[멋사 13기][AI_송원영]

Created by	원 원영 송
Category	개인 과제
Last updated time	@May 30, 2025 10:56 AM

1. 정확도를 높인 위해 시도한 방법과 과정을 노션에 서술 (사진 필수),
2. 정확도 그래프 (matplotlib 시각화),
3. 테스트 결과 (최소 2개 이상)

Test1 : VGG16 전이 학습 (Transfer Learning) + Earlystopping, ModelCheckpoint

VGG16 모델 :

- 사전학습(이미지넷으로 대규모 학습된) 모델의 "특징 추출" 능력을 활용
- 내 데이터셋에 “꼭 맞게” 마지막 분류기만 새로 학습
- 모델 크기가 크고, 학습/추론 속도가 느림/ 무겁지만 성능이 좋음

데이터 분할 (train/val/test) 및 ImageDataGenerator :

- 데이터를 **train(학습)/val(검증)/test(테스트)**로 분할해서 학습

→ 8 : 1 : 1 비율로 분할했음

EarlyStopping, ModelCheckpoint 콜백 도입 :

- EarlyStopping : 증 성능이 n번 연속(5번으로 지정했음 _ 좀 많은 거 같기도..) 좋아지지 않으면 학습 자동 중단, 불필요하게 에폭을 길게 돌려 “과적합” 방지, 시간·리소스도 절약

- ModelCheckpoint : 학습 중간중간 “가장 val_accuracy가 좋았던 순간”의 가중치를 파일로 저장, 최종적으로 “가장 잘 맞춘 모델”만 활용 가능

전이학습 :

- 분류기 : VGG 16이 뽑은 특징을 받아서 고양이인지, 개인지 최종적으로 판단하는 부분
- VGG 16 : 이미지에서 특징을 뽑아내는 부분 (합성곱 층이라 함.)
- 따라서 합성곱 층 고정이라는 것은 이미지 넷에서 미리 배운 가중치를 그대로 고정시키고, 분류기를 학습하여 내 데이터에 맞게끔 분류기를 학습하는 것

```
earlystop = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=
checkpoint = ModelCheckpoint('best_vgg16_model.h5', monitor='val_accuracy', s

history = model.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
    callbacks=[earlystop, checkpoint] // 여기에 있음!
)
```

- 코드

```
import os
import zipfile
import shutil
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_im
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
```

→ **TensorFlow의 Keras** 에서 `tensorflow.keras.applications` 라이브러리에 있는 VGG16 모델 import

```
# 1) data.zip 압축 해제
with zipfile.ZipFile('data.zip', 'r') as zip_ref:
    zip_ref.extractall('.')

# 2) train/val/test 폴더로 분할 (한 번만 실행, 이미 분할된 경우 생략)
classes = ['cat', 'dog']
split_ratio = [0.8, 0.1, 0.1]  # train:val:test

for split in ['train', 'val', 'test']:
    for c in classes:
        os.makedirs(os.path.join('data_split', split, c), exist_ok=True)

for c in classes:
    images = os.listdir(os.path.join('data', c))
    np.random.shuffle(images)
    n_total = len(images)
    n_train = int(n_total * split_ratio[0])
    n_val = int(n_total * split_ratio[1])
    train_files = images[:n_train]
    val_files = images[n_train:n_train + n_val]
    test_files = images[n_train + n_val:]

    for fname in train_files:
        shutil.copy(os.path.join('data', c, fname), os.path.join('data_split', 'train', c, fn
    for fname in val_files:
        shutil.copy(os.path.join('data', c, fname), os.path.join('data_split', 'val', c, fna
    for fname in test_files:
        shutil.copy(os.path.join('data', c, fname), os.path.join('data_split', 'test', c, fn

print("train/val/test 분할 완료!")
# 이후 사용할 폴더 경로 변수
train_dir = 'data_split/train'
```

```
val_dir = 'data_split/val'
test_dir = 'data_split/test'
```

→ 학습(train), 검증(val), 테스트(test)용으로각각**data_split/train/cat, data_split/val/dog**등 **새로운 폴더 구조 만듦**

