

컴퓨터비전 과제 2

#1. ML 기반의 숫자 인식

- **hw2_1.py** : Case 1(hw2_1-1.py), Case 2(hw2_1-2.py), Case 3(hw2_1-3.py)의 소스코드를 하나로 hw2_1.py로 통합하였습니다.
python3 hw2_1.py 명령어로 실행합니다. 실행 시 Case 1이 실행되도록 하고, 나머지 두 케이스는 주석처리해두었습니다.

```
# ... 중략 ...

if __name__ == "__main__":
    KNN_with_HOG()      # Case 1
    # SVM_with_HOG()    # Case 2
    # SVM_with_SIFT()   # Case 3
```

Case 1: kNN + HOG 사용 결과

1. 실행 결과 :

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_1-1.py
Number of training images: 60000
Number of test images: 10000
Extracting HOG features for training set...
Training feature extraction time: 0.72s
Extracting HOG features for test set...
Test feature extraction time: 0.12s
Training kNN...
Training time: 0.02s
Testing kNN...
Test inference time: 11.21s
Test Accuracy: 98.09%
Visualizing predictions for 10 random test samples...
```

(중략)

Training kNN...

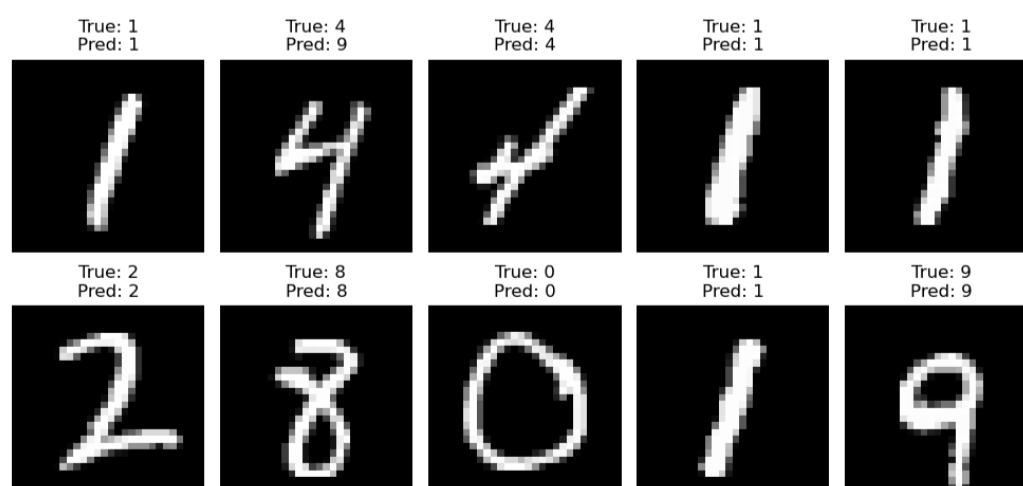
Training time: 0.02s

Testing kNN...

Test inference time: 11.21s

Test Accuracy: 98.09%

- 테스트 데이터셋에서 랜덤하게 10개 샘플을 추출하고, KNN 알고리즘으로 훈련된 모델이 예측한 값과 실제 레이블 정보를 시각화해본 결과



- 다섯 번을 랜덤하게 돌려봤는데, 네 번은 10개 모두 맞쳤으며 나머지 한 번의 실행결과(위)에서, 두 번째 샘플에서 예측이 실패했음을 확인할 수 있었습니다.

2. 결과해석 및 원인분석 :

Test Accuracy는 약 98%로, kNN 알고리즘과 HOG 기술자를 사용하여 만든 모델이 MNIST 데이터셋에서 높은 정확도로 숫자를 검출할 수 있음을 확인했습니다.

Training Time은 약 0.02초가 나왔습니다. kNN 알고리즘은 training 단계에서 단순히 데이터를 저장하므로 매우 짧게 소요되었습니다.

Test Inference Time은 약 11초로 뒤에 있을 SVM을 사용했을 때보다 상대적으로 오래 걸렸는데, 이는 kNN의 단점인 train 데이터가 많을수록 테스트 시간이 길어진다는 이유 때문인 것 같습니다. (실행 시마다 시간의 변동 폭이 조금 심합니다.)

마지막으로 랜덤하게 10개의 샘플을 추출한 뒤 예측결과를 확인하는 작업을 해보았는데, 모델이 대부분 예측을 잘 수행하였음을 확인할 수 있었습니다.

Case 2 : SVM + HOG 사용 결과

1. 실행 결과 :

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_1-2.py
Number of training images: 60000
Number of test images: 10000
Extracting HOG features for training set...
Training feature extraction time: 0.72s
Extracting HOG features for test set...
Test feature extraction time: 0.12s
Training SVM...
Training time: 71.66s
Testing SVM...
Test inference time: 3.05s
Test Accuracy: 98.86%
Visualizing predictions for 10 random test samples...
```

