[멋사 13기] [AI 이민형]

목표

- 전체 과제 목표: 73% 이상으로 정확도 높이기(80%까지?) + AI와 친해지기
- 내 목표:
    1. 80% 이상으로 높이기
        - 과제 목표보다 더 높이고 싶은 욕심이 생겼다.
        - 사용할수 있는 방법을 모두 써보면 어디까지 올라갈 수 있는지 궁금해졌다.
    2. 그래프를 예쁘게 만들어보자
        - 정확도를 높이면서도 학습정확도와 검증정확도를 최대한 일치하게
            - 일반화가 잘된 모델

3. 학습 목표:용어, 개념 다시한번 복습하면서 정리, 의문점들을 정리해보자

기본 코드 및 문제상황 분석

이미지 전처리

- 전체 8005개의 이미지 → (64,64) 크기로 통일 → 이미지 1개의 입력데이터 크기 약 **4096**
- 1개 이미지의 픽셀 (64,64) → 각 픽셀값 정규화(0~1)
  - 정규화 하는이유? → 픽셀값 통일?
    - 픽셀값: 0~255 정수형 → 정규화: 픽셀값을 0~1 사이 값으로 바꿔줌
    - 학습값 너무 큰 경우 학습이 불안해짐, 통일성X
- class_mode:정답이 될수 있는 class 를 설정:
  - binary OR etc. → 이진분류, 다중 분류 등 정답이 되는 class 정하기
  - 여기서는 고양이, 강아지 두 종류뿐이라서 binary 사용

class_mode 옵션 종류

값	반환되는 라벨 형식	사용 상황
`'categorical'`	원-핫 인코딩된 벡터	다중 클래스 분류 (예: 고양이, 강아지, 말)
`'binary'`	0 또는 1	**이진 분류** (예: 고양이 vs 강아지)
`'sparse'`	정수형 라벨 (0, 1, 2, ...)	다중 클래스지만 정수 레이블만 필요할 때
`'input'`	이미지 자체가 입력도 되고 출력도 됨	**오토인코더** 같은 입력=출력인 모델
`None`	라벨 없이 이미지만 반환	예측(추론)용, 비지도학습 등

- batch_size: 한번의 학습의 몇개의 데이터가 들어가는가?
  - batch_size= 5: 1개의 배치에 5개의 이미지가 들어간다.
  - 총 배치 개수: 8005 / 5 = 1601 ...몫:0, 나머지:0→ 1601개의 배치
    - **전체 데이터 수가 `batch_size` 로 나눠 떨어지지 않으면?**

        - 이미지 개수가 8003개 였다면? → 5개씩 들어가는 배치 다발 1600개 → 마지막 배치는 3개만
            - keras는 기본적으로 이 남는 불완전한 배치도 학습에 포함시켜줌
            - drop_last = True 하면 마지막 불완전 배치는 버림
                - 그럼 안좋은거 아닌가요?
    - 배치는 왜 랜덤으로 들어가나요?

validation_split=0.2 # ← 여기!
이건 전체 데이터의 20%를 validation set으로 자동 분리하겠다는 뜻이야.

즉, 8005장의 이미지 중:

6404장 (80%) → 학습용 (train)

1601장 (20%) → 검증용 (validation)

변경점 생각나는거

1. batch_size : 한번에 학습하는 다발에 몇개를 넣을건가?
    a. 전체 8005개의 이미지 데이터 → (64,64) 크기로 통일 → 입력

1차: 최적의 batch_size

배치가 너무 작으면

- 시간이 오래걸리고 노이즈가 큼
    - ?? 한번 학습할때 들어가는 이미지가 적게 들어가니까
    - 예를들면 한 배치에 2개씩 넣으면 그만큼 배치 다발 개수가 늘어나고 다발이 늘어난 만큼 학습 횟수가 늘어남
        - 횟수가 늘어나면 시간이 더 오래걸림
        - 대신 학습 횟수가 늘면 그만큼 더 똑똑해지는거 아니에요?

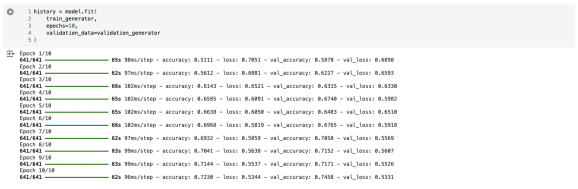
- 정답은 아니다. 배치를 하는 이유가 없어진다. 미니배치로 하는이유는 랜덤 데이터를 다발로 묶어서 학습에 넣으면서 관련성 적거나 의미없거나 중요하지 않거나 필요하지 않은 데이터를 걸러낼수 있다? → 분산? 분포?
- 배치에 들어가는 개수가 줄면 그만큼 배치 자체의 개수는 늘고 그럼 노이즈가 더 늘어난다→ 학습에 불필요한 데이터의 반영이 늘어난다.
- 미니 배치를 하는 이유는?

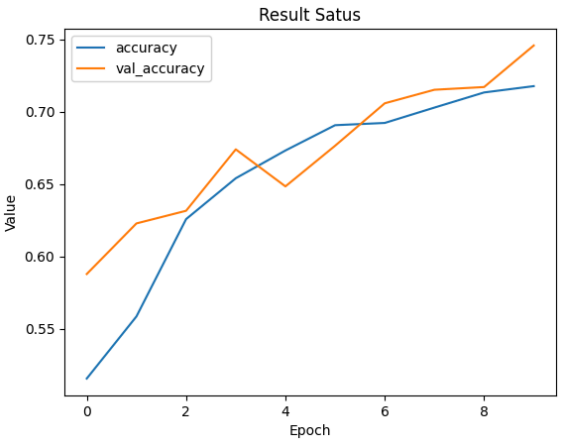
배치가 너무 크면

- local minima
  - 손실함수 그래프 전체에서의 최저지점을 찾아야하는데 각 구간별 작은지점에서 벗어나지 못하고 가장 최소지점인곳을 찾지 못하고 (일정구간)지역의 최소 지점을 전체 최소점이라고 착각하게됨
  - 근데 크다고 그렇게 되나요??? 왜죠???? 이동하는게 줄어들어서?

batch_size 5 → 10 으로 변경

- 실행결과
  - **첫번째** 에폭: 학습 정확도: 51%, 검증 정확도: 58%
  - **최종** 에폭 학습 정확도: **72**%, 검증 정확도: **74**%





- 왜 이렇게 되었나?

$$\text{배치\_개수} = \left\lceil \frac{\text{전체\_이미지\_개수}}{batch\_size} \right\rceil$$

- 학습용(train)

  6404/10 = 640.4 → 641개의 배치,

  - 마지막 배치는 0.4만큼 채워져 불완전한 배치가 될수 있다.
  - Keras에서는 `drop_last=False` 가 기본이라 그 배치도 학습에 포함
  - ImageDataGenerator에는 해당 옵션이 없음 → 우리 코드에서는 무조건 포함
- 검증용(validation)
- 1601/10=160.1 →161개 배치
  - 마지막 배치에는 입력 이미지 데이터가 1개만 포함될수 있음 →
  - `ImageDataGenerator` 에서는 무조건 포함되므로 그 배치도 학습에 포함
  - 평가에는 사용되므로 학습에 손실X

- 그래프 분석
  - train과 val 그래프가 거의 평행하게 큰 차이 없이 같이 상승
  - 검증 정확도가 학습 정확도보다 높음
    - 학습정확도가 더 높은게 일반적이지만, 초기에 검증정확도가 높은 경우도 흔하다 (Regulariztion 효과)
    - 나쁜 현상은 아니지만, 너무 오래 지속되면 ovrregularization 가능성도 생각해봐야함
- batch=10의 평가
  - 장점
    - 정확도 높음: 검증용 기준 74%
    - 과적합여부: 없음, 적정수준
    - 학습 안정성: 그래프에서 급격한 진동없이 안정적
  - 단점:
    - 속도가 느리다 →학습 시간이 오래걸림
      - 배치 사이즈 작을수록 업데이트 횟수 많아짐 → 학습 시간 많아짐

- 전체 데이터 6404장을 배치당 10으로 처리하면 641의 배치
- `steps_per_epoch= 641` 10번 반복 → 6400번 넘는 업데이트 발생

steps_per_epoch: 에폭당 수행하는 업데이트 횟수

- step: 한번의 파라미터 업데이트를 위한 연산 단위
  - 한 배치를 모델에 넣고 순전파 역전파 가중치 업데이트 한번 수행 → 1 **step**

- 노이즈에 민감
  - 작은 배치는 데이터 편향에 더 민감하다

    → 한 배치에 고양이만 몰리거나 개만 몰리면 모델이 쉽게 이에 휘둘린다.