→ 80%는**학습(train)**, 10%는**검증(val)**, 10%는**테스트(test)**비율로 **분할** (train/val/test 파일 리스트 생성)

```
img_height, img_width = 64, 64 # 모든 이미지를 64x64 크기로 변환
batch_size = 32 # 한 번에 32장씩 모델에 전달(미니배치)

train_datagen = ImageDataGenerator(
    rescale=1./255, # 픽셀값을 0~1로 정규화(모델 학습 안정화)
    rotation_range=20, # 최대 20도까지 이미지 회전(랜덤)
    width_shift_range=0.2, # 이미지 좌우 이동(20%)
    height_shift_range=0.2, # 이미지 상하 이동(20%)
    horizontal_flip=True # 이미지를 좌우로 랜덤 뒤집기
)
val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    train_dir, # 학습 이미지 폴더
    target_size=(img_height, img_width), # 이미지 크기 변환
    batch_size=batch_size,
    class_mode='categorical'
)
val_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical'
)
test_generator = test_datagen.flow_from_directory(
    test_dir,
```

```
    target_size=(img_height, img_width),
    batch_size=1,
    class_mode='categorical',
    shuffle=False
)

num_classes = train_generator.num_classes
print("클래스 수:", num_classes)
```

→ 이미지 배치 생성기 만듦 (폴더 → 배치)

```
Found 6402 images belonging to 2 classes.
Found 800 images belonging to 2 classes.
Found 803 images belonging to 2 classes.
클래스 수: 2
```

train(학습) 폴더에 6402장의 이미지

validation(검증) 800장

test(테스트) 803장

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(img_
base_model.trainable = False  # 전이학습 (ImageNet 데이터로 미리 학습된 가중치)

x = base_model.output
x = GlobalAveragePooling2D()(x) # CNN의 3차원 출력을 1차원으로 요약
x = Dropout(0.3)(x) # 학습 중 일부 뉴런을 무작위로 꺼서 과적합 방지
x = Dense(128, activation='relu')(x)
x = Dropout(0.3)(x)
outputs = Dense(num_classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=outputs) # 최종 모델 객체 생성
model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossen
model.summary() # 모델의 전체 구조, 파라미터 수를 표로 보여줌
```

→ VGG16(이미지넷 사전학습)을 고양이/개 분류기로 데이터에 맞게 구성 및 학습 준비

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 3)	0
block1_conv1 (Conv2D)	(None, 64, 64, 64)	1,792
block1_conv2 (Conv2D)	(None, 64, 64, 64)	36,928
block1_pool (MaxPooling2D)	(None, 32, 32, 64)	0
block2_conv1 (Conv2D)	(None, 32, 32, 128)	73,856
block2_conv2 (Conv2D)	(None, 32, 32, 128)	147,584
block2_pool (MaxPooling2D)	(None, 16, 16, 128)	0
block3_conv1 (Conv2D)	(None, 16, 16, 256)	295,168
block3_conv2 (Conv2D)	(None, 16, 16, 256)	590,080
block3_conv3 (Conv2D)	(None, 16, 16, 256)	590,080
block3_pool (MaxPooling2D)	(None, 8, 8, 256)	0
block4_conv1 (Conv2D)	(None, 8, 8, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 8, 8, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 8, 8, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 4, 4, 512)	0
block5_conv1 (Conv2D)	(None, 4, 4, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 4, 4, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 4, 4, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 2, 2, 512)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 512)	0
dropout (Dropout)	(None, 512)	0
dense (Dense)	(None, 128)	65,664
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 2)	258

Total params: 14,780,610 (56.38 MB)
Trainable params: 65,922 (257.51 KB)
Non-trainable params: 14,714,688 (56.13 MB)

→ VGG16이 이미 배운 “특징 추출” 부분은 고정시키고, 데이터(고양이/개) 분류를 위한 마지막 Dense 층만 학습하는 “전이학습” 모델의 전체 구조표