(중략)

Training SVM...

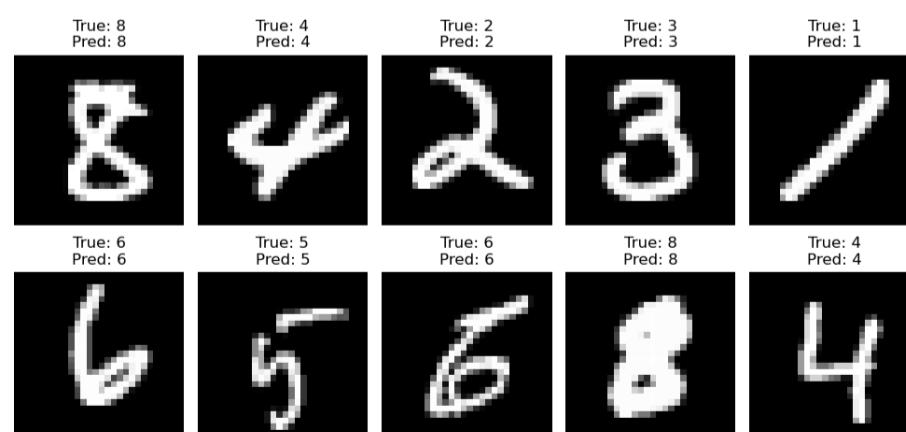
Training time: 71.66s

Testing SVM...

Test inference time: 3.05s

Test Accuracy: 98.86%

테스트 데이터셋에서 랜덤하게 10개 샘플을 추출하고, SVM 알고리즘으로 훈련된 모델이 예측한 값과 실제 레이블 정보를 시각화해본 결과



- 다섯 번을 랜덤하게 돌려봤는데, 모두 예측에 성공했음을 확인할 수 있었습니다.

2. 결과해석 및 원인분석 :

Test Accuracy는 약 99%로, SVM 알고리즘과 HOG 기술자를 사용하여 만든 모델이 MNIST 데이터셋에서 kNN 사용 시보다 높은 정확도로 숫자를 검출할 수 있음을 확인했습니다.

Training Time은 약 72초가 나왔습니다. SVM 알고리즘은 하기한 이유로 kNN 알고리즘보다 시간이 훨씬 오래 소요됩니다.

(a) SVM은 훈련 데이터의 서포트 벡터(초평면을 찾기 위한 데이터포인트)를 찾기 위해 이차 계획법 문제를 해결합니다. 데이터가 많아질수록 이 최적화 과정이 계산적으로 더 복잡해집니다.

(b) SVM은 커널 트릭 기법을 활용하여 최적의 결정 경계(초평면)를 학습합니다. 커널 트릭은 데이터를 고차원 공간으로 매핑하여 선형적으로 분리가능한 형태로 변환하는 기법입니다. 훈련 과정에서 각 데이터 샘플 간의 거리를 계산하는 작업이 필요한데, 이 작업은 훈련 데이터가 많을 수록 기하급수적으로 증가합니다.

이에 비해 kNN 알고리즘은 단순히 훈련 데이터를 저장하기만 하면 될 정도로 훈련과정이 거의 없다고 볼 수 있기 때문에 **훈련 시간이 매우 짧게 소요됩니다**.