    → 훈련중 정확도가 불안정하게 요동칠수 있음 → 다행이 이 그래프에서는 안정적
- 과도한 지역 최소점 탐색 (Local minima)
  - 배치가 작으면 손실함수의 곡선의 미세한 변화까지 민감하게 반응 → 전체 방향성보다 작은 패턴에 과도하게 반응할수 있음 → 전체적 최적화 품질 낮아질수 잇음

→ 더 좋은 global minima를 못 찾을 수도 있음

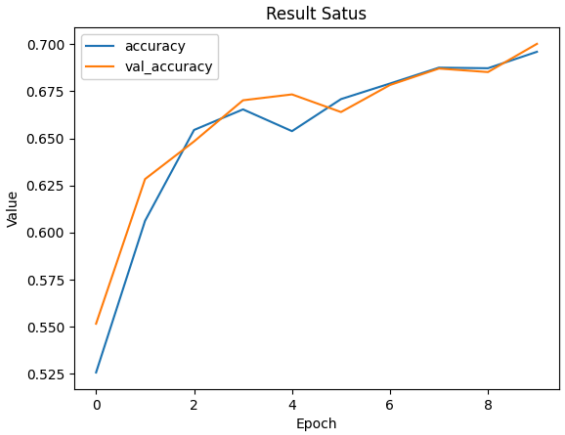
batch_size= 10→ 100

- 실행 결과

초반: 학습정확도: 51%, 검증정확도:55%

마지막: 학습정확도: **68**%, 검증정확도: **70**%

```
Epoch 1/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 60s 876ms/step - accuracy: 0.5158 - loss: 0.7242 - val_accuracy: 0.5515 - val_loss: 0.6777
Epoch 2/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 56s 859ms/step - accuracy: 0.5748 - loss: 0.6747 - val_accuracy: 0.6284 - val_loss: 0.6416
Epoch 3/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 57s 870ms/step - accuracy: 0.6597 - loss: 0.6259 - val_accuracy: 0.6483 - val_loss: 0.6226
Epoch 4/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 61s 931ms/step - accuracy: 0.6693 - loss: 0.6180 - val_accuracy: 0.6702 - val_loss: 0.6031
Epoch 5/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 56s 859ms/step - accuracy: 0.6636 - loss: 0.6128 - val_accuracy: 0.6733 - val_loss: 0.6122
Epoch 6/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 56s 861ms/step - accuracy: 0.6795 - loss: 0.5975 - val_accuracy: 0.6640 - val_loss: 0.5986
Epoch 7/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 56s 861ms/step - accuracy: 0.6804 - loss: 0.5985 - val_accuracy: 0.6783 - val_loss: 0.5866
Epoch 8/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 57s 869ms/step - accuracy: 0.6902 - loss: 0.5860 - val_accuracy: 0.6871 - val_loss: 0.5834
Epoch 9/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 55s 850ms/step - accuracy: 0.6795 - loss: 0.5977 - val_accuracy: 0.6852 - val_loss: 0.5886
Epoch 10/10
65/65 ━━━━━━━━━━━━━━━━━━━━ 56s 867ms/step - accuracy: 0.6887 - loss: 0.5822 - val_accuracy: 0.7002 - val_loss: 0.5677
```



- 그래프: accurancy와 val_accurancy 그래프는 초반에는 차이가 있지만 에폭이 진행되면서 일치되는 지점이 생기다가 지속적으로 같아지는 부분이 생긴다. 최종에는 다시 val_accurancy가 accurancy보다 더 커지면서 다시 차이가 벌어진다.
- 6404/100 = 64 → 65개의 배치
- accurancy와 val_accurancy그래프에서 각 곡선이 일치되는 지점이 많고 아름다운 모양새이지만, 정확도 자체는 낮다

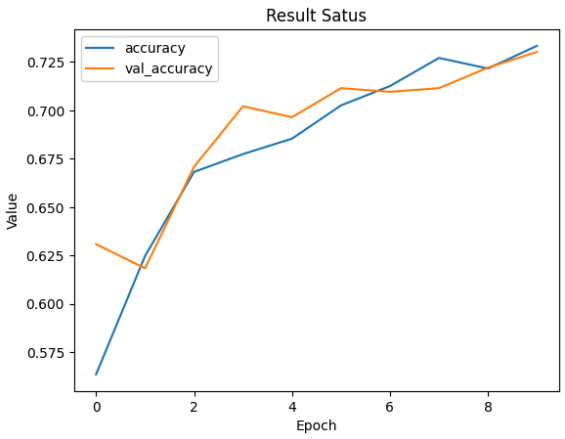
- 장점:
  - 그래프 곡선이 거의 평행 → 일반화가 잘된 모델
  - step(배치 개수가) 수가 적어서 학습 속도 빠름
  - 정확도가 안정적으로 증가하는 모양새
- 단점
  - 최종 정확도가 batch_size=10 일때보다 낮다 (74%→70%)
  - batch_size가 너무 커서 local minima에 갇힐 위험이 존재한다
  - gradient nois가 적어서 세세한 특징 학습은 다소 약하다

그래프는 acc이랑 val_acc이랑 거의 일치하지만 정확도 자체는 처음보다 오히려 줄었네

학습 안정성, 속도, 일반화 성능 면에서는 좋지만 최종 정확도는 배치가 10일때보다 뒤쳐짐

batch_size= 100 → 50

```
Epoch 1/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 54s 402ms/step - accuracy: 0.5366 - loss: 0.6944 - val_accuracy: 0.6309 - val_loss: 0.6517
Epoch 2/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 50s 384ms/step - accuracy: 0.6220 - loss: 0.6536 - val_accuracy: 0.6184 - val_loss: 0.6352
Epoch 3/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 49s 382ms/step - accuracy: 0.6713 - loss: 0.6101 - val_accuracy: 0.6708 - val_loss: 0.5985
Epoch 4/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 53s 410ms/step - accuracy: 0.6775 - loss: 0.5939 - val_accuracy: 0.7021 - val_loss: 0.5709
Epoch 5/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 51s 393ms/step - accuracy: 0.6868 - loss: 0.5888 - val_accuracy: 0.6964 - val_loss: 0.5771
Epoch 6/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 47s 367ms/step - accuracy: 0.6962 - loss: 0.5772 - val_accuracy: 0.7114 - val_loss: 0.5539
Epoch 7/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 401ms/step - accuracy: 0.7084 - loss: 0.5541 - val_accuracy: 0.7096 - val_loss: 0.5525
Epoch 8/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 85s 423ms/step - accuracy: 0.7304 - loss: 0.5346 - val_accuracy: 0.7114 - val_loss: 0.5521
Epoch 9/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 53s 412ms/step - accuracy: 0.7283 - loss: 0.5363 - val_accuracy: 0.7220 - val_loss: 0.5641
Epoch 10/10
129/129 ━━━━━━━━━━━━━━━━━━━━ 50s 385ms/step - accuracy: 0.7398 - loss: 0.5244 - val_accuracy: 0.7302 - val_loss: 0.5264
```