```
earlystop = EarlyStopping(
monitor='val_loss', # 검증 데이터의 손실값(val_loss)을 지켜본다
patience=3, # 3번 연속(val_loss가) 개선되지 않으면 학습을 멈춘다
restore_best_weights=True, # 성능이 가장 좋았던 순간의 가중치로 자동 복원
verbose=1) # 중단될 때 메시지 출력

checkpoint = ModelCheckpoint('best_vgg16_model.h5', # 저장 파일 이름
monitor='val_accuracy', # 검증 정확도(val_accuracy)를 기준으로 저장
save_best_only=True, # 이전보다 성능이 좋아지면만 저장
verbose=1) # 저장될 때마다 메시지 출력

history = model.fit(
    train_generator, # 학습 데이터
    epochs=10, # 최대 10번 반복 (조기 종료로 더 일찍 멈출 수도 있)
    validation_data=val_generator, # 검증 데이터
    callbacks=[earlystop, checkpoint] # 위 두 콜백 사용!
)
```

```
Epoch 1/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.5887 - loss: 0.6956
Epoch 1: val_accuracy improved from -inf to 0.77750, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy.
201/201 ━━━━━━━━━━━━━━━━━━━━ 26s 105ms/step - accuracy: 0.5890 - loss: 0.6954 - val_accuracy: 0.7775 - val_loss: 0.5051
Epoch 2/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 88ms/step - accuracy: 0.6745 - loss: 0.5898
Epoch 2: val_accuracy did not improve from 0.77750
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 93ms/step - accuracy: 0.6746 - loss: 0.5897 - val_accuracy: 0.7513 - val_loss: 0.4948
Epoch 3/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step - accuracy: 0.7082 - loss: 0.5514
Epoch 3: val_accuracy improved from 0.77750 to 0.77875, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy.
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 88ms/step - accuracy: 0.7083 - loss: 0.5514 - val_accuracy: 0.7788 - val_loss: 0.4789
Epoch 4/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step - accuracy: 0.7206 - loss: 0.5411
Epoch 4: val_accuracy improved from 0.77875 to 0.78250, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy.
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 93ms/step - accuracy: 0.7206 - loss: 0.5411 - val_accuracy: 0.7825 - val_loss: 0.4799
Epoch 5/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.7149 - loss: 0.5530
Epoch 5: val_accuracy improved from 0.78250 to 0.78875, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy.
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 91ms/step - accuracy: 0.7149 - loss: 0.5529 - val_accuracy: 0.7887 - val_loss: 0.4691
Epoch 6/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 88ms/step - accuracy: 0.7260 - loss: 0.5391
Epoch 6: val_accuracy did not improve from 0.78875
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 93ms/step - accuracy: 0.7260 - loss: 0.5391 - val_accuracy: 0.7625 - val_loss: 0.4907
Epoch 7/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - accuracy: 0.7231 - loss: 0.5351
Epoch 7: val_accuracy did not improve from 0.78875
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 89ms/step - accuracy: 0.7231 - loss: 0.5351 - val_accuracy: 0.7788 - val_loss: 0.4571
Epoch 8/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - accuracy: 0.7113 - loss: 0.5424
Epoch 8: val_accuracy did not improve from 0.78875
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 92ms/step - accuracy: 0.7113 - loss: 0.5424 - val_accuracy: 0.7725 - val_loss: 0.4754
Epoch 9/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step - accuracy: 0.7326 - loss: 0.5312
Epoch 9: val_accuracy did not improve from 0.78875
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 95ms/step - accuracy: 0.7326 - loss: 0.5313 - val_accuracy: 0.7800 - val_loss: 0.4817
Epoch 10/10
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step - accuracy: 0.7294 - loss: 0.5283
Epoch 10: val_accuracy did not improve from 0.78875
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 93ms/step - accuracy: 0.7294 - loss: 0.5283 - val_accuracy: 0.7837 - val_loss: 0.4590
Epoch 10: early stopping
Restoring model weights from the end of the best epoch: 7.
```

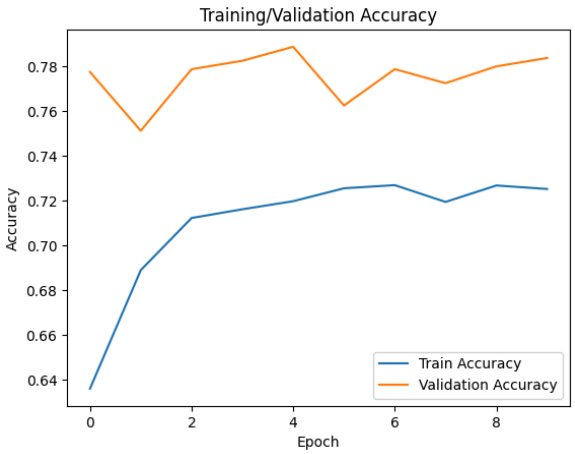
결과 :

- 학습 초반(1에폭) : 학습 정확도(accuracy) 58.9%, 검증 정확도(val_accuracy) 77.8%
- 학습 중~후반 : 학습 정확도는 상승세로 72~73% 도달, 검증 정확도는 약 77~79%
- 최고 검증 정확도 : 78.9%
- Early Stopping : Epoch 10: early stopping 으로 10에폭 동안 val_accuracy가 더 좋아지지 않으므로, 최고 성능(7번째 에폭)의 모델 가중치로 복원 (과적합 가능성 줄이기)