Test Inference Time은 약 3초로 앞에서의 kNN을 사용했을 때보다 상대적으로 빨랐습니다. **SVM은 training 과정에서 결정 경계를 한 번 계산하면 이후 이를 효율적으로 사용**하여 kNN보다 대체로 빠른 추론속도를 보장할 수 있습니다.

이에 비해 kNN 알고리즘은 테스트 샘플이 들어오면 training 데이터의 모든 샘플과 거리를 계산합니다. 데이터가 n 개라면 테스트 샘플 하나에 대한 n 번의 거리 계산이 필요하여 training 데이터가 많을수록 Inference Time이 크게 증가합니다. 거리계산 후, k 개의 최근접 이웃을 찾기 위해 훈련 데이터를 정렬하는 과정 또한 계산량을 증가시킵니다.

마찬가지로 랜덤하게 10개의 샘플을 추출한 뒤 예측결과를 확인하는 작업에서, 역시나 모델이 대부분 예측을 잘 수행하였음을 확인할 수 있었습니다. 다섯 번을 랜덤하게 돌려보았는데, $5 \times 10 = 50$ 번 모두 정확히 예측에 성공하였습니다.

Case 3 : SVM + SIFT 사용 결과

1. 실행 결과 :

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_1-3.py
Number of training images: 60000
Number of test images: 10000
Extracting SIFT features for training set...
Training feature extraction time: 17.90s
Extracting SIFT features for test set...
Test feature extraction time: 3.00s
Training SVM...
Training time: 48.54s
Testing SVM...
Test inference time: 1.13s
Test Accuracy: 11.58%
Visualizing predictions for 10 random test samples...
```

(중략)

Training SVM...

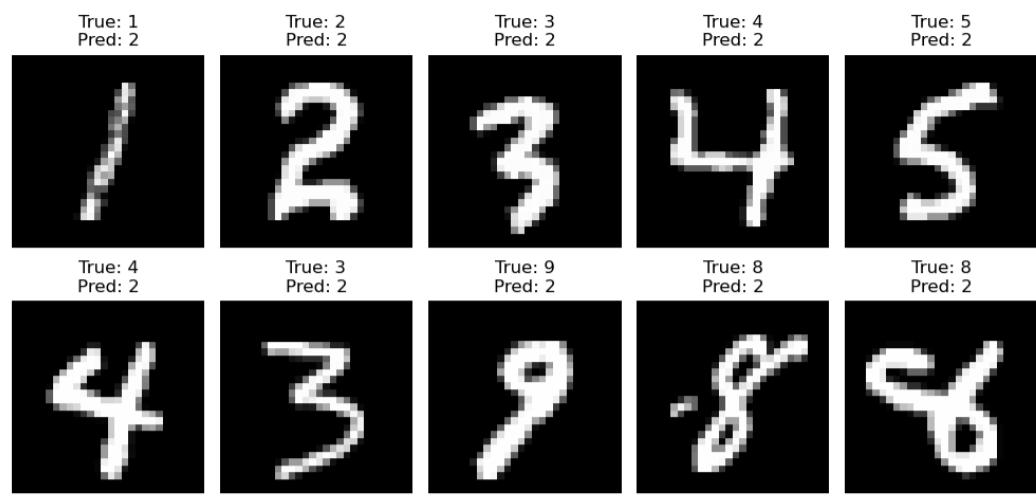
Training time: 48.54s

Testing SVM...

Test inference time: 1.13s

Test Accuracy: 11.58%

테스트 데이터셋에서 랜덤하게 10개 샘플을 추출하고, SVM 알고리즘으로 훈련된 모델이 예측한 값과 실제 레이블 정보를 시각화해본 결과



- 두 번째(2)를 제외하고는 전부 예측에 실패한 결과를 확인할 수 있었습니다.

2. 결과해석 및 원인분석 :

Test Accuracy는 약 11.5%로, 랜덤 추측과 유사한 수준으로 매우 떨어졌음을 확인할 수 있었습니다. 또한 랜덤하게 10개의 샘플을 추출한 뒤 예측결과를 확인해보니, 모델이 대부분 예측을 잘 수행하지 못했음을 확인할 수 있었습니다.

SVM의 특징인 긴 Training Time, 짧은 Test Inference Time은 그대로였기 때문에 SVM에 대한 설명은 생략하고 SIFT 사용 시의 Test Accuracy 감소 요인에 대해 분석해보려 합니다.