- 그래프: 둘다 거의 일치하면서 나란히 상승 중
- 초반부터 검증정확도가 학습정확도에 근접 → 일반화에 우수
- 그래프 마지막 부분에서 곡선 끝이 꺾이지 않고 더 상승할 여지를 보여줌

6404/50 = 129

최종 정확도: 학습 74%, 검증 73%

훈련 검증 거의 일치

학습 시간 중간

그래프: 꾸준한 상승, 진동 없음

단점: 50은 GPU 병렬 효율에서 64보다 비효율 → 64는 2제곱 GPU 메모리 블럭과 유사한 형태

129 step이라 100(64)에 비해 느림

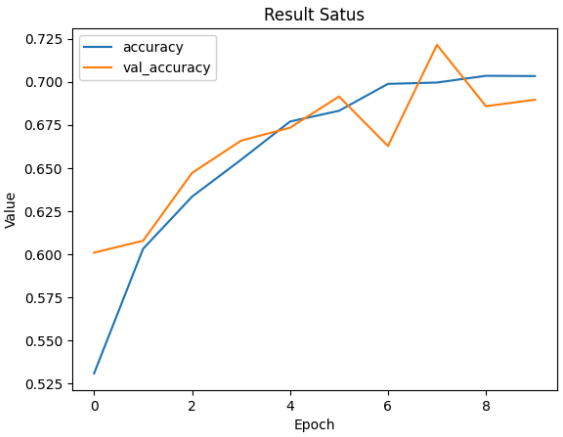
정확도 상승 폭은 낮아서 에폭을 늘리면 더 의미있는 변화 생길것으로 기대

batch_size= 50 → 64

- 실행 결과
  - 초반: 학습정확도: 52%, 검증정확도:60%
  - 마지막: 학습정확도: 70%, 검증정확도: 68%
  - 최고점: epoch=7에서 검증정확도: 72%

- 로그 및 그래프 분석
  - 학습그래프는 꾸준히 상승
  - 검증그래프도 상승하지만 끝무렵(5~7 epoch)에 진동(출렁임)

```
Epoch 1/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 57s 545ms/step - accuracy: 0.5205 - loss: 0.7352 - val_accuracy: 0.6009 - val_loss: 0.6832
Epoch 2/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 53s 524ms/step - accuracy: 0.5909 - loss: 0.6737 - val_accuracy: 0.6077 - val_loss: 0.6483
Epoch 3/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 53s 525ms/step - accuracy: 0.6338 - loss: 0.6402 - val_accuracy: 0.6471 - val_loss: 0.6310
Epoch 4/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 54s 537ms/step - accuracy: 0.6616 - loss: 0.6170 - val_accuracy: 0.6658 - val_loss: 0.5994
Epoch 5/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 54s 531ms/step - accuracy: 0.6808 - loss: 0.6023 - val_accuracy: 0.6733 - val_loss: 0.5986
Epoch 6/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 54s 536ms/step - accuracy: 0.6764 - loss: 0.5935 - val_accuracy: 0.6914 - val_loss: 0.5747
Epoch 7/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 55s 539ms/step - accuracy: 0.6966 - loss: 0.5750 - val_accuracy: 0.6627 - val_loss: 0.5996
Epoch 8/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 59s 579ms/step - accuracy: 0.7008 - loss: 0.5713 - val_accuracy: 0.7214 - val_loss: 0.5640
Epoch 9/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 54s 531ms/step - accuracy: 0.7150 - loss: 0.5527 - val_accuracy: 0.6858 - val_loss: 0.5765
Epoch 10/10
101/101 ━━━━━━━━━━━━━━━━━━━━ 81s 526ms/step - accuracy: 0.7082 - loss: 0.5652 - val_accuracy: 0.6896 - val_loss: 0.5686
```



- 50일때 제안된 단점을 극복하기 위해 GPU 메모리 블록에 적합한 사이즈로 변경해 진행했지만 학습 정확도, 검증 정확도 모두 `batch_size=64` 보다 `batch_size=50` 인 경우가 더 높았다.
- 이때 새로운 의문이 들었다. 에폭이 10이 아닌경우에도 `batch_size=50` 가 더 적절한 사이즈일까? 배치 사이즈나 입력데이터 종류와 개수에 따라 적합한 에폭이 달라질까?

  이러한 이유로 다음 테스트는 에폭을 변경하며 적절한 에폭을 찾아보았다. 또한 해당 에폭에 맞는 배치 사이즈도 있는지 확인해보았다.

2차:최적의 epoch 찾기

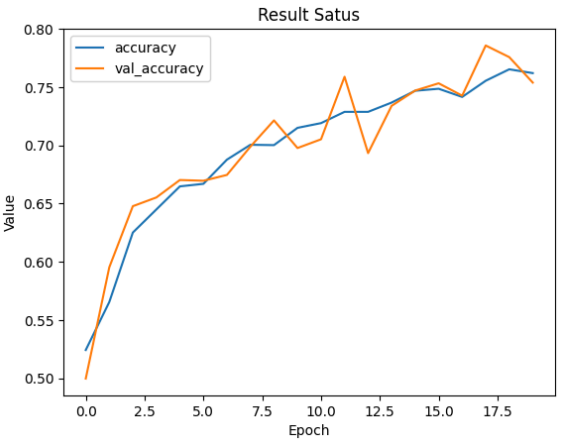
우선 가장 높은 정확도를 보인 `batch_size=50` 에서 출발하였다.

epoch= 10 → 20

- 실행 결과

- 초반: 학습정확도: 52%, 검증정확도:49%
- 마지막: 학습정확도: 76%, 검증정확도: 75%
- 최고점: epoch=18에서 검증정확도: 78%

```
Epoch 1/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 53s 398ms/step - accuracy: 0.5215 - loss: 0.7097 - val_accuracy: 0.4997 - val_loss: 0.6974
Epoch 2/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 399ms/step - accuracy: 0.5353 - loss: 0.6847 - val_accuracy: 0.5953 - val_loss: 0.6540
Epoch 3/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 50s 386ms/step - accuracy: 0.6023 - loss: 0.6502 - val_accuracy: 0.6477 - val_loss: 0.6259
Epoch 4/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 48s 373ms/step - accuracy: 0.6509 - loss: 0.6193 - val_accuracy: 0.6552 - val_loss: 0.6270
Epoch 5/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 50s 385ms/step - accuracy: 0.6650 - loss: 0.6071 - val_accuracy: 0.6702 - val_loss: 0.6075
Epoch 6/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 51s 394ms/step - accuracy: 0.6550 - loss: 0.6128 - val_accuracy: 0.6696 - val_loss: 0.5992
Epoch 7/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 85s 417ms/step - accuracy: 0.6877 - loss: 0.5856 - val_accuracy: 0.6746 - val_loss: 0.5844
Epoch 8/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 53s 414ms/step - accuracy: 0.6905 - loss: 0.5785 - val_accuracy: 0.6989 - val_loss: 0.5718
Epoch 9/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 51s 392ms/step - accuracy: 0.7013 - loss: 0.5621 - val_accuracy: 0.7214 - val_loss: 0.5589
Epoch 10/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 53s 407ms/step - accuracy: 0.7160 - loss: 0.5543 - val_accuracy: 0.6977 - val_loss: 0.5842
Epoch 11/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 403ms/step - accuracy: 0.7176 - loss: 0.5460 - val_accuracy: 0.7052 - val_loss: 0.5738
Epoch 12/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 404ms/step - accuracy: 0.7231 - loss: 0.5450 - val_accuracy: 0.7589 - val_loss: 0.5132
Epoch 13/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 405ms/step - accuracy: 0.7399 - loss: 0.5256 - val_accuracy: 0.6933 - val_loss: 0.5836
Epoch 14/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 406ms/step - accuracy: 0.7283 - loss: 0.5204 - val_accuracy: 0.7339 - val_loss: 0.5163
Epoch 15/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 52s 405ms/step - accuracy: 0.7420 - loss: 0.5324 - val_accuracy: 0.7470 - val_loss: 0.5091
Epoch 16/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 54s 417ms/step - accuracy: 0.7451 - loss: 0.5097 - val_accuracy: 0.7533 - val_loss: 0.5157
Epoch 17/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 49s 376ms/step - accuracy: 0.7395 - loss: 0.5140 - val_accuracy: 0.7427 - val_loss: 0.5072
Epoch 18/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 47s 367ms/step - accuracy: 0.7554 - loss: 0.4976 - val_accuracy: 0.7858 - val_loss: 0.4773
Epoch 19/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 48s 372ms/step - accuracy: 0.7647 - loss: 0.4768 - val_accuracy: 0.7758 - val_loss: 0.4797
Epoch 20/20
129/129 ━━━━━━━━━━━━━━━━━━━━ 49s 383ms/step - accuracy: 0.7672 - loss: 0.4804 - val_accuracy: 0.7539 - val_loss: 0.5006
```



