data['val_accuracies'][epoch],
```

```
    performance_data['train_losses'][epoch],  
    performance_data['val_losses'][epoch]  
))
```

위에서 **(?, ?, ?, ?, ?)**는 ‘파라미터 자리 표시자’이며, 뒤의 튜플의 값을 순서대로 넣는다. 이를 통해서 SQL Injection을 방지하며 효율적인 데이터 삽입이 가능하다.

6.5. Commit

모든 변경 사항을 저장하고 DB 연결을 종료하여 성능 데이터를 안전하게 저장한다.

```
# 저장 및 종료  
conn.commit()  
conn.close()
```

6.6. Check Table Info

Colab 내에서 SQL 쿼리를 사용하여 Table의 목록과 구조 정보 등을 확인할 수 있다. 아래를 통해서 살펴보자.

```
# 이미 생성된 DB에 다시 연결한 후 아래 작업을 수행  
conn = sqlite3.connect('/content/drive/MyDrive/kaggle/model_metrics.db')  
cursor = conn.cursor()  
  
# 테이블 목록 확인  
cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")  
tables = cursor.fetchall()  
  
print("생성된 테이블 목록:")  
for table in tables:  
    print(" -", table[0])  
  
# performance 테이블 구조 확인  
cursor.execute("PRAGMA table_info(performance);")  
columns = cursor.fetchall()  
  
print("\n테이블 구조:")
```

```

for col in columns:
    print(f" - Column: {col[1]} | Type: {col[2]}")

```

▼ 결과 확인

생성된 테이블 목록:

- performance

테이블 구조:

- Column: epoch | Type: INTEGER
- Column: train_accuracy | Type: REAL
- Column: val_accuracy | Type: REAL
- Column: train_loss | Type: REAL
- Column: val_loss | Type: REAL

이어서 저장된 데이터도 SQL 코드를 통해 확인해보자.

```

cursor.execute("SELECT * FROM performance;")
rows = cursor.fetchall()

print("저장된 데이터:")
for row in rows:
    print(row)

```

▼ 결과 확인

저장된 데이터:

```

(1, 54.47833065810594, 88.03418803418803, 2.151753524633554, 1.0587957067923113)
(2, 82.11878009630819, 91.45299145299145, 1.1991165185586, 0.9728676351633939)
(3, 89.66292134831461, 96.01139601139602, 1.0047374178201725, 0.8908318172801625)
(4, 94.34991974317818, 95.44159544159544, 0.8850136078321017, 0.8579314351081848)
(5, 96.37239165329053, 96.01139601139602, 0.8336357361231095, 0.8459914164109663)
(6, 93.25842696629213, 95.44159544159544, 0.9085827858020098, 0.8788693709806963)
(7, 94.8314606741573, 96.86609686609687, 0.8857054037925525, 0.8351851620457389)
(8, 96.72552166934189, 96.86609686609687, 0.8229895741511614, 0.8327939429066398)
(9, 97.88121990369181, 97.15099715099716, 0.7917162488668393, 0.7932811704548922)
(10, 98.29855537720707, 96.58119658119658, 0.7678831635377346, 0.8027781817046079)
(11, 98.71589085072232, 97.43589743589743, 0.7492248550439492, 0.779024289412932)
(12, 99.13322632423755, 97.15099715099716, 0.7319671957920759, 0.7720382294871591)
(13, 99.13322632423755, 97.15099715099716, 0.7290009911243732, 0.7611207501454786)
(14, 99.29373996789727, 97.72079772079772, 0.7200257402199965, 0.7649849382313815)
(15, 99.26163723916532, 97.72079772079772, 0.7165406107902527, 0.7581016258759932)
(16, 98.04173354735153, 96.86609686609687, 0.766174696958982, 0.8094540644775737)
(17, 97.36757624398074, 95.72649572649573, 0.8016980213996692, 0.8398389166051691)
(18, 98.26645264847512, 97.15099715099716, 0.7728611001601586, 0.7868229421702299)

```

스키마에서 알 수 있듯이 앞에서부터 epoch, train_accuracy, valid_accuracy, train_loss, valid_loss 순서이다. Pandas를 이용해서 깔끔한 테이블 형태로 데이터 출력도 가능하다. 아래에서 살펴보자.

