CNN과 사전학습 모델 비교

날짜	@2025년 3월 18일 → 2025년 3월 28일
태그	

이름	날짜	담당자	상태
CNN 이진 분류 모델 생성	@2025년 3월 18일		시작 전
Attention 적용법 학습	@2025년 3월 19일 → 2025년 3월 19일		시작 전
T-net 학습	@2025년 3월 20일		시작 전
CNN 딥다이브	@2025년 3월 20일 → 2025년 3월 21일		시작 전
Skip Connection	@2025년 3월 21일		시작 전
BAM / CBAM	@2025년 3월 22일		시작 전
Learning Late	@2025년 3월 24일		시작 전
Pretrained + Attention	@2025년 3월 24일 → 2025년 3월 25일		시작 전
DataBase	@2025년 3월 26일		시작 전

데이터셋

Fruits and Vegetables Image Recognition Dataset

Fruit and Vegetable Images for Object Recognition

https://www.kaggle.com/datasets/kritikseth/fruit-and-vegetable-image-recognition

▼ 데이터 다운로드 방법

```
# 구글 코랩에서 데이터셋 다운로드 및 준비
from google.colab import files
files.upload()  # 'kaggle.json' 파일 업로드

!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

# Orange Diseases 데이터셋 다운로드
!kaggle datasets download -d kritikseth/fruit-and-vegetable-image-recognition

# 압축 해제
!unzip fruit-and-vegetable-image-recognition.zip -d fruit_vegetable
```

설명

야채와 과일이 혼합된 36개의 클래스로 구성된 데이터

항목	개수(per class)
Train	100
Test	10
Validation	10

개요

주어진 야채 및 과일 이미지들이 있는 데이터셋을 바탕으로 분류모델을 생성하는것이 목표임

→ 하지만 데이터를 살펴보니 높은 해상도와 퀄리티를 지니므로 사전학습 모델을 사용하지 않고도 높은 성능을 도출할 수 있을 것으로 보임

따라서 베이스 CNN 모델부터 시작하여 추가적인 기법들을 도입하여 각 모델을 직접 생성하여 평가하고 사전학습 모델을 적용한 후 각 모델들을 비교하여 평가를 진행해보고 어느정도의 차이를 나타내는지 분석하고자 함

목표

채소 / 야채의 이진 분류 및 클래스 세부 분류 진행

1차 적용 구상

기본적인 모델을 사용하지만 이미지 처리와 관련하여 추가적으로 적용하면 효과가 있을것으로 보이는 기법들을 조사하여 적용해보고자 함

모델은 기본적인 CNN을 Base로 하여 사전학습 모델과의 성능 차이를 비교 (+ attention 적용)

- CNN (Base)

  이미지 분류 진행 모델

- Attention (지역 특징 추출)

  이미지 데이터의 컨텍스트 벡터를 추출하여 사용하기 위해 적용해보고자 함

  → 분류에 중요한 특징을 추출하기 위한 기법

- 사전 학습 모델

  이미지 분류의 성능을 높이기 위한 방법(기존 학습 모델의 구조를 그대로 사용하여 성능 향상)

모델 구상

분류를 수행하기 위한 여러가지의 기법을 적용하여 모델을 구상 (성능 기준)

1. **only CNN**

   가장 기본적인 모델을 통해 어느정도의 성능이 나오는지 확인

2. **CNN + attention (지역적 특성 강화)**

   attention을 적용하였을 때 지역적 특성을 파악하여 성능 개선이 가능할 것으로 판단

   우려사항 : 이미지 별 중요 특징이 다를 것으로 보이는데 어텐션 메커니즘을 단순 적용하였을 때 재대로된 성능 개선이 일어날 것인가?

3. **Pre-Model + Fine Tune**

   사전 학습 모델을 사용하여 분류 진행

   → 레이어가 깊고 성능이 좋기에 어느정도 성능이 보장될 것으로 보임

   우려사항 : 이미지를 분류하기에는 과한 모델을 사용하여 자원 낭비일수도 있을 것으로 보임

4. **Pre-Model(feature extraction) + attention**

   사전학습 모델을 특징 추출기로 사용하여 어텐션을 적용

   → 위 2번 모델에서 이미지 자체에 attention을 적용한다면 이미지에서의 특징을 뽑겠지만 추출된 feature로부터 attention을 적용한다면 분류 자체에 맞는 성능개선이 가능할 것으로 보임

   우려사항 : 특징 추출 및 attention까지 적용하면 연산량이 많고 모델이 무거워짐

1. CNN 모델 생성 (이진 분류)

다운받은 데이터를 활용하여 이진분류 모델 생성 (ImageDataGenerator 활용)

▼ 데이터 구성 변경

ImageDataGenerator를 사용하여 이진분류에 맞게 데이터를 로딩하기 위해 카테고리에 맞는 폴더를 구상 후 데이터 위치 이동

```
# 새로운 폴더 경로 (fruit/와 vegetable/을 만들 예정)
train_fruit_dir = os.path.join(base_dir, "fruit")
train_vegetable_dir = os.path.join(base_dir, "vegetable")

# 필요한 폴더 생성
os.makedirs(train_fruit_dir, exist_ok=True)
os.makedirs(train_vegetable_dir, exist_ok=True)

# 새로운 폴더 경로 (fruit/와 vegetable/을 만들 예정)
test_fruit_dir = os.path.join(test_dir, "fruit")
test_vegetable_dir = os.path.join(test_dir, "vegetable")

# 필요한 폴더 생성
os.makedirs(test_fruit_dir, exist_ok=True)
os.makedirs(test_vegetable_dir, exist_ok=True)

# 이동할 클래스 리스트 (소문자로 변환)
fruit_classes = [
    "banana", "apple", "pear", "grapes", "orange", "kiwi", "watermelon", "pomegranate",
    "pineapple", "mango"
]

vegetable_classes = [
    "cucumber", "carrot", "capsicum", "onion", "potato", "lemon", "tomato", "raddish",
    "beetroot", "cabbage", "lettuce", "spinach", "soy beans", "cauliflower", "bell pepper",
    "chilli pepper", "turnip", "corn", "sweetcorn", "sweetpotato", "paprika", "jalepeno",
    "ginger", "garlic", "peas", "eggplant"
]
```

```
# 폴더 이동 함수
def move_classes_to_new_folders(dir, fruit_dir, vegetable_dir):
    for class_name in os.listdir(dir):
        class_path = os.path.join(dir, class_name)

        # 폴더가 아닌 경우 (파일 등) 무시
        if not os.path.isdir(class_path):
            continue

        class_name_lower = class_name.lower()  # 소문자로 변환 후 비교

        # 과일 클래스라면 fruit 폴더로 이동
        if class_name_lower in fruit_classes:
            new_path = os.path.join(fruit_dir, class_name)
            shutil.move(class_path, new_path)
            print(f"✅ {class_name} → fruit/ 로 이동 완료")

        # 채소 클래스라면 vegetable 폴더로 이동
        elif class_name_lower in vegetable_classes:
            new_path = os.path.join(vegetable_dir, class_name)
            shutil.move(class_path, new_path)
            print(f"✅ {class_name} → vegetable/ 로 이동 완료")

        else:
            print(f"⚠️ {class_name} → 분류되지 않음 (확인 필요)")

# 실행
move_classes_to_new_folders(base_dir, train_fruit_dir, train_vegetable_dir)
move_classes_to_new_folders(test_dir, test_fruit_dir, test_vegetable_dir)
```

▼ 데이터 로드

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# 데이터 증강 및 전처리
datagen = ImageDataGenerator(
    rescale=1.0/255,  # 정규화
    validation_split=0.2  # 80% 학습, 20% 검증 데이터
)

# 학습 데이터 불러오기
train_generator = datagen.flow_from_directory(
    base_dir,
    target_size=(256, 256),  # 이미지 크기 조정
    batch_size=32,
    class_mode="binary",  # 이진 분류 (fruit=0, vegetable=1)
```

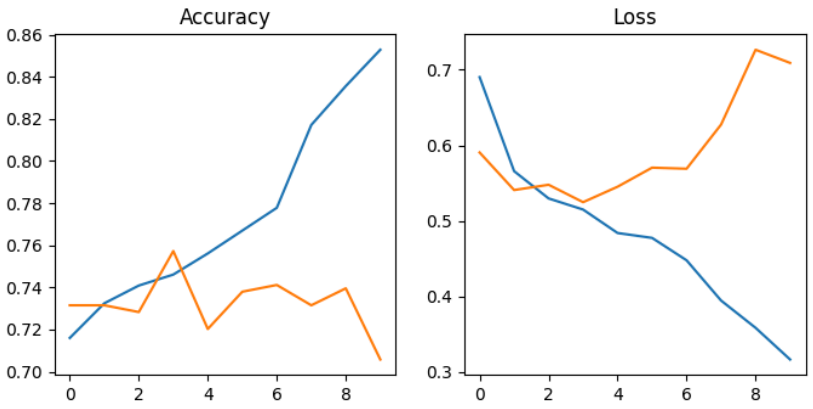
```
    subset="training"
)

# 검증 데이터 불러오기
val_generator = datagen.flow_from_directory(
    base_dir,
    target_size=(256, 256),
    batch_size=32,
    class_mode="binary",
    subset="validation"
)
```

▼ 모델 생성