```
# 훈련/검증 정확도 그래프
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch') # x축: 에폭(학습 반복 횟수)
plt.ylabel('Accuracy') # y축: 정확도
plt.legend() # 범례(Train/Validation)
plt.title('Training/Validation Accuracy')
```

```
plt.show()

# 훈련 / 검증 손실 그래프
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss') # y축: 손실(낮을수록 좋음)
plt.legend()
plt.title('Training/Validation Loss')
plt.show()
```

훈련/검증 정확도 그래프



훈련/검증 모두 성능이 점점 좋아지는 추세

→ 과적합(훈련만 오르고 검증은 떨어지는 경우)이 나타나지 않음

- **파란색(Train Accuracy):**

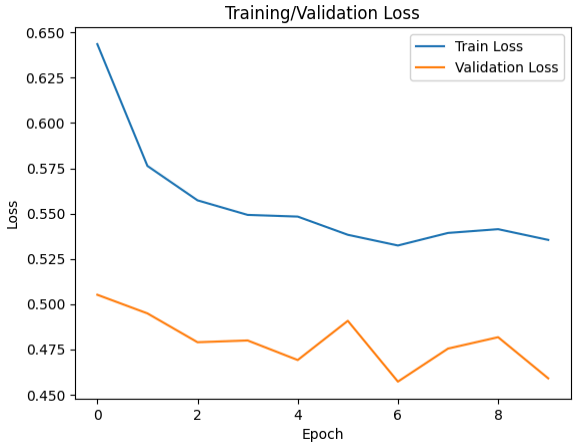
  학습 데이터에 대한 정확도

  → 약 0.64 → 0.72 부근까지 상승

- **주황색(Validation Accuracy):**

  검증 데이터(모델이 본 적 없는 데이터) 정확도

  → 초반부터 높고, 약간 오르내리지만 0.75~0.78 근방 유지



훈련/검증 손실 그래프

- **파란색(Train Loss):**

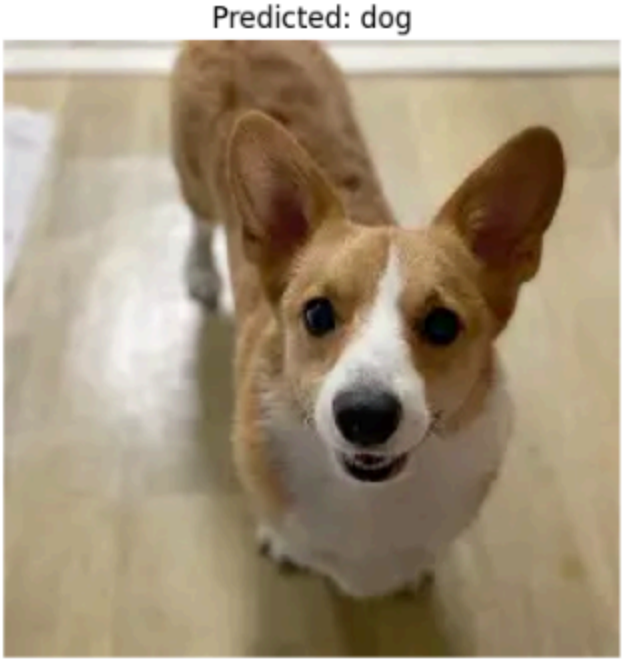
  점점 감소, 즉 학습이 잘 이루어지고 있음

- **주황색(Validation Loss):**

  전체적으로 줄고 있음 (중간에 살짝 요동)