```

import pandas as pd

df = pd.read_sql("SELECT * FROM performance",
                 sqlite3.connect('/content/drive/MyDrive/kaggle/model_me
df

```

▼ 결과 확인

epoch		train_accuracy	val_accuracy	train_loss	val_loss
0	1	54.478331	88.034188	2.151754	1.058796
1	2	82.118780	91.452991	1.199117	0.972868
2	3	89.662921	96.011396	1.004737	0.890832
3	4	94.349920	95.441595	0.885014	0.857931
4	5	96.372392	96.011396	0.833636	0.845991
5	6	93.258427	95.441595	0.908583	0.878869
6	7	94.831461	96.866097	0.885705	0.835185
7	8	96.725522	96.866097	0.822990	0.832794
8	9	97.881220	97.150997	0.791716	0.793281
9	10	98.298555	96.581197	0.767883	0.802778
10	11	98.715891	97.435897	0.749225	0.779024
11	12	99.133226	97.150997	0.731967	0.772038
12	13	99.133226	97.150997	0.729001	0.761121
13	14	99.293740	97.720798	0.720026	0.764985
14	15	99.261637	97.720798	0.716541	0.758102
15	16	98.041734	96.866097	0.766175	0.809454
16	17	97.367576	95.726496	0.801698	0.839839
17	18	98.266453	97.150997	0.772861	0.786823

7. Using DB Browser for SQLite

'DB Browser for SQLite'를 로컬에 설치한 후에 진행한다. Google Drive에 저장된 model_metrics.db 파일을 'DB Browser for SQLite'에 올리면 Database Structure를 살펴보거나 SQL을 사용하는 것이 가능하다. 아래에서 이미지를 통해 간단하게 살펴보자.

The screenshot shows the DB Browser for SQLite interface. The top menu bar includes 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Undo', 'Open Project', 'Save Project', and 'Attach Database'. Below the menu is a toolbar with icons for 'Create Table', 'Create Index', 'Print', and 'Refresh'. The main window displays the database schema under 'Database Structure'. The 'Tables (1)' section is expanded, showing a single table named 'performance'. The table's schema is defined by the following SQL code:

```
CREATE TABLE performance ( epoch INTEGER PRIMARY KEY, train_accuracy REAL, val_accuracy REAL, train_loss REAL, val_loss REAL )
    "epoch" INTEGER
    "train_accuracy" REAL
    "val_accuracy" REAL
    "train_loss" REAL
    "val_loss" REAL
```

Below the table definition are sections for 'Indices (0)', 'Views (0)', and 'Triggers (0)'.

먼저 `performance` 테이블의 모든 데이터를 출력하는 코드를 작성하고 결과를 확인해보자.

The screenshot shows the DB Browser for SQLite interface with the 'Execute SQL' tab selected. A SQL query is entered in the text area:

```
1 | SELECT *
2 | FROM performance;
```

The results are displayed in a table format below the query:

	epoch	train_accuracy	val_accuracy	train_loss	val_loss
1	84.4783306581069	88.054188034188	2.16176352463555	1.06879570679251	
2	82.1187800963082	91.4629914629915	1.1991165186586	0.972867635163394	
3	89.6629213483146	96.011396011396	1.00473741782017	0.890831817280162	
4	94.3499197431782	95.4415954415954	0.888013607832102	0.887931435108188	
5	96.5723916632905	96.011396011396	0.833636736123109	0.845991416410966	
6	93.2584269662921	98.4415954415954	0.90868278580201	0.878869370980696	
7	94.8314606741573	98.8660968660969	0.886705403792552	0.835186162045739	
8	96.7255216693419	98.8660968660969	0.822989574181161	0.83279394290664	
9	97.5812199036918	97.1809971509972	0.791716248866839	0.793281170454892	
10	98.2985583772071	98.5811965811966	0.767883163537735	0.802778181704608	
11	98.7158908807223	97.4358974358974	0.749224855043949	0.779024289412932	
12	99.1332263242376	97.1609971509972	0.731967195792076	0.772038229487159	

Execution finished without errors.
Result: 18 rows returned in 14ms
At line 1:
SELECT *
FROM performance;

아래로 내리면 18줄이 모두 잘 나온 것을 볼 수 있다. 이어서 2.4.절에서 저장되었던 `best_model.pth`과 동일한 모델은 무엇인지 찾아보자. `val_accuracy`가 가장 높은 모델로 찾아주면 된다. 쿼리와 결과는 아래와 같다.

S... S...

```
1 SELECT *
2 FROM performance
3 ORDER BY val_accuracy DESC
4 LIMIT 1;
5
```

epoch	train_accuracy	val_accuracy	train_loss	val_loss
14	09.2937399678973	97.7207977207977	0.720025740219997	0.764984938231381

Execution finished without errors.
Result: 1 rows returned in 14ms
At line 1:
SELECT *
FROM performance
ORDER BY val_accuracy DESC

epoch14에서 best_model인 것을 알 수 있고, 2.4.절에서 살펴본 것과 동일하다. 이와 같은 방식으로 DB를 연결하여 사용할 수 있음을 확인했다.

8. Google Colab