```
import tensorflow as tf
from tensorflow.keras import layers, models

# CNN 모델 정의
def fruit_vegetable_cnn():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),

        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),  # 과적합 방지
        layers.Dense(1, activation='sigmoid')  # 이진 분류 (0: Fruit, 1: Vegetable)
    ])

    return model

# 모델 생성
model = fruit_vegetable_cnn()

# 모델 컴파일
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 모델 구조 확인
model.summary()
```

▼ 학습

```
epochs = 10
# 모델 학습
history = model.fit(
    train_generator,
    epochs,
    validation_data=val_generator
)
epochs_range = range(10)

plt.figure(figsize=(8,8))
plt.subplot(2,2,1)
plt.plot(epochs_range, history.history['accuracy'],label='Training Accuracy')
plt.plot(epochs_range , history.history['val_accuracy'],label ='Validation Accuracy')
plt.title('Accuracy')

plt.subplot(2,2,2)
plt.plot(epochs_range , history.history['loss'], label='Training Loss')
plt.plot(epochs_range, history.history['val_loss'], label='Validation Loss')
plt.title('Loss')

plt.show()
```



→ 그래프를 통해 과적합이 일어나는 것을 알 수 있음

▼ 결과(Test)

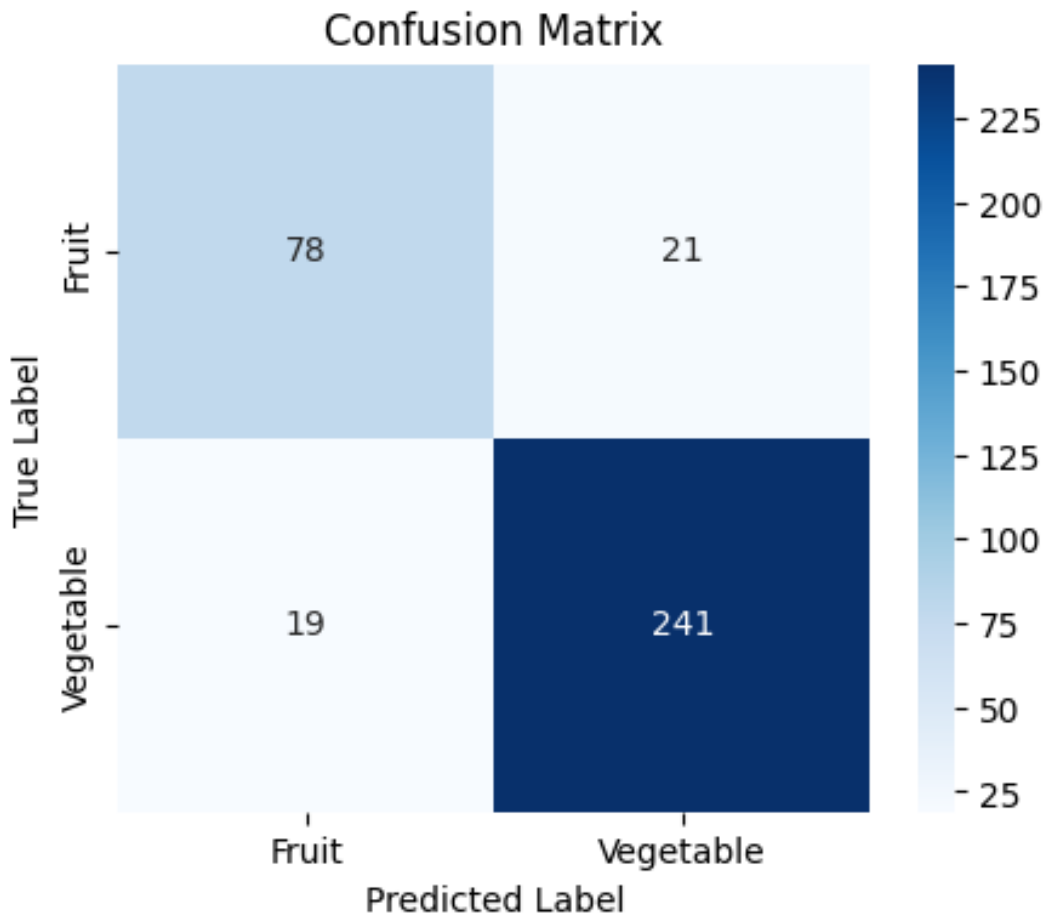
```
# 테스트 데이터 불러오기
test_datagen = ImageDataGenerator(rescale=1.0/255)

test_generator = test_datagen.flow_from_directory(
    "fruit_vegetable/test",  # 테스트 데이터 경로
    target_size=(256, 256),
    batch_size=32,
    class_mode="binary",
```

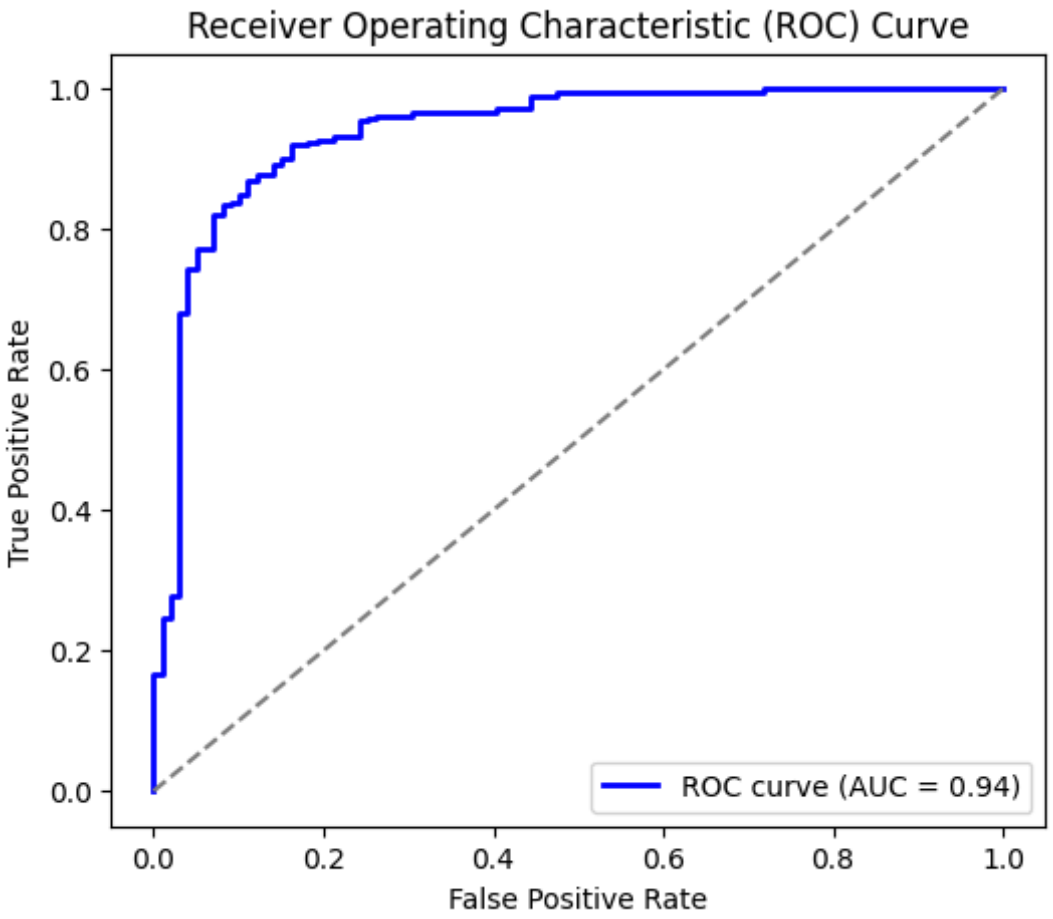
```
    shuffle=False  # 순서를 유지해서 예측 결과 비교 가능
)

# 테스트 데이터에서 평가
test_loss, test_acc = model.evaluate(test_generator)

print(f"테스트 데이터 정확도: {test_acc * 100:.2f}%")
print(f"테스트 데이터 손실 값: {test_loss:.4f}")
```

```
테스트 데이터 정확도: 88.86%
테스트 데이터 손실 값: 0.2973
```

```
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc
import numpy as np
import matplotlib.pyplot as plt

# 예측값 가져오기
y_true = test_generator.classes  # 실제 라벨
y_pred_prob = model.predict(test_generator)  # 예측 확률값
y_pred = (y_pred_prob > 0.5).astype(int).flatten()  # 0.5 기준으로 이진 분류

# 혼동 행렬 생성
cm = confusion_matrix(y_true, y_pred)

# 시각화
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Fruit', 'Vegetable'], yticklabels=['Fruit', 'Vegetable'])

# 그래프 설정
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# ROC 곡선 계산
fpr, tpr, _ = roc_curve(y_true, y_pred_prob)
roc_auc = auc(fpr, tpr)

# 시각화
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='grey', linestyle='--')  # 기준선
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
```

```
plt.legend(loc="lower right")
plt.show()
```





결론

단순 CNN을 사용하는 경우 이진분류에서 88%의 정확도를 보임

테스트 데이터에 대해서는 나쁘지 않은 성능으로 보일수도 있지만 validation 값이 빠르게 튀는 것을 보니 이미지를 통해 야채와 과일을 분류하는 것은 적합하지 않다고 판단.

→ 단순히 생각해도 야채와 과일은 보이는 것으로 판단되는 것이 아니기에 이미지에 그 특성이 나타나긴 어렵다고 판단되어 분류 기준을 수정하여 각 클래스 자체를 분류해보기 위한 모델을 생성

클래스 분류를 위한 데이터 로드

▼ 데이터 다운로드

```
# 1. Kaggle API 키 업로드 및 설정
from google.colab import files
files.upload()  # kaggle.json 업로드

!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

# 2. Google Drive 마운트 및 프로젝트 폴더 생성
from google.colab import drive
import os
```

```
drive.mount('/content/drive')

# 프로젝트 폴더 설정
project_dir = "" #"/content/drive/MyDrive/KTB_personal_project"
dataset_dir = os.path.join(project_dir, "fruit_vegetable")
os.makedirs(dataset_dir, exist_ok=True)

# 3. 데이터셋 다운로드 (압축 파일은 일단 /content 에 다운로드)
!kaggle datasets download -d kritikseth/fruit-and-vegetable-image-recognition -p /content