- 학습 정확도: 꾸준히 상승
- 검증 정확도: 조금씩 출렁이면서도 꾸준히 상승
- 그래프: 두 정확도가 거의 나란히 상승하고 있다. 에폭 13~20애서 성능이 안정적으로 수렴하고 있다. val_accuracy는 18~19ep에 **최고치 0.7858, 0.7758**에 도달

  → 과적합 없이 안정적으로 일반화된 학습이 이루어졌다고 볼 수 있음

  → 18~19epoch이 적절한 `EarlyStopping` 적용 위치

- 에폭:20 , 배치:50에 대한 평가

항목	평가
정확도	20ep에서 최종 정확도, 최고 정확도 모두 향상
일반화 성능	과적합 없이 최고 성능 도달, val_loss도 지속 감소
적정 에폭 수	Epoch 18~19에서 최고치 도달 → EarlyStopping 이상적
권장 설정	`batch_size=50` , `epochs≈18~20` , 필요 시 `EarlyStopping` 적용

→ valid_accurancy에 출렁임이 조금씩 있는데 이게 적절한걸까? 일반화가 잘 된걸까?

이전 실험과 비교

실험 조건	최종 검증 정확도	최고 검증 정확도	val_loss 최저
batch=50, epoch=10	0.7302	0.7302	0.5264
batch=64, epoch=10	0.6896	0.724	0.5640
batch=50, epoch=20	**0.7539**	**0.7858**	**0.4773**

`batch_size=50` 로 설정하고 에폭을 20까지 늘리자 학습 정확도는 0.76, 검증 정확도는 0.75를 넘기며

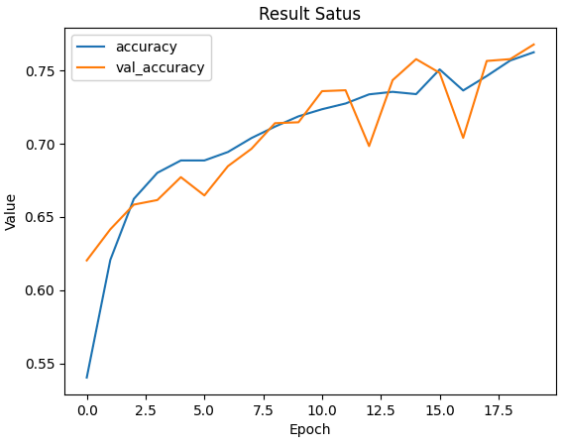
이전 실험 대비 가장 높은 성능을 달성했다. 특히 Epoch 18~19에서 최고 검증 정확도 0.7858을 기록했으며,

과적합 없이 val_loss가 지속적으로 감소해 일반화 성능 또한 우수하였다.

epoch= 20, batch_size=64

에폭을 더 늘려서 적용하니 더 높은 정확도와 성능을 보여주었다. 이때 배치를 다시 64로 변경해서 에폭 20에 더 적합한 배치를 찾아보겠다.

```
Epoch 1/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 24s 209ms/step - accuracy: 0.5137 - loss: 0.7009 - val_accuracy: 0.6202 - val_loss: 0.6578
Epoch 2/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 178ms/step - accuracy: 0.6072 - loss: 0.6509 - val_accuracy: 0.6415 - val_loss: 0.6341
Epoch 3/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 187ms/step - accuracy: 0.6683 - loss: 0.6136 - val_accuracy: 0.6583 - val_loss: 0.6133
Epoch 4/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 180ms/step - accuracy: 0.6671 - loss: 0.5987 - val_accuracy: 0.6615 - val_loss: 0.6090
Epoch 5/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 188ms/step - accuracy: 0.6919 - loss: 0.5728 - val_accuracy: 0.6771 - val_loss: 0.6019
Epoch 6/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.6944 - loss: 0.5814 - val_accuracy: 0.6646 - val_loss: 0.5974
Epoch 7/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 186ms/step - accuracy: 0.6988 - loss: 0.5735 - val_accuracy: 0.6846 - val_loss: 0.5747
Epoch 8/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 178ms/step - accuracy: 0.6988 - loss: 0.5654 - val_accuracy: 0.6964 - val_loss: 0.5607
Epoch 9/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 188ms/step - accuracy: 0.7073 - loss: 0.5681 - val_accuracy: 0.7139 - val_loss: 0.5550
Epoch 10/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 178ms/step - accuracy: 0.7232 - loss: 0.5446 - val_accuracy: 0.7146 - val_loss: 0.5471
Epoch 11/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 186ms/step - accuracy: 0.7251 - loss: 0.5403 - val_accuracy: 0.7358 - val_loss: 0.5293
Epoch 12/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 182ms/step - accuracy: 0.7287 - loss: 0.5341 - val_accuracy: 0.7364 - val_loss: 0.5325
Epoch 13/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 186ms/step - accuracy: 0.7332 - loss: 0.5381 - val_accuracy: 0.6983 - val_loss: 0.6029
Epoch 14/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7359 - loss: 0.5241 - val_accuracy: 0.7433 - val_loss: 0.5265
Epoch 15/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 185ms/step - accuracy: 0.7314 - loss: 0.5278 - val_accuracy: 0.7577 - val_loss: 0.5089
Epoch 16/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 178ms/step - accuracy: 0.7496 - loss: 0.5015 - val_accuracy: 0.7483 - val_loss: 0.5192
Epoch 17/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 185ms/step - accuracy: 0.7562 - loss: 0.4839 - val_accuracy: 0.7039 - val_loss: 0.5582
Epoch 18/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7430 - loss: 0.5139 - val_accuracy: 0.7564 - val_loss: 0.5045
Epoch 19/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 20s 202ms/step - accuracy: 0.7470 - loss: 0.5153 - val_accuracy: 0.7577 - val_loss: 0.4880
Epoch 20/20
101/101 ━━━━━━━━━━━━━━━━━━━━ 20s 197ms/step - accuracy: 0.7588 - loss: 0.4983 - val_accuracy: 0.7676 - val_loss: 0.4802
```



- 실행 결과
  - 초반: 학습정확도: 51%, 검증정확도: 62%
  - 마지막: 학습정확도: 75%, 검증정확도: 76%
  - 최고점: epoch=15에서 검증정확도: 75%
- 결과 분석

    - 정확도 최고 수치는 `batch_size=50` 때보다 낮았지만, 최종 검증정확도는 오히려 `batch_size=64` 가 더 높게 나왔다. 또한, `batch_size=64` 의 검증 그래프는 아직 완전히 수렴하지 않고 epoch=20에도 최고 수치를 갱신하고 있다. 이때, 에폭을 늘려 학습을 더 진행하면 더 높은 정확도를 얻을 가능성이 있다. 이에 다음 테스트는 `batch_size=64` 에서 에폭을 더 늘려보았다.
- 의문: 최종 검증정확도는 `batch_size=64` 가 더 높지만 최고 정확도는 `batch_size=50` 가 더 높은데 `batch_size=50` 이 더 좋은게 아닌가?