성능 저하의 원인은, HOG 대신 SIFT 기술자를 사용했기 때문으로 추측했습니다.

SIFT는 수업에서 배웠다시피 일반적으로 자연 이미지에서 유용하며, 건물 / 사물 / 풍경 등에서 강력한 특징추출이 가능합니다. 하지만 **MNIST 데이터셋은 매우 단순한 숫자 이미지로 국소적으로 강렬한 특징점이 드물** 것입니다. 또한 숫자 형태의 특징이 자연 이미지에 비해 상대적으로 단조로우며, 회전 / 크기 변환에 크게 의존하지 않습니다.

SIFT는 특징점을 기반으로 계산하기 때문에, 숫자 이미지에서는 의미있는 정보를 제대로 포착하지 못할 것이라고 추론할 수 있습니다.

(이 사실을 기반하여, 일부러 대조를 위해 SIFT를 사용하여 성능저하를 확인해보았습니다.)

따라서 SIFT에서 추출된 특징 벡터가 MNIST 숫자를 잘 표현하지 못하였기 때문에 예상한 대로 SVM 모델은 훈련 과정에서 적합한 결정 경계를 학습하지 못했다고 볼 수 있습니다.

반면 HOG는 이미지의 에지 방향 그래디언트 정보를 히스토그램으로 요약하여 전체적인 형태를 표현하는 방법으로, 패턴 인식에 강하며 숫자와 같은 간단한 형태를 효과적으로 특징화할 수 있습니다. 따라서 HOG에서 추출된 특징 벡터가 MNIST 숫자를 잘 표현할 수 있기 때문에 SVM의 적합한 결정 경계를 통해 유의미한 수준의 Test Accuracy를 달성할 수 있습니다.

#2. Empire State Building 확인(실행결과 캡처)

- 사진에 엠파이어 스테이트 빌딩이 존재하는 케이스



city1.jpg

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_2.py city1.jpg
0: 448x640 1 empire-state-building, 233.1ms
Speed: 6.7ms preprocess, 233.1ms inference, 8.4ms postprocess per image at shape (1, 3, 448, 640)

Empire State Building in Image : True
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

- 사진에 엠파이어 스테이트 빌딩이 존재하지 않는 케이스



city6.jpg

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_2.py city6.jpg
0: 448x640 (no detections), 210.6ms
Speed: 1.9ms preprocess, 210.6ms inference, 0.4ms postprocess per image at shape (1, 3, 448, 640)

Empire State Building in Image : False
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

- 사진에 엠파이어 스테이트 빌딩이 아닌 다른 빌딩이 존재하는 케이스