# 4. 압축 해제 후 프로젝트 폴더로 이동
!unzip -q /content/fruit-and-vegetable-image-recognition.zip -d "{dataset_dir}"

print(f"✅ 데이터셋이 다음 경로에 준비되었습니다: {dataset_dir}")
```

▼ ImageDataGenerator을 활용한 데이터 로딩

```
import os
import shutil
from google.colab import drive
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator


#drive.mount('/content/drive')

# 기존 데이터 경로
base_dir = "fruit_vegetable/train"
test_dir = "fruit_vegetable/test"
val_dir = "fruit_vegetable/validation"


# 데이터 증강 및 전처리
datagen = ImageDataGenerator(
    rescale=1.0/255,  # 정규화
)

# 학습 데이터 불러오기
train_generator = datagen.flow_from_directory(
    base_dir,
    target_size=(256, 256),  # 이미지 크기 조정
    batch_size=32,
    class_mode="categorical",  # 이진 분류 (fruit=0, vegetable=1)
)

# 검증 데이터 불러오기
```

```
val_generator = datagen.flow_from_directory(
    val_dir,
    target_size=(256, 256),
    batch_size=32,
    class_mode="categorical",
)

# 테스트 데이터 불러오기
test_datagen = ImageDataGenerator(rescale=1.0/255)

test_generator = test_datagen.flow_from_directory(
    test_dir,  # 테스트 데이터 경로
    target_size=(256, 256),
    batch_size=32,
    class_mode="categorical",
    shuffle=False  # 순서를 유지해서 예측 결과 비교 가능
)

print(train_generator.class_indices)  # 클래스별 인덱스 확인
print(train_generator.num_classes)  # 총 클래스 개수 확인
print(train_generator.batch_size)  # 배치 크기 확인

print(test_generator.class_indices)  # 클래스별 인덱스 확인
print(test_generator.num_classes)  # 총 클래스 개수 확인
print(test_generator.batch_size)  # 배치 크기 확인
```

▼ 데이터 시각화

```
# 클래스 인덱스를 이름으로 매핑하는 딕셔너리 생성
class_indices = train_generator.class_indices  # {'fruit': 0, 'vegetable': 1}
# 인덱스를 이름으로 바꾸기 위해 반전
idx_to_class = {v: k for k, v in class_indices.items()}

# 배치 데이터 가져오기
images, labels = next(train_generator)

# 이미지 시각화
fig, axes = plt.subplots(3, 3, figsize=(8, 8))
axes = axes.ravel()

for i in range(9):
    axes[i].imshow(images[i])
    # 원-핫 인코딩 벡터를 정수 인덱스로 변환
    class_index = np.argmax(labels[i])
    class_name = idx_to_class[class_index]
    axes[i].set_title(class_name)
    axes[i].axis('off')
```

```
plt.tight_layout()
plt.show()
print(images.shape)
```

참고

클라우드를 마운트하는 것은 데이터를 매번 다운받지 않고 사용할 수 있지만 코랩 환경에서 사용 시 마운트하여 사용하게 되면 학습 속도가 현저히 떨어짐

→ 로컬이 아닌 클라우드 환경의 경우엔 단순 다운받아 사용하는것이 좋음

모델 학습을 위한 함수(라이브러리) 설정

▼ 라이브러리

```
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

import tensorflow as tf
from tensorflow.keras import layers, models, backend
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard

from tensorflow.keras.applications import VGG16, ResNet50
```

▼ 학습 함수 선언

```
def train_model(model, model_name, optimizer, train_data, val_data, metrics=['accuracy'], lo
ss_fn='categorical_crossentropy', epochs=25, verbose=1):
    # 저장 경로 설정
    checkpoint_path = f'/content/drive/MyDrive/KTB_personal_project/{model_name}.keras'
    log_dir = f'/content/drive/MyDrive/KTB_personal_project/logs/{model_name}'

    # 콜백 설정
    checkpoint_cb = ModelCheckpoint(
        filepath=checkpoint_path,
        monitor='val_accuracy',
        save_best_only=True,
        mode='max',
        verbose=1
    )
    tensorboard_cb = TensorBoard(log_dir=log_dir, histogram_freq=1)
```

```
    # 모델 컴파일 & 학습
    model.compile(
        optimizer=optimizer,
        loss=loss_fn,
        metrics=metrics
    )

    history = model.fit(
        train_data,
        validation_data=val_data,
        epochs=epochs,
        verbose=verbose,
        callbacks=[checkpoint_cb, tensorboard_cb]
    )

    return model, history
```

▼ 시각화 함수 선언

```
def visualize_model_performance(history, model, test_generator):
    """
    모델 학습 결과와 테스트 평가 및 혼동 행렬을 시각화하는 함수

    Parameters:
        history: model.fit() 결과 객체
        model: 학습된 모델 객체
        test_generator: 테스트 데이터 제너레이터
    """
    # 1. Accuracy & Loss 시각화
    epochs_range = range(len(history.history['accuracy']))

    plt.figure(figsize=(12, 6))

    # Accuracy Plot
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, history.history['accuracy'], label='Training Accuracy')
    plt.plot(epochs_range, history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss Plot
    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, history.history['loss'], label='Training Loss')
    plt.plot(epochs_range, history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
```

```
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# 2. 테스트 데이터에서 평가
test_loss, test_acc = model.evaluate(test_generator, verbose=0)
print(f"\n✅ 테스트 데이터 정확도: {test_acc * 100:.2f}%")
print(f"✅ 테스트 데이터 손실 값: {test_loss:.4f}")

# 3. 혼동 행렬 시각화
y_true = test_generator.classes
y_pred_prob = model.predict(test_generator)
y_pred = np.argmax(y_pred_prob, axis=1)

cm = confusion_matrix(y_true, y_pred)
class_labels = list(test_generator.class_indices.keys())

plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_labels, yticklabels=class_labels)

plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```

2. CNN (클래스 분류)

위 과정을 통해 이진분류는 적합하지 않다고 판단.

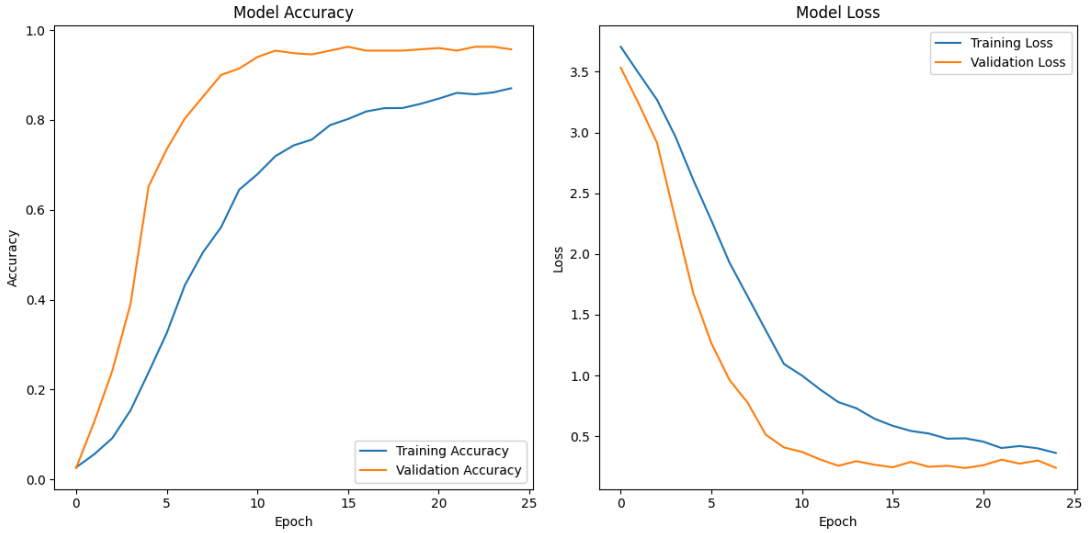
클래스 분류를 위한 CNN모델 생성

▼ 모델 생성

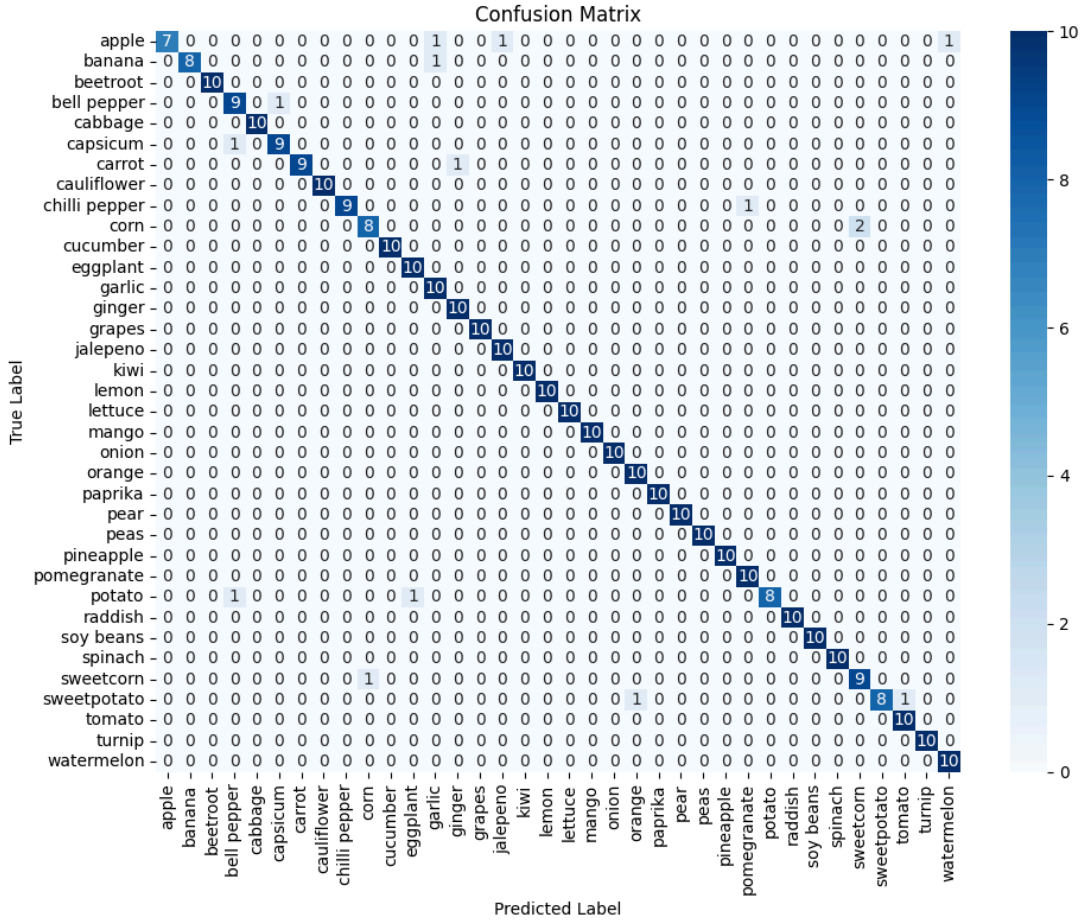
```
# CNN 모델 정의
def cnn():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)),
        layers.MaxPooling2D((2, 2)),

        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
```

```
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),

        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),  # 과적합 방지
        layers.Dense(36, activation='softmax')
    ])

    return model
```