  → 모델을 추론용으로 저장할때는 마지막 에폭 결과가 기준이 된다

  - `batch=50`은 **18ep에서 최고 성능을 냈지만**,

    → 그때 저장하지 않았다면,

    → 마지막 결과(20ep)는 그보다 낮아졌기 때문에 **최종 제출 모델 성능은 낮음**
  - `batch=64`는 **마지막 에폭에서 최고치를 갱신 중**이기 때문에

    → 현재 결과가 **실제 추론 성능에도 가장 잘 반영됨**

  → `batch=64` 는 아직도 리즈가 아니다

  - `batch=64` 의 마지막 20번째 에폭에서도 최고치를 갱신하고 있었다. 에폭을 늘려서 학습이 더 진행되면 더 높은 정확도를 얻을 수 있다.
- 의문2: `batch=50` 은 일반화 성능이 좋고, 빠르게 수렴한다는 장점이 있었는데, 모델은 어디에 더 중점을 둬 비교해야 하는걸까? 높은 정확도? 좋은 일반화? 빠르게 수렴?
    - 가장 일반적인 기준: 일반화 성능
        - 딥러닝에서 중요한건 새로운 데이터(검증 데이터)에 잘 작동하는기? (일반화)
        - 훈련 정확도가 높아도 검증 데이터에 대한 성능이 낮다면 그건 실패한 모델
            - 딥러닝에서 학습의 목적: **새로운 데이터에 대해 더 정확한 판단**을 내리기 위해 학습 단계에서 데이터에 대해 학습해서 **최적의 파라미터를 찾는것**. (공부, 모의고사)
            - 딥러닝에서 **검증이란?**: 학습단계에서 찾은 최적의 파라미터를 적용한 모델로 **새로운 데이터에 대한 판단**을 하는것 (최종 시험, 수능) = 일반화
            - 검증이 잘 되는가 = 일반화 성능이 좋은가 = 학습한 뒤 새로운 데이터가 왔을때 잘 판단을 하는가
            - 일반화→ 학습단계에서 학습한 파라미터가 새로운데이터에서도 좋은 성능을 내는가? 새로운데이터에도 적용할수있는 일반적인 파라미터를 찾는것
        - 그래서 일반화 성능을 높이려면?

- 높은 정확도만 찾는게 능사는 아니다
- 과적합을 피해야 한다
    - 과적합(overfitting): 데이터를 과하게 학습한것, **학습 데이터에 너무 과하게 적응**, 최적화 된것 → 모델이 overfitting 되면 **학습정확도가 매우 높게 나오지만 검증 정확도는 낮게 나온다.** 즉 범용성이 없는 모델이다(일반화성능 👎)→ 학습 데이터에대해서만 적용할수있는 수준으로 파라미터가 학습된것, **학습데이터에만 사용할수있는 파라미타** → 새로운데이터, **검증용 데이터에는 적용이 어려움** → **일반화 성능** 👎
    - 과적합: 학습데이터에 너무 오버 피팅된것, 학습한 파라미터가 **학습데이터에 너무 과하게 맞춰짐** → 그래서 일반화 성능을 높이기 위해 과적합을 피하는것이다.
    - 비유: 학생이 교과서 문제만 **너무 달달 외우고**, 시험에서 새로운 유형의 문제가 나왔을때 **응용을 하지못하는 상황**
    - 과적합인지 확인하는 법:
        - 훈련 정확도는 상승, 검증 정확도는 감소하거나 어느 순간 정체
            - 훈련데이터는 암기, 새로운 데이터(검증용)에는 일반화하지 못하는 상태
        - 훈련 손실(`loss`)은 감소, 검증 손실(`val_loss`)은 최소를 찍었다가 다시 증가하는 경우
            - 과적합의 대표적인 징후, 정확도 상승과 같은 맥락 → `val_loss`가 다시 오르기 시작하는 시점이 `EarlyStopping` 기준으로 적합
        - 훈련 정확도와 검증 정확도의 차이가 너무 벌어지는 경우

Epoch	accuracy	val_accuracy
5	0.80	0.78 → 정상적 차이
15	0.97	0.75 → ❗ 과적합 가능성 매우 큼

            - 일반적으로 15~20%이면 과적합 가능성
    - 과적합을 막는 방법:

방법	설명
EarlyStopping	검증 성능이 더 이상 좋아지지 않으면 학습 중단
Dropout	일부 뉴런을 랜덤하게 꺼서 과한 학습 방지

정규화 (L2 등)	가중치가 너무 커지지 않도록 페널티 부여
데이터 증강	다양한 데이터로 학습시켜 일반화 능력 강화
모델 단순화	너무 깊거나 복잡한 모델을 줄임

- 최고 정확도보다 중요한건 일관된 성능
  - 한번 최고점을 찍는 것보다 그 정확도를 일관되게 유지하는게 더 실전적이다
    - 현실에선 Epoch 18에서 78%, Epoch 20에서 72%면 최종 에폭로그를 결과로 내보내므로

      → **실제 사용할 땐 72%가 결과**입니다.
    - 따라서 **지속적으로 높은 정확도를 유지하는 모델이 더 좋습니다.**
- 빠른 수렴은 좋은 모델의 지표
  - 빠르게 수렴하는 건 좋지만 일반화 성능을 해치지 않는 선이여야한다
    - 수렴한다?, 일반화 성능을 해치지 않는 선은?
    -

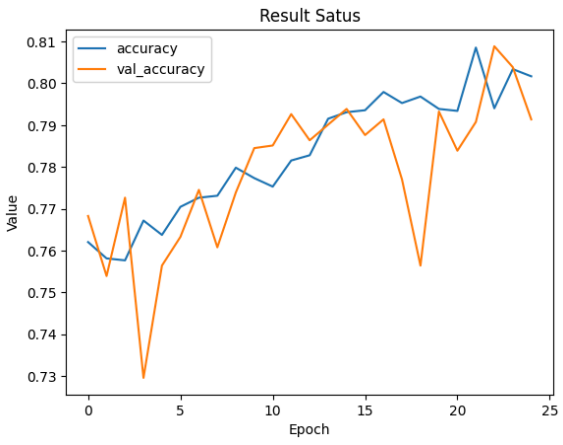
    예를 들어 5ep 만에 75%에 도달했지만, 10ep부터 과적합된다면?

    → 조기 종료(EarlyStopping)가 필요하고

    → 빠른 수렴은 **과적합 가능성을 함께 동반**할 수도 있어요.