city20.jpg

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_2.py city20.jpg
0: 640x512 (no detections), 211.0ms
Speed: 4.4ms preprocess, 211.0ms inference, 5.7ms postprocess per image at shape (1, 3, 640, 512)

Empire State Building in Image : False
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

#3. Empire State Building 위치 찾기

I. 위치 찾기 결과 예시

1. city1.jpg(존재)

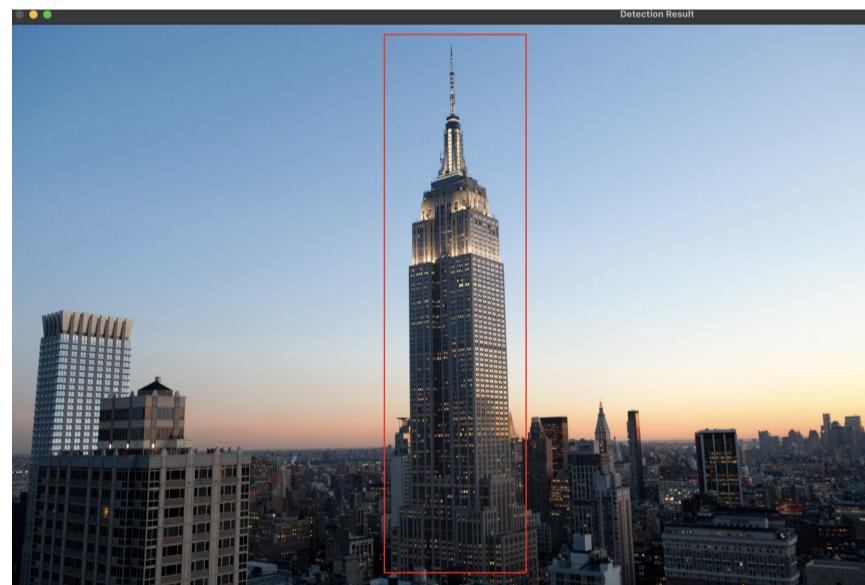
- 원본 이미지



- 실행 터미널

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_3.py city1.jpg
  0: 448x640 1 empire-state-building, 232.1ms
  Speed: 8.5ms preprocess, 232.1ms inference, 8.2ms postprocess per image at shape (1, 3, 448, 640)
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

- 실행 결과



2. city2.jpg(존재)

- 원본 이미지



- 실행 터미널

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_3.py city2.jpg
0: 640x480 1 empire-state-building, 236.8ms
Speed: 4.5ms preprocess, 236.8ms inference, 8.3ms postprocess per image at shape (1, 3, 640, 480)
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

- 실행 결과



3. city6.jpg(엠파이어 스테이트 빌딩 미존재)

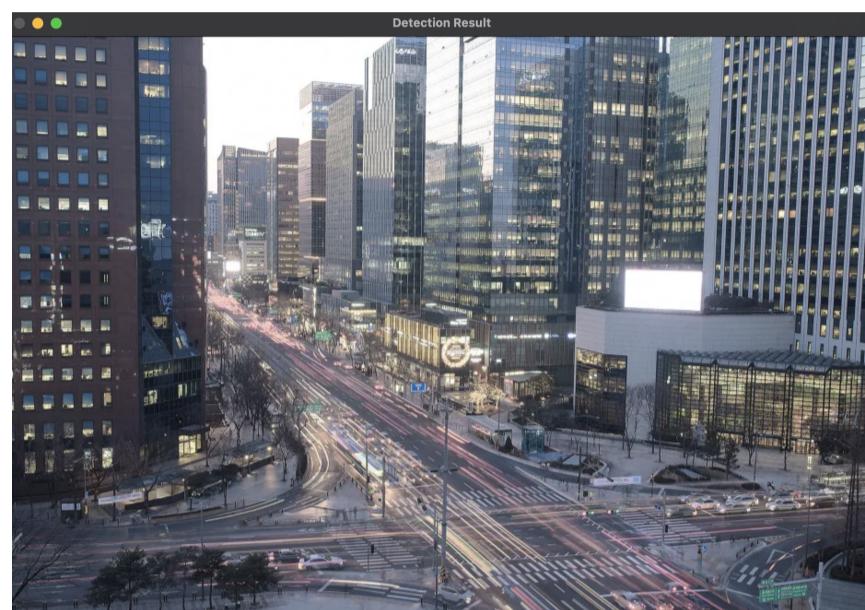
- 원본 이미지



- 실행 터미널

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_3.py city6.jpg
0: 448x640 (no detections), 235.7ms
Speed: 4.1ms preprocess, 235.7ms inference, 5.5ms postprocess per image at shape (1, 3, 448, 640)
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

- 실행 결과(confidence 값이 낮아 필터링되어, 아무 것도 검출하지 않음)



4. city10.jpg(다른 건물인 크라이슬러 빌딩 이미지)

- 원본 이미지



- 실행 터미널