▼ **결과**



✅ 테스트 데이터 정확도: 95.82%
✅ 테스트 데이터 손실 값: 0.2348



결론

처음 예상한 대로 일반적인 CNN만 사용하더라도 충분히 높은 성능을 보이는 것을 볼 수 있음

분류하지 못한 몇가지 데이터를 직접 확인해본 결과 corn과 sweetcorn같이 품종자체가 분류하기 쉽지 않았던 경우였음

→ 추가 기법을 적용하여 변화를 확인

CNN에 적용할 수 있는 attention 기법의 종류

정확히 말하면 기법의 종류라기보단 적용 위치에 따라 달라지는 방법을 말함

1. **Spatial Attention**: 공간적으로 중요한 부분을 강조 (예: CBAM, Self-Attention)
2. **Channel Attention**: 채널(특징 맵) 간 중요한 정보를 학습 (예: Squeeze-and-Excitation, CBAM)
3. **Self-Attention (Transformers 계열)**: 전체 이미지의 전역적인 관계를 학습 (예: ViT, CNN+Transformer Hybrid)

현재 프로젝트에선 이미지 간 관계가 아닌 이미지로부터 중요한 특성을 파악하여야 하기에 1, 2번이 적합하다.

또한 1번의 경우 이미지에서 학습 시 중요한 위치를 파악하기 위한 기법이지만 데이터세트의 이미지에선 대상의 위치가 이미지별로 다르므로 공간의 중요성을 파악하긴 어려움이 있음

→ 따라서 최종적으로 특징 추출 후 사용하는 channel attention을 적용하고자 함

Channel_Attention

channel attention 기법은 컨볼루션 레이어를 통해 추출된 feature vector에 attention을 적용하여 모델의 입력으로 사용

→ 기존에 존재하던 Squeeze-and-Excitation(SE) Block을 활용하여 Channel_Attention() 블록을 만들어 사용

Channel_Attention() 동작 과정

1. **채널 수 확인**

```
channel = input_feature.shape[-1]  # 입력 Feature Map의 채널 수 가져오기
```

- `input_feature.shape[-1]` → CNN Feature Map에서 **채널 개수(C)**를 가져옴.
- 이 값은 나중에 FC Layer(Dense)에서 사용됨.

---

2. **Global Average Pooling (GAP)**

```
avg_pool = layers.GlobalAveragePooling2D()(input_feature)  # (Batch, C)
avg_pool = layers.Reshape((1, 1, channel))(avg_pool)  # (Batch, 1, 1, C)
```

- *GAP(Global Average Pooling)**을 수행하여 각 채널의 평균값을 계산.
- 출력 형태는 `(Batch, Channels)`이므로 `(1, 1, C)` 형태로 Reshape.

의미: CNN의 Feature Map에서 각 채널이 얼마나 중요한지 요약하는 과정.

---

3. **MLP (Fully Connected Layers)**

```
dense_1 = layers.Dense(channel // ratio, activation='relu')(avg_pool)  # 채널 수 축소
dense_2 = layers.Dense(channel, activation='sigmoid')(dense_1)  # 원래 채널 크기로 복원
```

- **첫 번째 Dense Layer (`dense_1`)**
  - 채널 수를 `C / ratio` 만큼 줄임 (기본 `ratio=8`)
  - `activation='relu'` 적용
  - 예: 채널이 128개라면 `128 → 16`으로 축소
- **두 번째 Dense Layer (`dense_2`)**
  - 원래 채널 수 `C`로 복원
  - `activation='sigmoid'` 적용하여 0~1의 가중치 생성

의미: MLP를 사용해 **각 채널의 중요도를 학습**하는 과정.

---

4. **원래 Feature Map과 곱하기**

```
return layers.Multiply()([input_feature, dense_2])
```

- 각 채널별 가중치를 원래 Feature Map과 곱함.
- 중요도가 높은 채널은 크게 반영되고, 중요도가 낮은 채널은 억제됨.

Channel Attention 전체 코드

```
# Channel Attention (SE Block) 정의
def channel_attention(input_feature, ratio=8):
    channel = input_feature.shape[-1]  # 채널 수 가져오기

    # Global Average Pooling (채널별 평균값 계산)
    avg_pool = layers.GlobalAveragePooling2D()(input_feature)
    avg_pool = layers.Reshape((1, 1, channel))(avg_pool)

    # 두 개의 Fully Connected Layer (MLP)
    dense_1 = layers.Dense(channel // ratio, activation='relu')(avg_pool)  # 축소
    dense_2 = layers.Dense(channel, activation='sigmoid')(dense_1)  # 복원 (Sigmoid로 가중치 계
산)

    # 원래 Feature Map에 가중치 적용
    return layers.Multiply()([input_feature, dense_2])
```

Channel Attention의 전체 흐름

1. CNN의 Feature Map에서 각 채널의 평균값(GAP) 계산
2. MLP(Dense Layers)를 사용하여 각 채널의 중요도를 학습
3. Sigmoid 활성화를 통해 0~1의 가중치 생성
4. 원래 Feature Map과 곱해서 중요한 채널만 강조

Channel Attention의 효과

불필요한 채널 억제 & 중요한 채널 강조 → 성능 향상
특징을 자동으로 학습하여 이미지 분류 성능 향상
ResNet, EfficientNet 등 최신 CNN 모델에서도 사용되는 기법

3. CNN + attention

기본 CNN 모델과 위에서 생성한 Channel_attention()을 사용하여 모델 학습 진행

데이터 로드 및 시각화 코드는 동일하므로 모델 구조와 결과만 작성

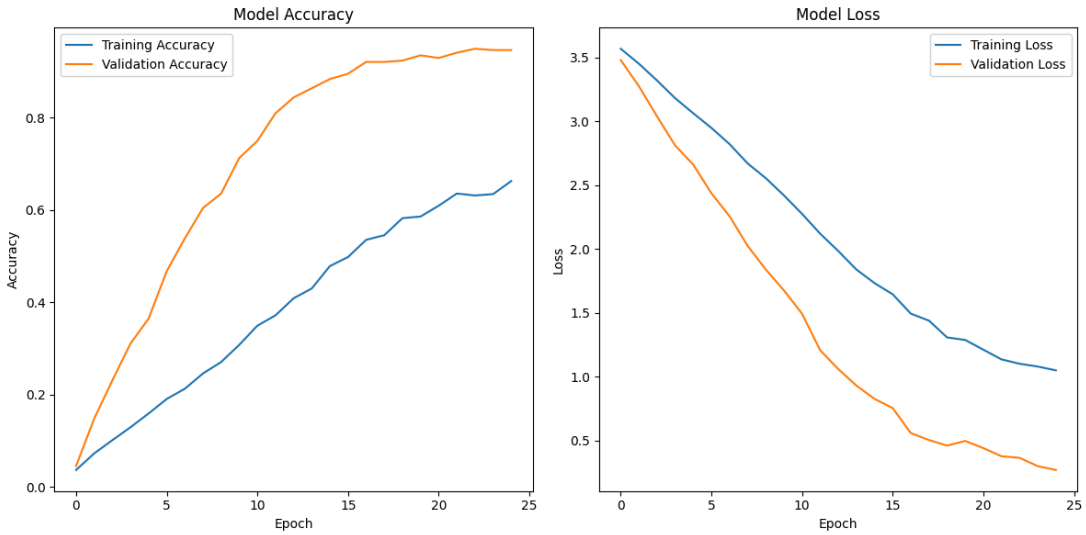
▼ 모델 생성

```
# CNN 모델 정의 (Channel Attention 추가)
def cnn_with_attention():
    inputs = layers.Input(shape=(256, 256, 3))

    # Conv Block 1
    x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = layers.MaxPooling2D((2, 2))(x)
```