```
from google.colab import files

uploaded = files.upload()  # 여기서 이미지 선택
for fn in uploaded.keys():
    img = load_img(fn, target_size=(img_height, img_width))
    img_array = img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)

    model.load_weights('best_vgg16_model.h5')
    pred = model.predict(img_array)
    pred_class = np.argmax(pred, axis=1)[0]
    class_indices = {v:k for k,v in train_generator.class_indices.items()}

    print("예측 클래스:", class_indices[pred_class])

    plt.imshow(load_img(fn))
    plt.title(f"Predicted: {class_indices[pred_class]}")
    plt.axis('off')
    plt.show()
```

```
파일 선택 개1.webp
• 개1.webp(image/webp) - 3994 bytes, last modified: 2025. 5. 27. - 100% done
Saving 개1.webp to 개1.webp
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
예측 클래스: dog
```



Test2 : VGG16 전이 학습 파인튜닝

- 기존 방식과 똑같이 분류기 5에폭 학습, VGG 16(합성층 곱)은 고정, 분류기(Dense)만 학습
- 기존 Test1과 달라진 점은 기존 전이학습과 동일하게 Dense(분류층)을 학습하지만, CGG16의 마지막 block5(합성곱층 뒷부분)을 학습했음.
- learning rate를 아주 작게 줄여서 기존의 좋은 가중치가 망가지지 않게 함.
- fine-tuning : 기존에 잘 배운 뇌의 일부까지 내 문제에 맞게 새로 훈련시키는 것

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# 1. 콜백 정의 (그대로)
earlystop = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=
checkpoint = ModelCheckpoint('best_vgg16_model.h5', monitor='val_accuracy', s
```

```
# 검증손실이 val_loss가 3번 연속 개선 안될 경우 학습을 멈추고 가장 성능이 좋은 모델 가중
# 학습 중 검증 정확도가 높을때마다 best_vgg16_model.h5로 저장

# 2. 먼저 "분류기만" 학습 (warm-up)
history = model.fit(
    train_generator,
    epochs=5, # 최대 5번 학습 반복.
    validation_data=val_generator,
    callbacks=[earlystop, checkpoint]
)
```

→ VGG16의 본체(특징추출)는 고정해두고, 내 데이터로 “분류기(Dense 층)”만 5번 에폭 동안 먼저 학습시키는 코드

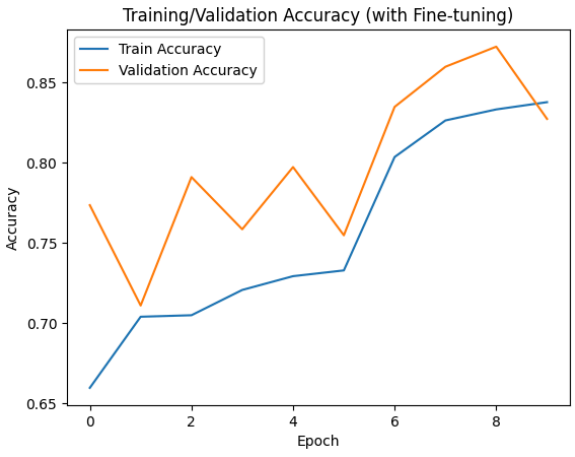
```
# 3. VGG16 뒷부분 일부만 학습 허용 (fine-tuning 단계)
for layer in base_model.layers:
    if 'block5' in layer.name:    # block5_로 시작하는 레이어만 추가학습 허용
        layer.trainable = True

# 4. 학습률을 충분히 낮춤! (기존보다 10배 낮게)
# 이미 잘 학습된 가중치가 크게 흔들리지 않도록 learning rate를 기존 0.001보다 낮은 0.0
model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crosse

# 5. 다시 전체 학습 (fine-tuning) block5 + Dense 까지 함께 학습
history_ft = model.fit(
    train_generator,
    epochs=5,   # 추가로 5번만 더! (EarlyStopping이 있으니 더 빨리 멈출 수 있음)
    validation_data=val_generator,
    callbacks=[earlystop, checkpoint]
)
```