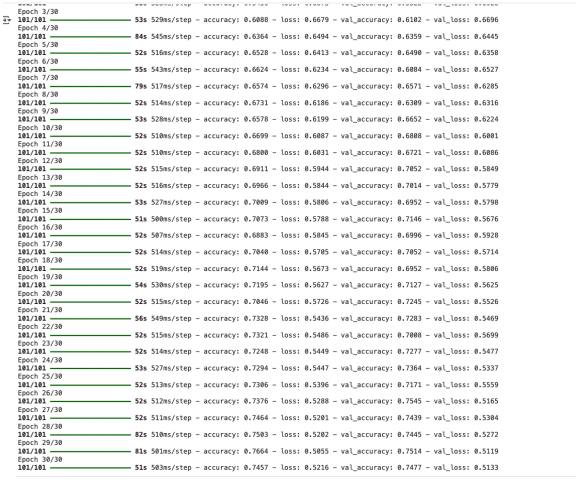
배치 64 에폭 25

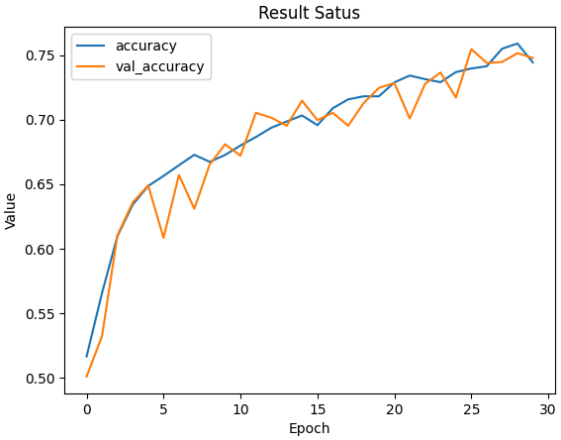
```
Epoch 1/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 180ms/step - accuracy: 0.7610 - loss: 0.4884 - val_accuracy: 0.7683 - val_loss: 0.4784
Epoch 2/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 185ms/step - accuracy: 0.7607 - loss: 0.4840 - val_accuracy: 0.7539 - val_loss: 0.5073
Epoch 3/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 179ms/step - accuracy: 0.7491 - loss: 0.4977 - val_accuracy: 0.7726 - val_loss: 0.4733
Epoch 4/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 184ms/step - accuracy: 0.7702 - loss: 0.4790 - val_accuracy: 0.7295 - val_loss: 0.5540
Epoch 5/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7604 - loss: 0.4962 - val_accuracy: 0.7564 - val_loss: 0.5042
Epoch 6/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 183ms/step - accuracy: 0.7634 - loss: 0.4888 - val_accuracy: 0.7633 - val_loss: 0.4815
Epoch 7/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7711 - loss: 0.4615 - val_accuracy: 0.7745 - val_loss: 0.4842
Epoch 8/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 184ms/step - accuracy: 0.7682 - loss: 0.4754 - val_accuracy: 0.7608 - val_loss: 0.4966
Epoch 9/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7724 - loss: 0.4669 - val_accuracy: 0.7739 - val_loss: 0.4744
Epoch 10/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 185ms/step - accuracy: 0.7786 - loss: 0.4606 - val_accuracy: 0.7845 - val_loss: 0.4674
Epoch 11/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 176ms/step - accuracy: 0.7721 - loss: 0.4628 - val_accuracy: 0.7851 - val_loss: 0.4552
Epoch 12/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 183ms/step - accuracy: 0.7762 - loss: 0.4579 - val_accuracy: 0.7926 - val_loss: 0.4576
Epoch 13/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 176ms/step - accuracy: 0.7818 - loss: 0.4497 - val_accuracy: 0.7864 - val_loss: 0.4530
Epoch 14/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 183ms/step - accuracy: 0.7942 - loss: 0.4506 - val_accuracy: 0.7901 - val_loss: 0.4570
Epoch 15/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 175ms/step - accuracy: 0.7955 - loss: 0.4398 - val_accuracy: 0.7939 - val_loss: 0.4463
Epoch 16/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 183ms/step - accuracy: 0.7992 - loss: 0.4352 - val_accuracy: 0.7876 - val_loss: 0.4598
Epoch 17/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 176ms/step - accuracy: 0.8026 - loss: 0.4285 - val_accuracy: 0.7914 - val_loss: 0.4450
Epoch 18/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 183ms/step - accuracy: 0.7990 - loss: 0.4314 - val_accuracy: 0.7770 - val_loss: 0.4780
Epoch 19/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7983 - loss: 0.4360 - val_accuracy: 0.7564 - val_loss: 0.4951
Epoch 20/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 184ms/step - accuracy: 0.7870 - loss: 0.4369 - val_accuracy: 0.7933 - val_loss: 0.4452
Epoch 21/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.7976 - loss: 0.4367 - val_accuracy: 0.7839 - val_loss: 0.4494
Epoch 22/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 19s 183ms/step - accuracy: 0.8115 - loss: 0.4159 - val_accuracy: 0.7908 - val_loss: 0.4401
Epoch 23/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 20s 176ms/step - accuracy: 0.7949 - loss: 0.4221 - val_accuracy: 0.8089 - val_loss: 0.4270
Epoch 24/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 181ms/step - accuracy: 0.8123 - loss: 0.4051 - val_accuracy: 0.8039 - val_loss: 0.4335
Epoch 25/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 18s 177ms/step - accuracy: 0.8088 - loss: 0.4129 - val_accuracy: 0.7914 - val_loss: 0.4379
```



결과: 학습정확도 80, 검증 정확도 79 , 최고치 검증정확도: 80

배치 64 에폭 30





결과: 학습정확도: 74, 검증 정확도 74, 최고치 검증정확도: 75

에폭 늘릴수록 정확도 올라갔지만 소요되는 시간이 너무 길어진다. 64에 30으로 했을때 한 에폭당 약 1분정도씩 걸리면서 총 30분이상의 시간이 소요되었다.

에폭 25VS 30 비교

항목	Epoch 25	Epoch 30
최고 val_accuracy	**0.8039** (ep24)	0.7477 (ep30)
최저 val_loss	0.4270 (ep23)	0.5133 (ep30)
걸린 시간	약 **25분**	약 **30분 이상**
성능 향상	수렴 및 최고 성능	성능 오히려 감소 (과적합 가능성)

- 에폭 증가가 항상 성능 향상으로 이어지지는 않았다
  - 에폭 25에서는 정확도, 손실 모두 최고점을 기록했다
  - 에폭 30에서는 검증 정확도가 하락하고, 검증 손실은 증가했다.
- 시간 대비 효율 약화
  - 에폭 25는 약 25분, 에폭 30은 약 30분 소요되었다

    → 5 에폭 더, 5분 더 학습 했지만 성능은 오히려 떨어짐

에폭이 커도 좋은건 아니다. 소요되는 시간이 더 길어진다, 일정 수준 이상부터 더 늘어나지 않거나 오히려 더 낮은 에폭으로 더 높은 정확도를 가질수 있다.

에폭 결론

- 에폭이 커도 항상 좋은 결과가 보장되는 것은 아니다.
- 일정수준 이상에서는 성능이 더 좋아지지 않거나 오히려 하락할 수도 있다.
- 학습 시간+ 자원 낭비 고려하면 효율적인 에폭 수 선택이 중요하다

일정 에폭 수 이상 학습은 모델성능 향상보다는 과적합 위험 증가와 시간믄 더 소모 될수 있다고 보여준다.

따라서 이 과제에선 에폭 25 부근이 최적의 학습 종료 시점으로 판단된다. 여기에 `EarlyStopping`이나 `ModelCheckpoint`를 활용하여 효율적인 학습 종료를 설정하는 것이 바람직하다.

학습 종료 설정

추가 고려 사항

정확도 향상을 위해 또 고려해볼만한 방법은 활성화 함수, 손실함수, 최적화 기술, 모델 변경 등이 있다.

활성화 함수

활성화함수는 성능 향상과 과적합 방지를 위해 LeakyRelu 적용도 좋은 방법이다

손실함수는 이 과제의 입력데이터에서 선택가능한 class는 고양이, 강아지 두개 뿐이므로 현재 손실함수인 binary_corrossEntropy가 적절하다

최적화 기술은 현재 Adom이지만 L2가 적용되면 더 좋은 성능을 낼수 있다. Adom을 적용하고 L2를 적용하는것도 좋지만 둘을 합친 기술도 좋다.