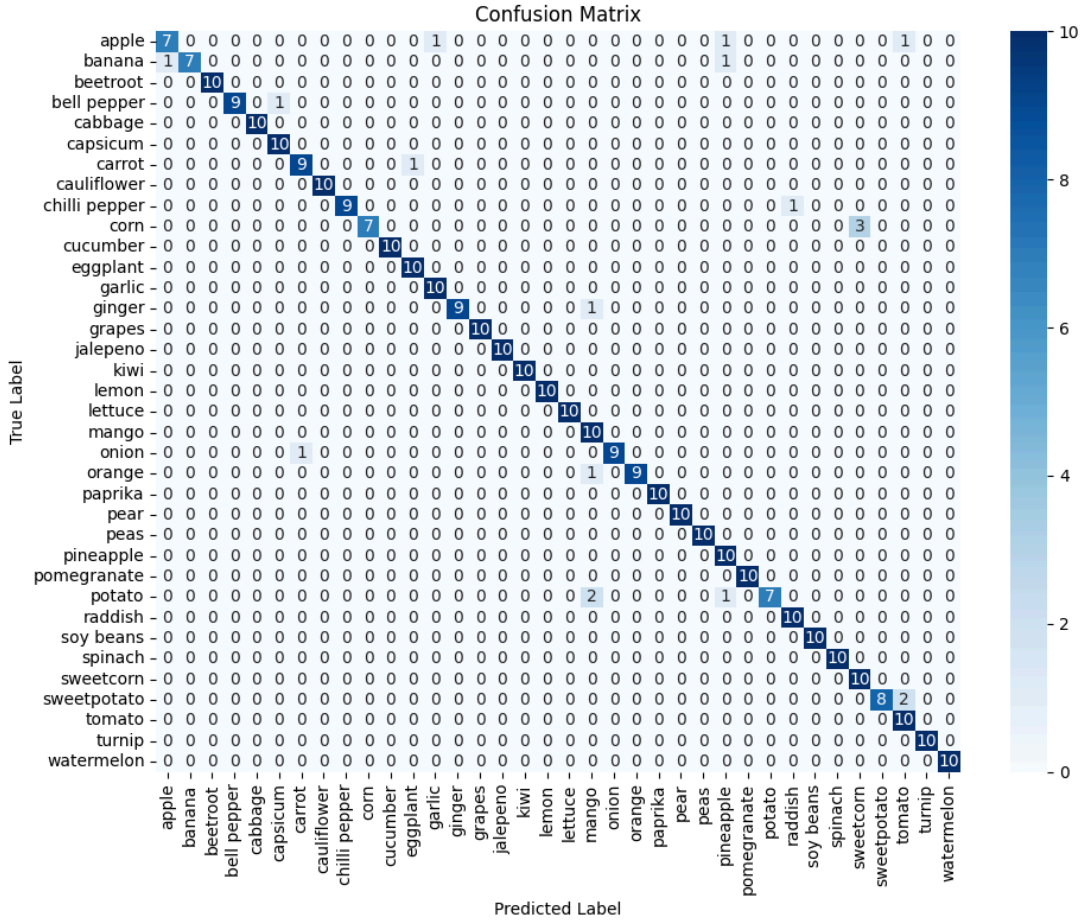
```
# Conv Block 2
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2))(x)

# Conv Block 3
x = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = channel_attention(x)  # Channel Attention 추가

# Fully Connected Layer
x = layers.Flatten()(x)
x = layers.Dense(128, activation='relu')(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(36, activation='softmax')(x)

model = models.Model(inputs, outputs)
return model
```

기존 CNN 컨볼루션 층 이후에 attention블록 추가

▼ 결과



✅ 테스트 데이터 정확도: 94.71%
✅ 테스트 데이터 손실 값: 0.2671



결론

큰 차이는 아니지만 기본 CNN보다 어텐션블록을 적용했을 때 같은 에폭에서 손실이 더 크고 정확도가 좀더 낮음을 알 수 있다.

원인

어텐션 블록을 통해 도출되는 특성벡터는 중요한 특징이 강조되어 주요하지 않은 특징은 값이 작아진다. 이때 이러한 값의 변화가 이전 컨볼루션 층의 가중치 변화를 약하게 하는 현상을 발생시킨다.

또한 어텐션 층이 더해져 학습해야하는 층의 개수가 더욱 많아져 초기 손실 감소폭이 적어지는 원인도 있다.

4. VGG16(Fine - Tuning)

기존 CNN을 통해 어느정도의 성능이 나오는 것을 확인하였으므로 사전학습 모델을 사용하여 성능 비교를 진행

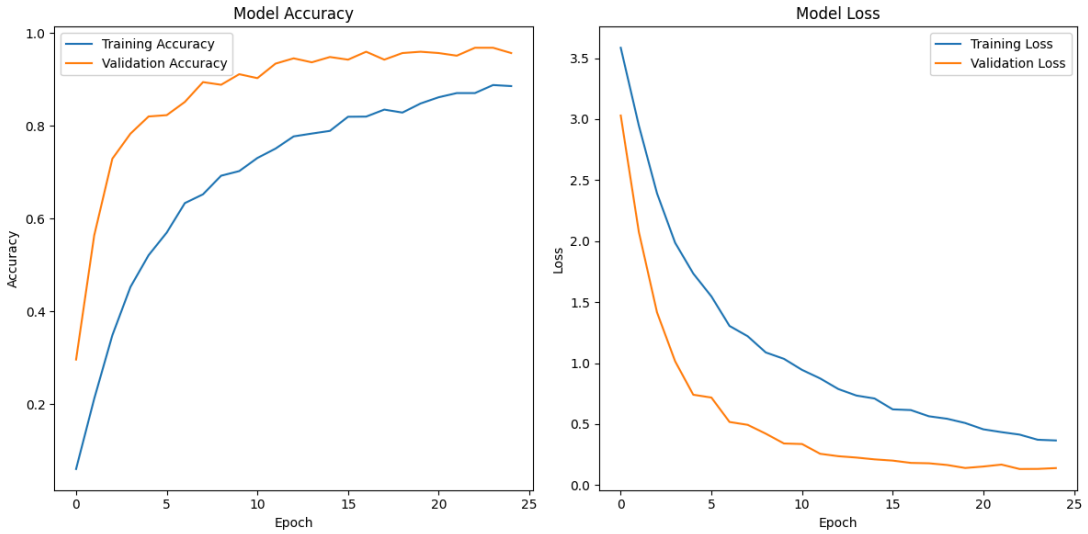
▼ 모델 생성

```
def vgg16_model(input_shape=(256, 256, 3), num_classes=36, dropout_rate=0.5, train_base
=True):
    # 사전학습된 VGG16 모델 (fully connected 층 제외)
    base_model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)

    # 사전학습된 층 학습 여부 설정
    base_model.trainable = train_base

    # 커스터마이징 분류기 추가
```

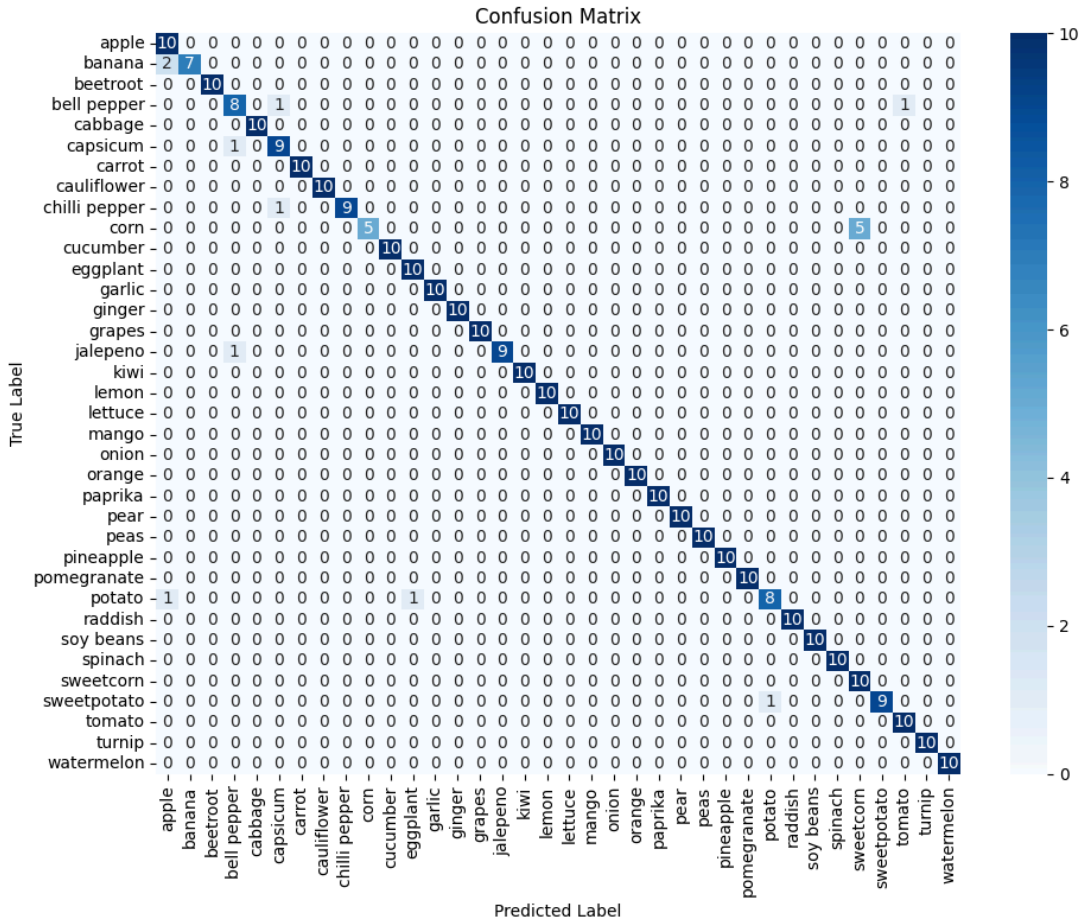
```
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(dropout_rate)(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    # 전체 모델 구성
    model = Model(inputs=base_model.input, outputs=outputs)

    return model
```

레이어 전체를 학습하는 Full Fine Tuning을 진행(학습데이터 = imagenet)

▼ 결과



✅ 테스트 데이터 정확도: 95.82%
✅ 테스트 데이터 손실 값: 0.1388



결론

그래프를 통해 알 수 있듯 같은 에폭을 돌렸을 때 기본 CNN보다는 초기 학습 속도가 늦지만 최종적으론 정확도가 같고 손실이 낮은 것을 볼 수 있다.