→ VGG16의 마지막 block5 합성곱 레이어만 내 데이터로 학습을 하용하여 세밀하게 데이터 맞추는 파인튜닝, 기존보다 작은 learning rate로 block5 + Dense 모두 다시 학습

```
  self._warn_if_super_not_called()
Epoch 1/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.6268 - loss: 0.6545
Epoch 1: val_accuracy improved from -inf to 0.77347, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considere
201/201 ━━━━━━━━━━━━━━━━━━━━ 29s 119ms/step - accuracy: 0.6270 - loss: 0.6544 - val_accuracy: 0.7735 - val_loss: 0.499
Epoch 2/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 85ms/step - accuracy: 0.7099 - loss: 0.5565
Epoch 2: val_accuracy did not improve from 0.77347
201/201 ━━━━━━━━━━━━━━━━━━━━ 20s 98ms/step - accuracy: 0.7098 - loss: 0.5566 - val_accuracy: 0.7109 - val_loss: 0.5364
Epoch 3/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.7029 - loss: 0.5635
Epoch 3: val_accuracy improved from 0.77347 to 0.79099, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considere
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 91ms/step - accuracy: 0.7029 - loss: 0.5635 - val_accuracy: 0.7910 - val_loss: 0.4804
Epoch 4/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 90ms/step - accuracy: 0.7157 - loss: 0.5489
Epoch 4: val_accuracy did not improve from 0.79099
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 96ms/step - accuracy: 0.7157 - loss: 0.5489 - val_accuracy: 0.7584 - val_loss: 0.4855
Epoch 5/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - accuracy: 0.7315 - loss: 0.5417
Epoch 5: val_accuracy improved from 0.79099 to 0.79725, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considere
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 90ms/step - accuracy: 0.7314 - loss: 0.5417 - val_accuracy: 0.7972 - val_loss: 0.4647
Restoring model weights from the end of the best epoch: 5.
Epoch 1/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.6815 - loss: 0.5946
Epoch 1: val_accuracy did not improve from 0.79725
201/201 ━━━━━━━━━━━━━━━━━━━━ 27s 106ms/step - accuracy: 0.6817 - loss: 0.5943 - val_accuracy: 0.7547 - val_loss: 0.439
Epoch 2/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 88ms/step - accuracy: 0.7976 - loss: 0.4389
Epoch 2: val_accuracy improved from 0.79725 to 0.83479, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considere
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 96ms/step - accuracy: 0.7977 - loss: 0.4389 - val_accuracy: 0.8348 - val_loss: 0.3584
Epoch 3/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.8223 - loss: 0.4033
Epoch 3: val_accuracy improved from 0.83479 to 0.85982, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considere
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 91ms/step - accuracy: 0.8224 - loss: 0.4033 - val_accuracy: 0.8598 - val_loss: 0.3327
Epoch 4/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step - accuracy: 0.8304 - loss: 0.3846
Epoch 4: val_accuracy improved from 0.85982 to 0.87234, saving model to best_vgg16_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considere
201/201 ━━━━━━━━━━━━━━━━━━━━ 19s 96ms/step - accuracy: 0.8304 - loss: 0.3846 - val_accuracy: 0.8723 - val_loss: 0.3123
Epoch 5/5
201/201 ━━━━━━━━━━━━━━━━━━━━ 0s 85ms/step - accuracy: 0.8417 - loss: 0.3493
Epoch 5: val_accuracy did not improve from 0.87234
201/201 ━━━━━━━━━━━━━━━━━━━━ 18s 89ms/step - accuracy: 0.8417 - loss: 0.3493 - val_accuracy: 0.8273 - val_loss: 0.3741
Restoring model weights from the end of the best epoch: 4.
```