모델변경: 현 과제의 입력데이터 특성에 의해 VGG에 비해 MobileNet이 속도와 성능 측면에서 더 유리하다. 가볍게 MobileNet

3차: 모델 수정 및 변경

VGG

```
# 데이터 증강기 + 검증 세트 분리
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)

# 학습용 제너레이터
train_generator = train_datagen.flow_from_directory(
    '/content/data/',
    target_size=(64, 64),
    batch_size=64,
    class_mode='binary',
    subset='training'
)

# 검증용 제너레이터
validation_generator = train_datagen.flow_from_directory(
    '/content/data/',
    target_size=(64, 64),
    batch_size=64,
    class_mode='binary',
    subset='validation'
)
#여기까지는 변경점 X
------------------------------------------------------------
from tensorflow.keras.applications import VGG16
```

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D,

# VGG16 기본 모델 불러오기 (64x64에 맞추기 위해 include_top=False 설정)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(6
base_model.trainable = False  # 기본 학습된 가중치는 고정 (전이학습)

# 위에 새 출력층 붙이기
model = tf.keras.Sequential([
    base_model,
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

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

```
Epoch 1/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 422s 4s/step - accuracy: 0.7772 - loss: 0.4635 - val_accuracy: 0.7664 - val_loss: 0.4718
Epoch 2/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 412s 4s/step - accuracy: 0.7850 - loss: 0.4590 - val_accuracy: 0.7545 - val_loss: 0.4832
Epoch 3/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 411s 4s/step - accuracy: 0.7964 - loss: 0.4373 - val_accuracy: 0.7570 - val_loss: 0.4818
Epoch 4/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 410s 4s/step - accuracy: 0.7830 - loss: 0.4523 - val_accuracy: 0.7839 - val_loss: 0.4655
Epoch 5/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 409s 4s/step - accuracy: 0.7961 - loss: 0.4258 - val_accuracy: 0.7826 - val_loss: 0.4749
Epoch 6/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 407s 4s/step - accuracy: 0.7838 - loss: 0.4468 - val_accuracy: 0.7770 - val_loss: 0.4589
Epoch 7/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 409s 4s/step - accuracy: 0.8030 - loss: 0.4235 - val_accuracy: 0.7614 - val_loss: 0.4823
Epoch 8/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 405s 4s/step - accuracy: 0.7911 - loss: 0.4357 - val_accuracy: 0.7552 - val_loss: 0.4853
Epoch 9/25
 17/101 ━━━━━━━━━━━━━━━━━━━━ 4:30 3s/step - accuracy: 0.7835 - loss: 0.4650
```

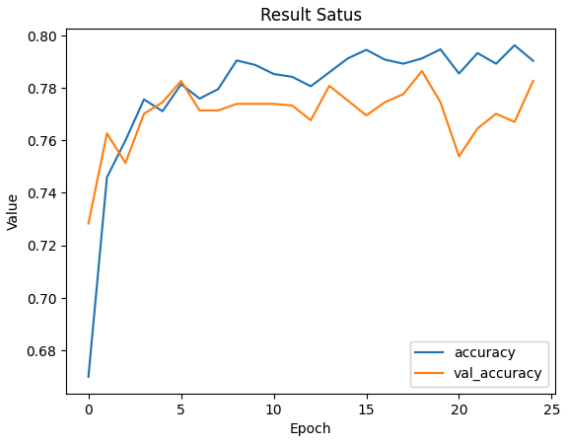
VGG는 초반 부터 검증정확도가 76%가 나왔지만 한 에폭이 무려 약 400초가량 걸렸다. VGG 모델 자체가 깊고 큰 층과 매우 많은 파라미터를 가지고 있어서 매우 느린 속도를 보이고 있다. 정답 클래스가 2개뿐인 비교적 단순한 데이터셋에서는 너무 과도한 모델인것이다.

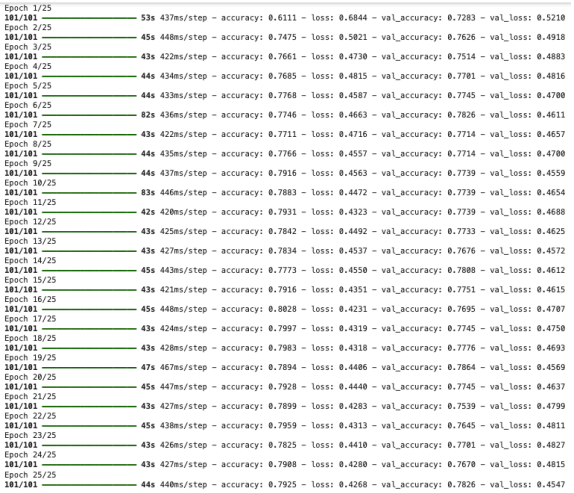
MobileNet

```
# 데이터 증강기 + 검증 세트 분리
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)

# 학습용 제너레이터
train_generator = train_datagen.flow_from_directory(
    '/content/data/',
    target_size=(64, 64),
    batch_size=64,
    class_mode='binary',
    subset='training'
)

# 검증용 제너레이터
validation_generator = train_datagen.flow_from_directory(
    '/content/data/',
    target_size=(64, 64),
    batch_size=64,
    class_mode='binary',
    subset='validation'
)

from tensorflow.keras.applications import MobileNetV2

base_model = MobileNetV2(weights='imagenet', include_top=False, input_sha
base_model.trainable = False

model = tf.keras.Sequential([
    base_model,
    GlobalAveragePooling2D(),
```

```
    Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

MobileNet: 모바일 등 리소스가 제한된 환경에서 효율적인 계산을 위해 설계된 경량 심층 신경망, 기존과는 다른 새로운 아키텍쳐 적용. 채널과 공간 차원의 컨볼루션을 따로 수행하여 합치는 방식





최종 검증정확도 78% 달성하였다. 하지만 이미 중간 에폭부터 78인근을 돌고있어서 이부분에서는 EarlyStopping을 적용해서 더 최적한 모델이 될수 있을것이다.

각 에폭당 43~55초가량으로 초반 모델보다는 늦지만 VGG보다는 훨씬 빠르게 진행되었다. 또한 정확도도 목표치에 많이 근접하는 모습을 보였다.

- 성능 평가

평가 항목	해석
성능	기존 CNN 모델 대비 우수함. 검증 정확도 78.2%로 과제 목표(73%) 초과 달성.
과적합 여부	없음. 학습/검증 정확도 차이 적고, 손실도 동반 감소함.
모델 효율성	MobileNetV2 덕분에 빠른 학습과 가벼운 연산 비용.

VGG와 MobileNet 비교

항목	**VGG16/VGG19**	**MobileNet (v1/v2/v3)**
성능 (정확도)	높음 (기초 튼튼)	매우 높음 (효율까지 고려)
모델 크기	크고 무거움	작고 가벼움

추론 속도	느림	빠름
적합한 환경	서버, 성능 우선	실시간/모바일, 경량 모델
전이학습 활용도	★★★★☆	★★★★★

고양이 강아지 이진분류에서 적합성 판단

요소	왜 중요?	MobileNet 적합 여부
이진 분류	복잡한 분류 아님 → 작은 모델도 충분	👍
저해상도 이미지 (보통 64x64~224×224)	고해상도 아닌 경우 MobileNet 충분	👍
실습 목적 (과제, 배포 등)	빠른 학습/추론이 유리	👍
학습 자원 제한 (랩탑, Colab)	GPU 메모리 적으면 VGG는 버겁다	👍

성능 기준으로도 MobileNet이 경쟁력이 있다.

모델	파라미터 수	Top-1 정확도 (ImageNet 기준)
VGG16	138M	71.5%
MobileNetV2	**3.4M**	**71.8%**
MobileNetV3-Small	**2.5M**	약 67.4% (최적화됨)

MobileNet은 **성능은 거의 동일하면서도, 크기는 1/40 수준**

옵티마이저

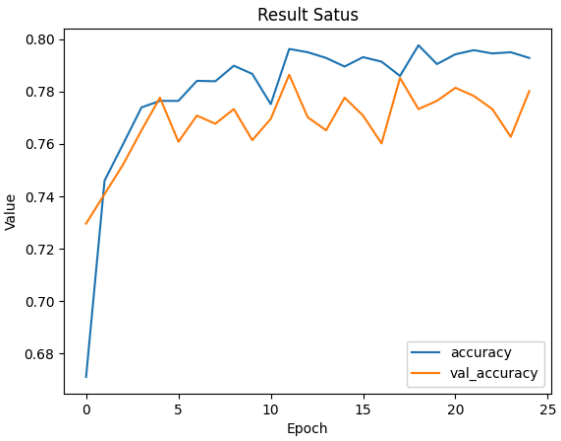
```
# 데이터 증강기 + 검증 세트 분리
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)
```