데이터셋 자체가 그렇게 복잡한 특성을 지니지 않는 데이터라 확연한 차이가 드러나지 않지만 초기 예상대로 더 높은 성능을 가진다고 할 수 있다.

5. ResNet50 (Fine - Tuning)

VGG와 함께 많이 쓰이는 사전학습 모델인 ResNet을 사용하여 성능비교를 진행

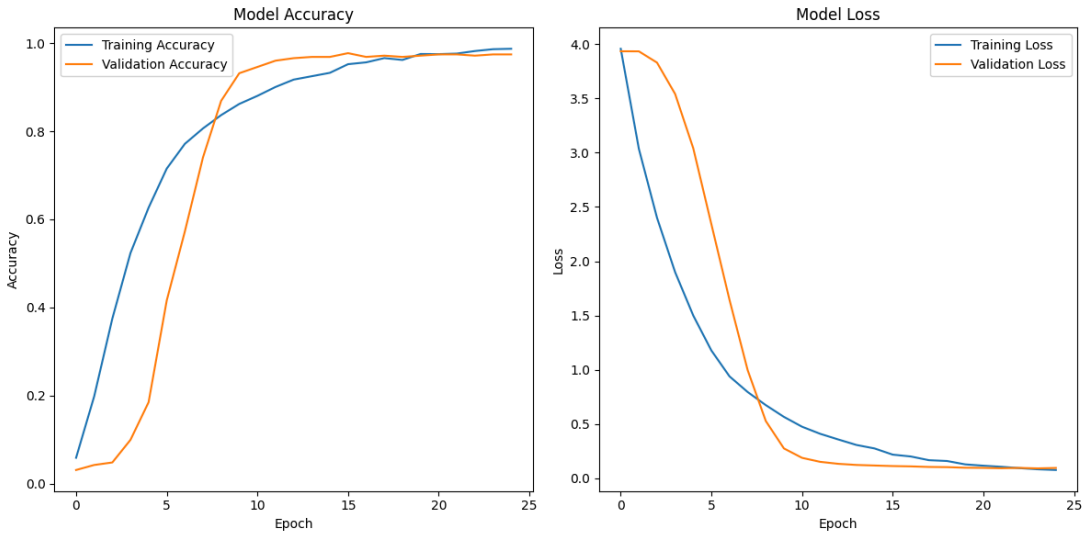
▼ 모델 생성

```
def resnet50_model(input_shape=(256, 256, 3), num_classes=36, dropout_rate=0.5, train_base=True):
    # 사전학습된 ResNet50 모델 (fully connected 층 제외)
    base_model = ResNet50(weights='imagenet', include_top=False, input_shape=input_shape)

    # 사전학습된 층 학습 여부 설정
    base_model.trainable = train_base

    # 커스터마이징 분류기 추가
    x = base_model.output
```

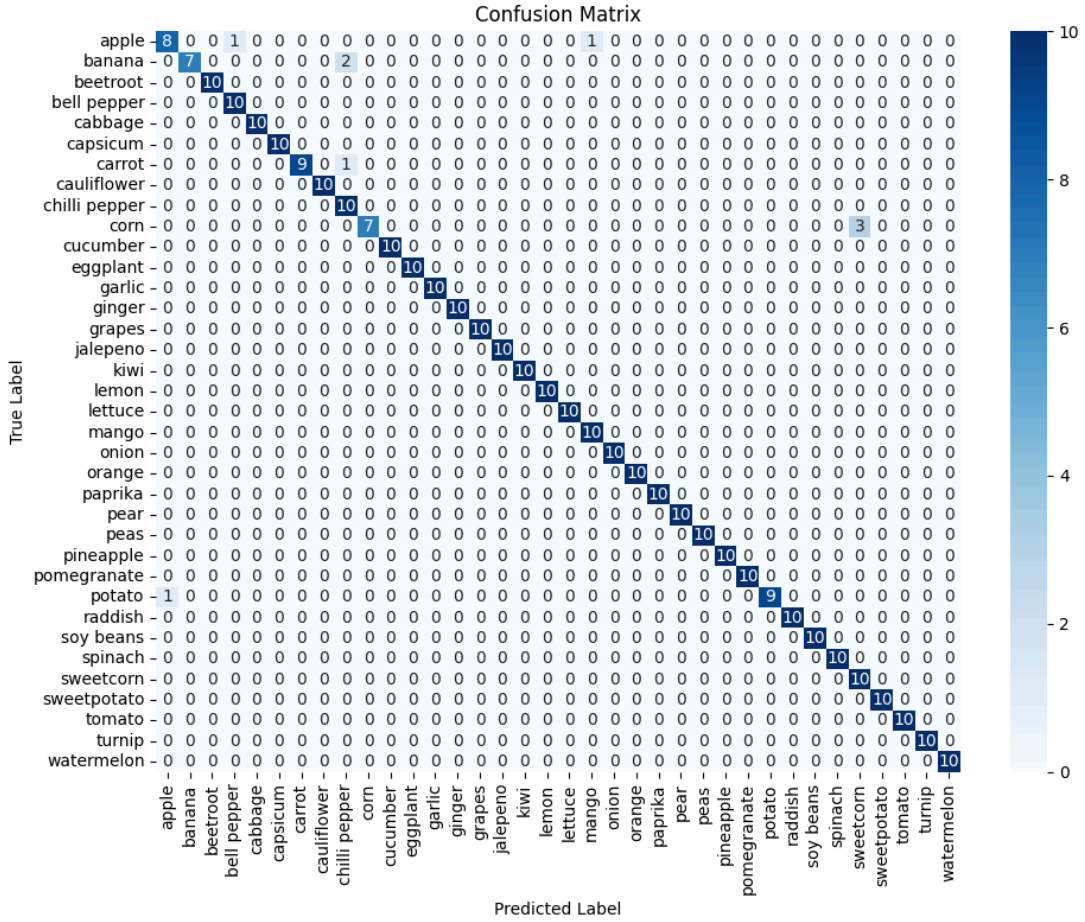
```
    x = GlobalAveragePooling2D()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(dropout_rate)(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    # 전체 모델 구성
    model = Model(inputs=base_model.input, outputs=outputs)

    return model
```

Full FIne Tuning 진행

▼ 결과



✅ 테스트 데이터 정확도: 97.49%
✅ 테스트 데이터 손실 값: 0.0924



결론

ResNet은 VGG와 달리 skip connection을 적용하여 깊은 레이어에서 발생할 수 있는 vanishing gradient문제를 해결한 모델로 학습 진행 그래프를 보면 알 수 있듯 초기 정확도의 증가 폭이 큰 것을 알 수 있다.