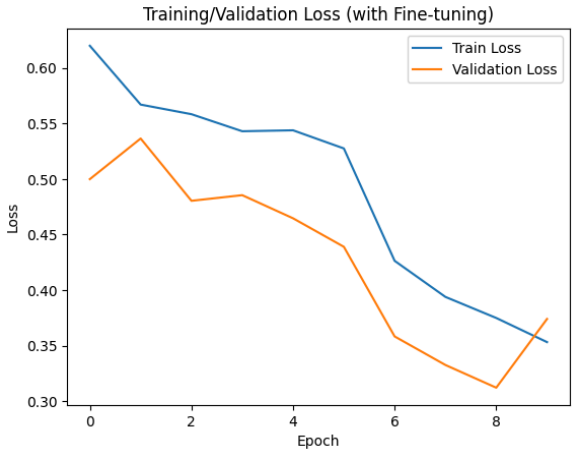
- 첫 번째 fit (분류기 학습, epochs = 5)
  - 첫 단계 에서 **최고 검증 정확도(val_accuracy)는 0.7992 (약 79.7%)**
  - Restoring model weights from the end of the best epoch: 5. → EarlyStopping 작동(최고 epoch 5로 복원)
- 두 번째 fit (파인튜닝, epochs = 5)
  - 파인튜닝(fine-tuning)에서 **최고 검증 정확도는 0.8723 (약 87.2%)**
  - Restoring model weights from the end of the best epoch: 4. → 최고 성능이 4번째 epoch, 그 시점의 모델로 복원
  - 분류기만 학습한 것보다 **정확도가 크게 올랐음**

```
# 1. 에폭별 정확도와 손실값 합치기
acc = history.history['accuracy'] + history_ft.history['accuracy']
val_acc = history.history['val_accuracy'] + history_ft.history['val_accuracy']
loss = history.history['loss'] + history_ft.history['loss']
val_loss = history.history['val_loss'] + history_ft.history['val_loss']

# 2. 정확도 시각화
plt.plot(acc, label='Train Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training/Validation Accuracy (with Fine-tuning)')
plt.show()

# 3. 손실 시각화
plt.plot(loss, label='Train Loss')
plt.plot(val_loss, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training/Validation Loss (with Fine-tuning)')
plt.show()
```



Training/Validation Accuracy (정확도 그래프)

- **Train Accuracy(파란색)**:
  - 점점 상승하다가, 0.72 부근에서 안정됨
- **Validation Accuracy(주황색)**:
  - 초기부터 높게 시작해서 0.75~0.79 근처에서 약간의 변화 있음



Training/Validation Loss (손실 그래프)

- **Train Loss(파란색)**:
  - 꾸준히 감소, 학습이 잘 진행됨을 의미
- **Validation Loss(주황색)**:
  - 전체적으로 줄어드는 추세, 중간에 약간의 변동은 있지만 다시 내려감

해석 요약

- **학습/검증 모두 성능이 꾸준히 올라가고, 손실도 잘 줄어듦**
- 과적합(Train은 계속 오르는데 Validation은 멈추거나 떨어지는 경우)이 보이지 않음

```
from google.colab import files

uploaded = files.upload()  # 여기서 이미지 선택
for fn in uploaded.keys():
    img = load_img(fn, target_size=(img_height, img_width))
    img_array = img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)

    model.load_weights('best_vgg16_model.h5')
    pred = model.predict(img_array)
    pred_class = np.argmax(pred, axis=1)[0]
    class_indices = {v:k for k,v in train_generator.class_indices.items()}

    print("예측 클래스:", class_indices[pred_class])

    plt.imshow(load_img(fn))
    plt.title(f"Predicted: {class_indices[pred_class]}")
```

```
plt.axis('off')
plt.show()
```

한 줄 소감

VGG 모델이 꽤 무겁다보니 에폭이 3번째인데도 실행시간이 20분이 넘어가다보니...너무 오래 걸려서 모델을 바꿀까 하다가, 코랩 런타임 바꾸기에서 cpu를 gpu로 바꿨더니 3분만에 됐다. 너무 속이 시원했다 ! ;-)

(하지만 gpu는 사용량이 제한되어 있으므로 사용안할 때는 다시 cpu로 바꿔야 한다!)