```
● (AI-24) ikjoon@IJui-MacBookAir hw2 % python3 hw2_3.py city10.jpg
0: 640x448 1 CB, 215.5ms
Speed: 4.7ms preprocess, 215.5ms inference, 7.4ms postprocess per image at shape (1, 3, 640, 448)
○ (AI-24) ikjoon@IJui-MacBookAir hw2 %
```

- 실행 결과(confidence 값이 낮아 필터링되어, 아무 것도 검출하지 않음)



II. 사용한 방법에 대한 설명

pretrained YOLOv8(yolov8m)를 베이스 모델로 사용하고 **엠파이어 스테이트 빌딩(다른 클래스의 타 빌딩 포함)** 데이터셋을 이용해 파인튜닝 작업을 진행했습니다. YOLOv8은 객체 검출 분야에서 널리 사용되는 딥러닝 모델로, 빠른 속도와 높은 정확도를 제공합니다. 모델은 이미지 입력 시 Bounding Box(경계 상자)와 클래스 레이블을 출력합니다.

파인튜닝에 사용할 데이터셋은 roboflow에서 제공하는 데이터셋들을 활용하였습니다. 최대한 많고 품질이 좋은 엠파이어 스테이트 빌딩 및 기타 유사한 빌딩들의 데이터셋을 여러 개 확보하였습니다. 각각의 데이터셋 출처는 하기 참고문헌에 명시하였습니다. 이 프로젝트 데이터셋들을 하나의 데이터셋으로 merge하였고 이를 roboflow 개인계정에 업로드하였습니다. 그리고 '파이썬 소스코드로 데이터셋 다운로드(내려받기)받기' 기능을 활용하여 최종 데이터셋을 Google Colab 환경으로 불러들였습니다. 아래는 train.ipynb 소스코드 일부분입니다.