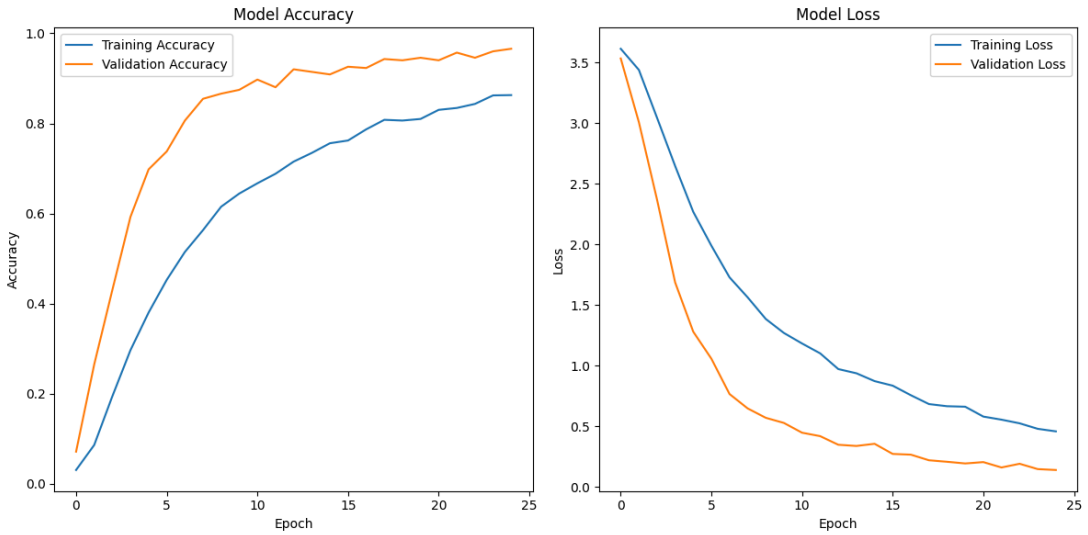
VGG + attention

▼ 모델 생성

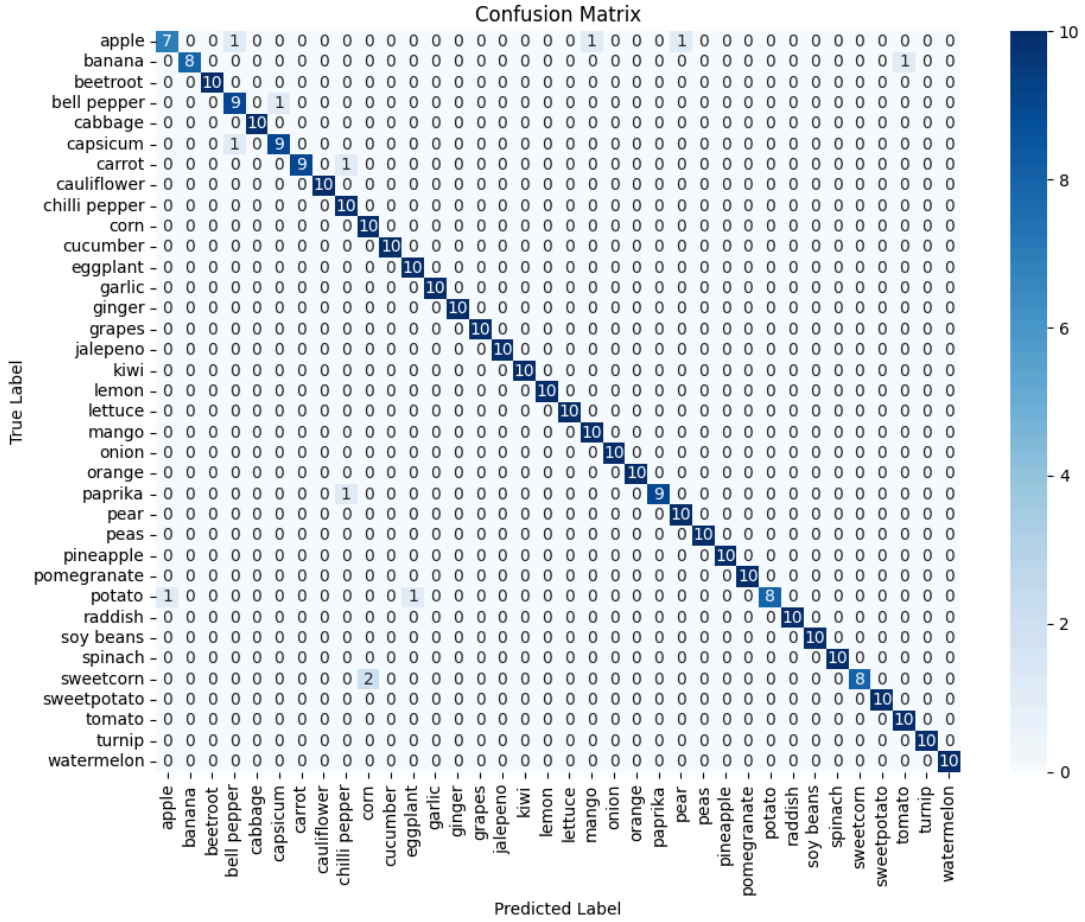
```
def vgg16_attention_model(input_shape=(256, 256, 3), num_classes=36, dropout_rate=0.5, train_base=True):
    # 사전학습된 VGG16 모델 (fully connected 층 제외)
    base_model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)

    # 사전학습된 층 학습 여부 설정
    base_model.trainable = train_base

    # 커스터마이징 분류기 추가
    x = base_model.output
    x = channel_attention(x)
    x = GlobalAveragePooling2D()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(dropout_rate)(x)
    outputs = Dense(num_classes, activation='softmax')(x)
```

```
# 전체 모델 구성
model = Model(inputs=base_model.input, outputs=outputs)

return model
```

▼ **결과**



✅ 테스트 데이터 정확도: 96.66%
✅ 테스트 데이터 손실 값: 0.1396



결론

기존 사전학습 모델인 VGG 모델을 특징 추출기로 사용하고 생성된 벡터에 어텐션을 적용하여 학습을 진행하였는데 오히려 정확도가 떨어진 것을 알 수 있다.

이유는 위 CNN에서 적용된것과 같이 어텐션을 적용하여 초기 학습 속도가 느리고 파라미터 수가 늘어 이러한 현상이 발생한 것을 보임

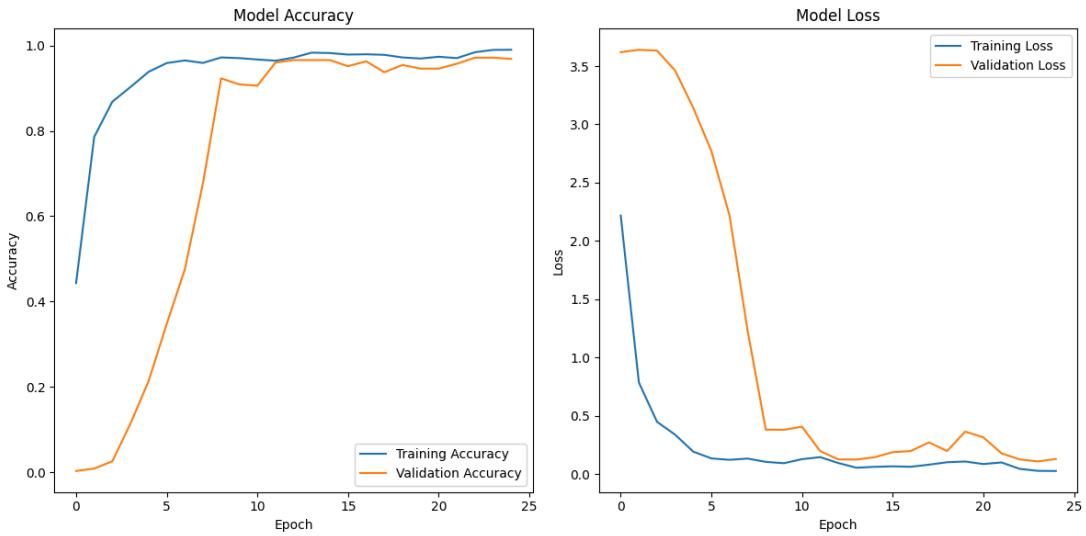
ResNet + attention

▼ 모델 생성

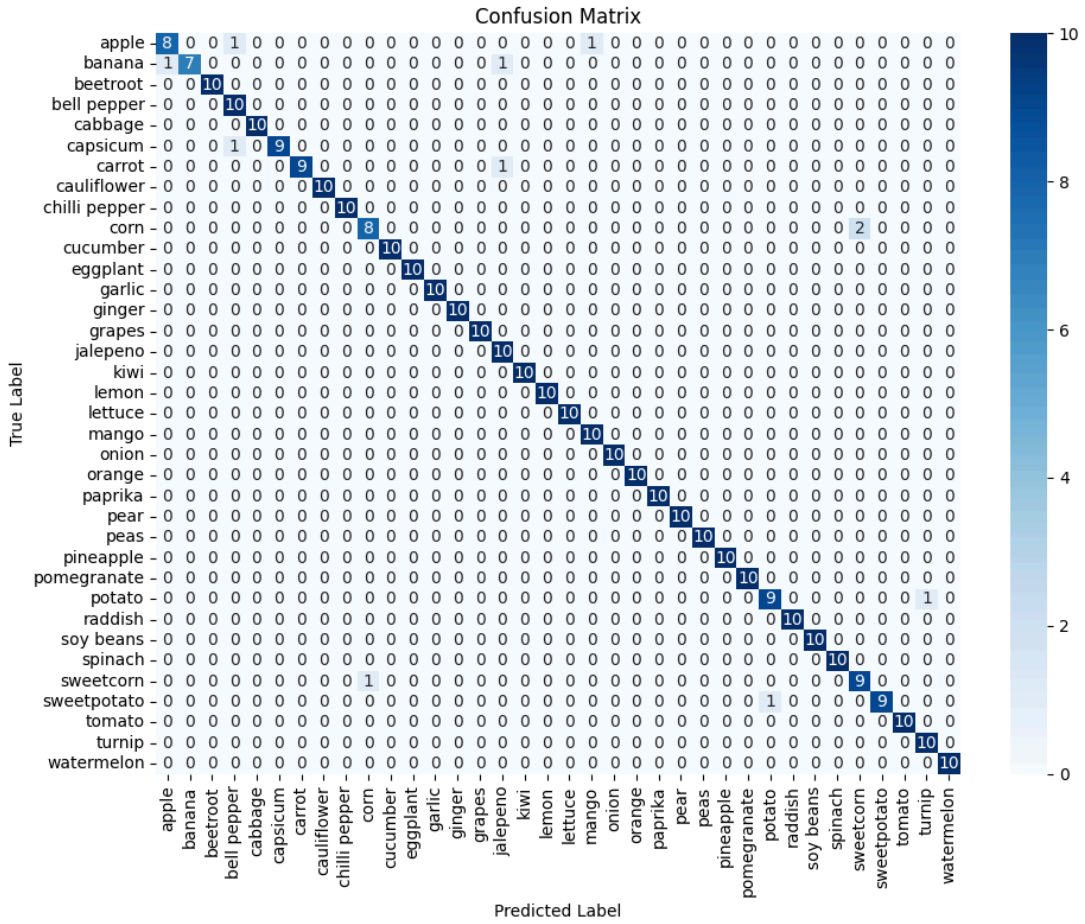
```
def res_attention_model(input_shape=(256, 256, 3), num_classes=36, dropout_rate=0.5, train_base=True):
    # 사전학습된 VGG16 모델 (fully connected 층 제외)
    base_model = ResNet50(weights='imagenet', include_top=False, input_shape=input_shape)

    # 사전학습된 층 학습 여부 설정
    base_model.trainable = train_base

    # 커스터마이징 분류기 추가
    x = base_model.output
    x = channel_attention(x)
```

```
    x = GlobalAveragePooling2D()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(dropout_rate)(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    # 전체 모델 구성
    model = Model(inputs=base_model.input, outputs=outputs)

    return model
```

▼ 결과



✅ 테스트 데이터 정확도: 96.94%
✅ 테스트 데이터 손실 값: 0.1261



결론

ResNet + attention 역시 기존 사전학습 모델만 사용하는 것에 비해 정확도가 낮은 것을 알 수 있다

이 역시 어텐션 블록이 추가되어 간단한 데이터를 사용하였을 때 초기 학습 속도가 느려 그런 것으로 보인다.