```
# 학습용 제너레이터
train_generator = train_datagen.flow_from_directory(
    '/content/data/',
    target_size=(64, 64),
    batch_size=64,
    class_mode='binary',
    subset='training'
)

# 검증용 제너레이터
validation_generator = train_datagen.flow_from_directory(
    '/content/data/',
    target_size=(64, 64),
    batch_size=64,
    class_mode='binary',
    subset='validation'
)

from tensorflow.keras.applications import MobileNetV2

base_model = MobileNetV2(weights='imagenet', include_top=False, input_sha
base_model.trainable = False

model = tf.keras.Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(1, activation='sigmoid')
])
from tensorflow.keras.optimizers import AdamW

# 모델 컴파일 시 AdamW 적용
model.compile(
    optimizer=AdamW(learning_rate=0.001, weight_decay=1e-4),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

아담에 L2 정규화를 같이 적용하는 AdamW

- 결과

```
Epoch 1/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 55s 481ms/step - accuracy: 0.6148 - loss: 0.6755 - val_accuracy: 0.7295 - val_loss: 0.5425
Epoch 2/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 438ms/step - accuracy: 0.7422 - loss: 0.5111 - val_accuracy: 0.7408 - val_loss: 0.5047
Epoch 3/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 424ms/step - accuracy: 0.7626 - loss: 0.4866 - val_accuracy: 0.7520 - val_loss: 0.4985
Epoch 4/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 46s 453ms/step - accuracy: 0.7746 - loss: 0.4737 - val_accuracy: 0.7651 - val_loss: 0.4841
Epoch 5/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 425ms/step - accuracy: 0.7685 - loss: 0.4675 - val_accuracy: 0.7776 - val_loss: 0.4689
Epoch 6/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 45s 446ms/step - accuracy: 0.7825 - loss: 0.4429 - val_accuracy: 0.7608 - val_loss: 0.4744
Epoch 7/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 45s 441ms/step - accuracy: 0.7863 - loss: 0.4442 - val_accuracy: 0.7708 - val_loss: 0.4613
Epoch 8/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 425ms/step - accuracy: 0.7931 - loss: 0.4480 - val_accuracy: 0.7676 - val_loss: 0.4669
Epoch 9/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 429ms/step - accuracy: 0.7910 - loss: 0.4330 - val_accuracy: 0.7733 - val_loss: 0.4544
Epoch 10/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 46s 456ms/step - accuracy: 0.7839 - loss: 0.4401 - val_accuracy: 0.7614 - val_loss: 0.4786
Epoch 11/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 427ms/step - accuracy: 0.7796 - loss: 0.4448 - val_accuracy: 0.7695 - val_loss: 0.4618
Epoch 12/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 439ms/step - accuracy: 0.8058 - loss: 0.4219 - val_accuracy: 0.7864 - val_loss: 0.4526
Epoch 13/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 45s 444ms/step - accuracy: 0.7937 - loss: 0.4451 - val_accuracy: 0.7701 - val_loss: 0.4674
Epoch 14/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 428ms/step - accuracy: 0.7908 - loss: 0.4357 - val_accuracy: 0.7651 - val_loss: 0.4706
Epoch 15/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 428ms/step - accuracy: 0.7870 - loss: 0.4450 - val_accuracy: 0.7776 - val_loss: 0.4557
Epoch 16/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 434ms/step - accuracy: 0.7966 - loss: 0.4298 - val_accuracy: 0.7708 - val_loss: 0.4692
Epoch 17/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 423ms/step - accuracy: 0.7908 - loss: 0.4306 - val_accuracy: 0.7601 - val_loss: 0.4985
Epoch 18/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 439ms/step - accuracy: 0.7856 - loss: 0.4425 - val_accuracy: 0.7851 - val_loss: 0.4477
Epoch 19/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 45s 443ms/step - accuracy: 0.7919 - loss: 0.4368 - val_accuracy: 0.7733 - val_loss: 0.4600
Epoch 20/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 438ms/step - accuracy: 0.7878 - loss: 0.4350 - val_accuracy: 0.7764 - val_loss: 0.4691
Epoch 21/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 430ms/step - accuracy: 0.7971 - loss: 0.4278 - val_accuracy: 0.7814 - val_loss: 0.4662
Epoch 22/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 45s 448ms/step - accuracy: 0.7896 - loss: 0.4307 - val_accuracy: 0.7783 - val_loss: 0.4752
Epoch 23/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 81s 443ms/step - accuracy: 0.7935 - loss: 0.4290 - val_accuracy: 0.7733 - val_loss: 0.4737
Epoch 24/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 43s 425ms/step - accuracy: 0.7934 - loss: 0.4400 - val_accuracy: 0.7626 - val_loss: 0.4678
Epoch 25/25
101/101 ━━━━━━━━━━━━━━━━━━━━ 44s 438ms/step - accuracy: 0.7979 - loss: 0.4293 - val_accuracy: 0.7801 - val_loss: 0.4716
```



최종 검증정확도: 78%

- 성능 평가

항목	평가
정확도	과제 기준(73%)을 넘는 우수한 결과 (최대 78.6%)
일반화	val_accuracy가 꾸준히 유지됨 → **일반화 성능 우수**
과적합 여부	없음. train-val 간 격차가 좁고, val_loss 증가 없음
최적 epoch	Epoch 12~14에서 val_accuracy 최고치, EarlyStopping 추천
속도 대비 효율	25 epoch 내에서 최대 성능 확보 → **시간 투자 대비 효율 좋음**

L2 정규화란?

L2 정규화는 **가중치(Weight)가 너무 커지지 않게 제어하는 규제 방식**

왜 필요할까?

- 딥러닝은 **학습 데이터에 과도하게 적응**해서 과적합 가능성
- 과적합이 일어나면 훈련 데이터에선 잘 맞지만, 새로운 데이터에선 성능이 저하
- **L2 정규화는 너무 큰 가중치에 벌을 주면서 과도한 학습을 막는다**

AdamW란?

- `AdamW`는 `Adam` 최적화 알고리즘에 L2 정규화(weight decay)를 제대로 적용한 버전

왜 AdamW인가?

- 기존의 Adam은 내부적으로 모멘텀(Momentum)이 있어서 weight decay가 **정확히 반영되지 않는다**
- 그래서 weight decay가 **실제로는 효과가 덜함**.
- Adam + kernel_regularizer=L2() 방식은 엄밀한 weight decay가 아님.

AdamW는 뭐가 다른가?

- **가중치 업데이트와 weight decay를 분리**
- 논문적으로도 검증된 방법:

  "Decoupled Weight Decay Regularization" (2017)

느낀점

이번 이미지분류과제를 통해 딥러닝, 특히 CNN 모델이 어떻게 작동하는지, 어떤 요소들이 정확도에 영향을 주는지 직접 확인할수 있었다.

에폭을 무조건 많이 주는 것이 좋은 것이 아니라는 것도 이번 실험에서 중요한 교훈이었다. 에폭 25와 30을 비교했을 때, 오히려 에폭 25일 때 정확도가 더 높았고, 검증 손실도 낮았다. 이 과정에서 **과적합**이라는 개념과 그 위험성도 확실히 이해할 수 있었다.

더 높은 정확도 뿐 아니라 과제에 더 적합하고 좋은 성능을 내기위해 여러가지를 고려하게 되면서 딥러닝 수업에서 이론으로 들었던 내용들을 실제 데이터를 확인하고, 더 다양한 기법들을 찾아보게되었다. 또한 각 개념들을 리마인드하면서 다시 정리할수 있었던 매우 의미있는 시간이었다. 앞으로 AI 관련 프로젝트를 진행하면서 이 과제를 통해 얻은 경험을 통해 더 체계적으로 접근할 수 있을 것이다.

다만 100%에 가깝게 정확도를 더 높이고 싶었던 목표까지는 도달하지 못해서 아쉬웠다. 과제에서는 이정도 정확도로 마무리하지만 추가적인 공부와 테스트로 정확도를 더 올리자는 목표를 세웠다.