DataBase 업로드 적용

모델의 학습데이터를 DB에 업로드하는 과정을 수행

- DB : Aiven(mysql) 사용

▼ DB 사용을 위한 세팅

코랩에서 사용하기 위한 라이브러리 설치

```
!pip install mysql-connector-python
```

SSL접속을 위한 pem 파일 업로드

```
from google.colab import files
uploaded = files.upload()  # 여기서 ca.pem 업로드
```

테이블 생성

```
import mysql.connector

db_config = {
    'host': 'ktb-jensen-chanhue.h.aivencloud.com',
    'user': 'avnadmin',
    'password': 'AVNS_KptliEWmHo9_Mpqw1xs',
    'database': 'defaultdb',
    'port': 21870,
    'ssl_ca': '/content/ca.pem'
}



conn = mysql.connector.connect(**db_config)
cursor = conn.cursor()

create_table_query = """
CREATE TABLE IF NOT EXISTS training_logs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    model_name VARCHAR(100) NOT NULL,
    epoch INT NOT NULL,
    start_time DATETIME NOT NULL,
    end_time DATETIME NOT NULL,
    train_accuracy FLOAT,
    val_accuracy FLOAT,
    train_loss FLOAT,
    val_loss FLOAT
);
"""

cursor.execute(create_table_query)
conn.commit()
cursor.close()
conn.close()

print("✅ training_logs 테이블 생성 완료")
```

로그 저장을 위한 객체 생성

```
import mysql.connector
from datetime import datetime
import tensorflow as tf

class MySQLLogger(tf.keras.callbacks.Callback):
    def __init__(self, model_name, db_config):
        super().__init__()
        self.model_name = model_name
```

```
        self.db_config = db_config

    def on_train_begin(self, logs=None):
        self.conn = mysql.connector.connect(**self.db_config)
        self.cursor = self.conn.cursor()

    def on_epoch_begin(self, epoch, logs=None):
        self.epoch_start_time = datetime.now()

    def on_epoch_end(self, epoch, logs=None):
        epoch_end_time = datetime.now()

        # 로그에서 값 가져오기
        train_acc = logs.get('accuracy')
        val_acc = logs.get('val_accuracy')
        train_loss = logs.get('loss')
        val_loss = logs.get('val_loss')

        query = """
            INSERT INTO training_logs (
                model_name, epoch, start_time, end_time,
                train_accuracy, val_accuracy, train_loss, val_loss
            ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
        """
        values = (
            self.model_name,
            epoch + 1,
            self.epoch_start_time,
            epoch_end_time,
            float(train_acc) if train_acc is not None else None,
            float(val_acc) if val_acc is not None else None,
            float(train_loss) if train_loss is not None else None,
            float(val_loss) if val_loss is not None else None
        )

        self.cursor.execute(query, values)
        self.conn.commit()

    def on_train_end(self, logs=None):
        self.cursor.close()
        self.conn.close()
```

학습함수 수정

```
def train_model(model, model_name, optimizer, train_data, val_data, metrics=['accuracy'], lo
ss_fn='categorical_crossentropy', epochs=25, verbose=1):
    # 저장 경로 설정
    checkpoint_path = f'/content/drive/MyDrive/KTB_personal_project/{model_name}.keras'
```

```
    log_dir = f'/content/drive/MyDrive/KTB_personal_project/logs/{model_name}'

    # DB 설정
    db_config = {
        'host': 'ktb-jensen-chanhue.h.aivencloud.com',
        'port': 21870,
        'user': 'avnadmin',
        'password': 'AVNS_KptliEWmHo9_Mpqw1xs',
        'database': 'defaultdb',
        'ssl_ca': '/content/ca.pem'  # ← 여기가 핵심!
    }



    # 콜백 설정
    checkpoint_cb = ModelCheckpoint(
        filepath=checkpoint_path,
        monitor='val_accuracy',
        save_best_only=True,
        mode='max',
        verbose=1
    )
    tensorboard_cb = TensorBoard(log_dir=log_dir, histogram_freq=1)
    mysql_cb = MySQLLogger(model_name, db_config)

    # 모델 컴파일 & 학습
    model.compile(
        optimizer=optimizer,
        loss=loss_fn,
        metrics=metrics
    )

    history = model.fit(
        train_data,
        validation_data=val_data,
        epochs=epochs,
        verbose=verbose,
        callbacks=[checkpoint_cb, tensorboard_cb, mysql_cb]
    )

    return model, history
```

▼ DB 내용 출력

```
import mysql.connector
import pandas as pd

# Aiven 연결 정보 (이전과 동일)
```

```
db_config = {
    'host': 'ktb-jensen-chanhue.h.aivencloud.com',
    'port': 21870,
    'user': 'avnadmin',
    'password': 'AVNS_KptliEWmHo9_Mpqw1xs',
    'database': 'defaultdb',
    'ssl_ca': '/content/ca.pem'
}

# 출력할 테이블 이름
table_name = "training_logs"  # ← 여기에 원하는 테이블명 넣으면 됨

# 연결 후 데이터 읽기
conn = mysql.connector.connect(**db_config)
query = f"SELECT * FROM {table_name}"
df = pd.read_sql(query, conn)
conn.close()

# 출력
print("✅ 테이블 내용:")
print(df)
```

```
✅ 테이블 내용:
      id     model_name  epoch           start_time             end_time  \
0      1            cnn      1  2025-03-29 11:35:08  2025-03-29 11:36:39
1      2            cnn      2  2025-03-29 11:36:40  2025-03-29 11:38:04
2      3            cnn      3  2025-03-29 11:38:04  2025-03-29 11:39:28
3      4            cnn      4  2025-03-29 11:39:28  2025-03-29 11:40:50
4      5            cnn      5  2025-03-29 11:40:51  2025-03-29 11:42:12
..   ...            ...    ...                  ...                  ...
151  152  res_attention     21  2025-03-29 19:24:58  2025-03-29 19:26:23
152  153  res_attention     22  2025-03-29 19:26:23  2025-03-29 19:27:47
153  154  res_attention     23  2025-03-29 19:27:48  2025-03-29 19:29:15
154  155  res_attention     24  2025-03-29 19:29:16  2025-03-29 19:30:40
155  156  res_attention     25  2025-03-29 19:30:41  2025-03-29 19:32:04

     train_accuracy  val_accuracy  train_loss  val_loss
0          0.054896      0.094017    3.555460  3.402500
1          0.076405      0.165242    3.408160  3.186440
2          0.104655      0.284900    3.257390  2.927810
3          0.136116      0.424501    3.118590  2.698940
4          0.171429      0.424501    2.962610  2.441280
..              ...           ...         ...       ...
151        0.973676      0.945869    0.085815  0.315079
152        0.970465      0.957265    0.099545  0.177949
153        0.984270      0.971510    0.044728  0.125727
154        0.989727      0.971510    0.027568  0.108326
155        0.990048      0.968661    0.026221  0.129004

[156 rows x 9 columns]
```

최종 결론

모델 결과 정리

모델	정확도(%)	손실	Learning Rate
CNN	95.82	0.2348	0.0001
CNN + attention	94.71	0.2671	0.0001
VGG16	95.82	0.1388	0.00001
ResNet50	97.49	0.0924	0.0001
VGG16 + attention	96.66	0.1396	0.00001
ResNet50 + attention	96.94	0.1261	0.0001

CNN vs Pretrained

기본 구조의 CNN과 사전학습 모델의 차이를 살펴보면 확실히 사전학습 모델을 사용한 경우 정확도가 더 높거나 손실이 감소한 것을 알 수 있다

Model vs Model + attention

단순 모델을 사용한 결과와 각 모델에 어텐션 블록을 추가하여 사용한 결과를 비교하였을 때 비교적 정확도가 낮고 손실이 큼을 알 수 있다.

이는 어텐션 블록을 통과한 특성벡터는 값의 범위가 바뀌고 사라지는 값 또한 존재하여 초기 학습속도가 느린 문제가 있고 또한 비교적 단순한 데이터이기에 기존 모델로도 충분한 성능이 잘 나온다는 이유가 복합적으로 작용한 것으로 보인다.

결론

이번 프로젝트는 야채와 과일 이미지를 활용해 여러 분류 모델들을 만들어 보는 프로젝트였다.

기본적인 주제는 사전학습 모델을 활용하는 것이었지만 데이터를 확인해본 결과 사전학습 모델의 사용이 꼭 필요한 것인가에 대한 의문이 들었고 기본 CNN모델을 활용하여 분류 모델을 만들었을 때 사전학습 모델과 어느정도의 성능차이가 존재할지 알아보고자 했다.

전체 과정을 통해 만든 모델은 6가지로 기본 CNN, VGG16, ResNet50과 각 모델에 attention기법을 적용한 모델들이었다.

적합한 비교를 위해 Learning Rate의 차이를 제외하고는 모두 동일한 하이퍼파라미터를 사용하여 비교를 진행하였다.

처음 예상은 사전학습 모델이 더 성능이 높고 attention까지 적용하면 더욱 성능이 좋을 것이라 생각하였다.

결론적으론 사전학습 모델이 성능이 높은것은 맞았지만 attention을 적용한 결과로는 더 성능이 떨어짐을 보였다.

이는 attention 블록을 통해 초기 학습속도가 느려지는 현상이 발생하고 데이터가 기본 모델로도 높은 분류 성능을 보일 만큼 복잡하고 어려운 데이터가 아니었기에 오히려 무거운 모델에서는 반대의 현상을 보인 것으로 보인다.

차후에는 다른 데이터를 사용하여 모델이 더욱 학습하기 어려운 상황에서 각각의 모델이 어떻게 변화하는지를 살펴보고 추가적으로 경량화 기법을 적용하여 생기는 변화 현상을 알아보고자 한다.

참고자료

https://junklee.tistory.com/111

https://ganghee-lee.tistory.com/43

https://lswook.tistory.com/105

https://blog.naver.com/winddori2002/222057978305