```

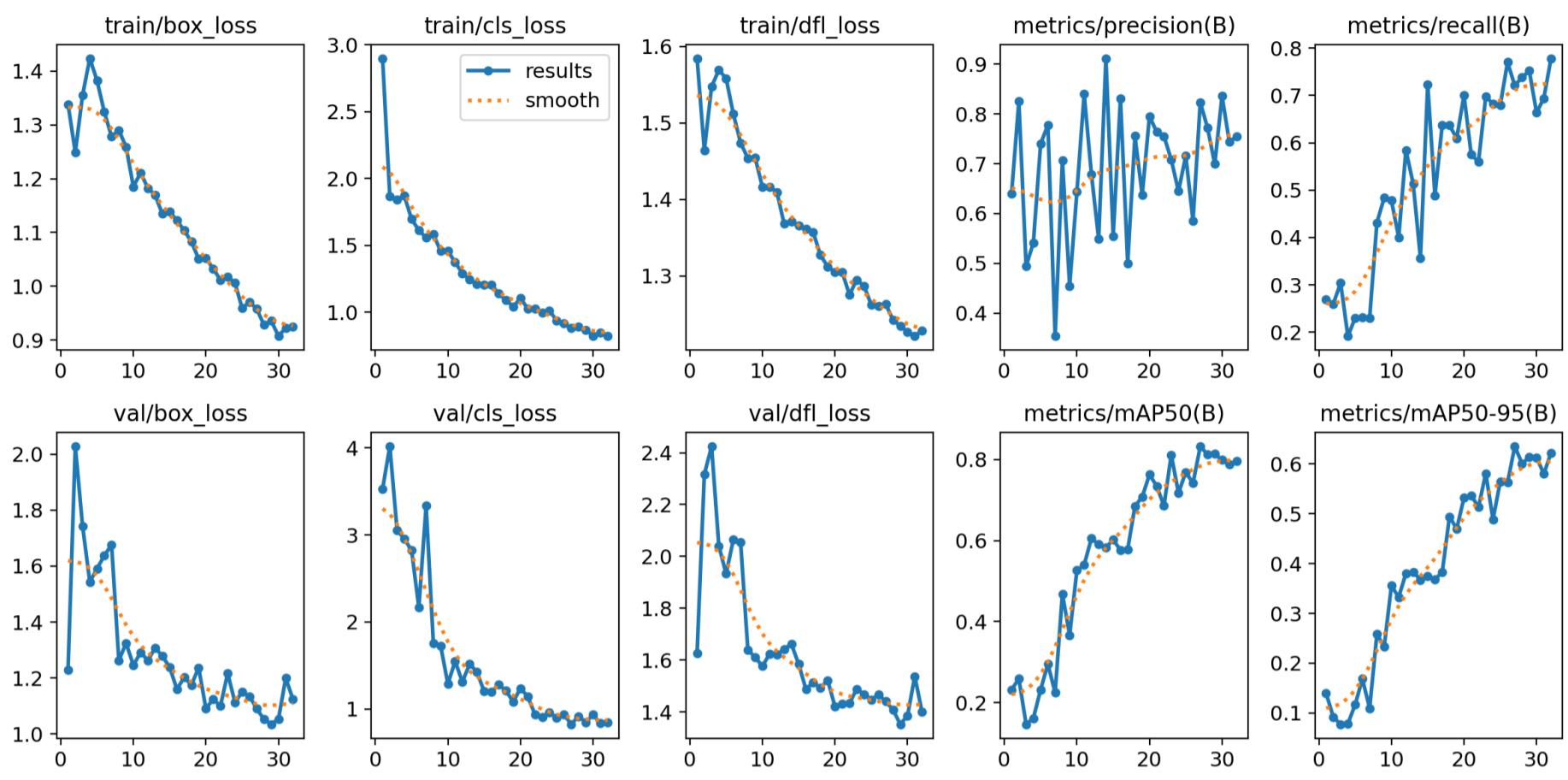
from ultralytics import YOLO
model = YOLO('yolov8m.pt')

Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8m.pt to 'yolov8m.pt'...
100%|██████████| 49.7M/49.7M [00:00-00:00, 122MB/s]

model.train(data = '/content/esb_detect_merged-2/data.yaml', epochs = 50, patience = 5)

0.6967, 0.6977, 0.6987, 0.6997, 0.7007, 0.7017, 0.7027, 0.7037, 0.7047,
0.70571, 0.70671, 0.70771, 0.70871, 0.70971, 0.71071, 0.71171, 0.71271, 0.71371, 0.71471,
0.71572, 0.71672, 0.71772, 0.71872, 0.71972, 0.72072, 0.72172, 0.72272, 0.72372, 0.72472,
0.72573, 0.72673, 0.72773, 0.72873, 0.72973, 0.73073, 0.73173, 0.73273, 0.73373, 0.73473, 0.73573, 0.73673, 0.73773, 0.73873,
0.73974, 0.74074, 0.74174, 0.74274, 0.74374, 0.74474, 0.74575, 0.74675, 0.74775, 0.74875,
0.74975, 0.75075, 0.75175, 0.75275, 0.75375, 0.75475, 0.75576, 0.75676, 0.75776, 0.75876, 0.75976, 0.76076, 0.76176, 0.76276,
0.76376, 0.76476, 0.76577, 0.76677, 0.76777, 0.76877, 0.76977, 0.77077, 0.77177, 0.77277,
0.77377, 0.77477, 0.77577, 0.77677, 0.77777, 0.77877, 0.77977, 0.78077, 0.78177, 0.78277, 0.78377, 0.78477, 0.78577, 0.78677,
0.78777, 0.78877, 0.78977, 0.79077, 0.79177, 0.79277, 0.79377, 0.79477, 0.79577, 0.79677,
0.79777, 0.79877, 0.79977, 0.80077, 0.80177, 0.80277, 0.80377, 0.80477, 0.80577, 0.80677,
0.80777, 0.80877, 0.80977, 0.81077, 0.81177, 0.81277, 0.81377, 0.81477, 0.81577, 0.81677,
0.81777, 0.81877, 0.81977, 0.82077, 0.82177, 0.82277, 0.82377, 0.82477, 0.82577, 0.82677,
0.82777, 0.82877, 0.82977, 0.83077, 0.83177, 0.83277, 0.83377, 0.83477, 0.83577, 0.83677,
0.83777, 0.83877, 0.83977, 0.84077, 0.84177, 0.84277, 0.84377, 0.84477, 0.84577, 0.84677,
0.84777, 0.84877, 0.84977, 0.85077, 0.85177, 0.85277, 0.85377, 0.85477, 0.85577, 0.85677,
0.85777, 0.85877, 0.85977, 0.86077, 0.86177, 0.86277, 0.86377, 0.86477, 0.86577, 0.86677,
0.86777, 0.86877, 0.86977, 0.87077, 0.87177, 0.87277, 0.87377, 0.87477, 0.87577, 0.87677,
0.87777, 0.87877, 0.87977, 0.88077, 0.88177, 0.88277, 0.88377, 0.88477, 0.88577, 0.88677,
0.88777, 0.88877, 0.88977, 0.89077, 0.89177, 0.89277, 0.89377, 0.89477, 0.89577, 0.89677,
0.89777, 0.89877, 0.89977, 0.90077, 0.90177, 0.90277, 0.90377, 0.90477, 0.90577, 0.90677,
0.90777, 0.90877, 0.90977, 0.91077, 0.91177, 0.91277, 0.91377, 0.91477, 0.91577, 0.91677,
0.91777, 0.91877, 0.91977, 0.92077, 0.92177, 0.92277, 0.92377, 0.92477, 0.92577, 0.92677,
0.92777, 0.92877, 0.92977, 0.93077, 0.93177, 0.93277, 0.93377, 0.93477, 0.93577, 0.93677,
0.93777, 0.93877, 0.93977, 0.94077, 0.94177, 0.94277, 0.94377, 0.94477, 0.94577, 0.94677,
0.94777, 0.94877, 0.94977, 0.95077, 0.95177, 0.95277, 0.95377, 0.95477, 0.95577, 0.95677,
0.95777, 0.95877, 0.95977, 0.96077, 0.96177, 0.96277, 0.96377, 0.96477, 0.96577, 0.96677,
0.96777, 0.96877, 0.96977, 0.97077, 0.97177, 0.97277, 0.97377, 0.97477, 0.97577, 0.97677,
0.97777, 0.97877, 0.97977, 0.98077, 0.98177, 0.98277, 0.98377, 0.98477, 0.98577, 0.98677,
0.98777, 0.98877, 0.98977, 0.99077, 0.99177, 0.99277, 0.99377, 0.99477, 0.99577, 0.99677,
0.99777, 0.99877, 0.99977, 1.00077, 1.00177, 1.00277, 1.00377, 1.00477, 1.00577, 1.00677,
1.00777, 1.00877, 1.00977, 1.01077, 1.01177, 1.01277, 1.01377, 1.01477, 1.01577, 1.01677,
1.01777, 1.01877, 1.01977, 1.02077, 1.02177, 1.02277, 1.02377, 1.02477, 1.02577, 1.02677,
1.02777, 1.02877, 1.02977, 1.03077, 1.03177, 1.03277, 1.03377, 1.03477, 1.03577, 1.03677,
1.03777, 1.03877, 1.03977, 1.04077, 1.04177, 1.04277, 1.04377, 1.04477, 1.04577, 1.04677,
1.04777, 1.04877, 1.04977, 1.05077, 1.05177, 1.05277, 1.05377, 1.05477, 1.05577, 1.05677,
1.05777, 1.05877, 1.05977, 1.06077, 1.06177, 1.06277, 1.06377, 1.06477, 1.06577, 1.06677,
1.06777, 1.06877, 1.06977, 1.07077, 1.07177, 1.07277, 1.07377, 1.07477, 1.07577, 1.07677,
1.07777, 1.07877, 1.07977, 1.08077, 1.08177, 1.08277, 1.08377, 1.08477, 1.08577, 1.08677,
1.08777, 1.08877, 1.08977, 1.09077, 1.09177, 1.09277, 1.09377, 1.09477, 1.09577, 1.09677,
1.09777, 1.09877, 1.09977, 1.10077, 1.10177, 1.10277, 1.10377, 1.10477, 1.10577, 1.10677,
1.10777, 1.10877, 1.10977, 1.11077, 1.11177, 1.11277, 1.11377, 1.11477, 1.11577, 1.11677,
1.11777, 1.11877, 1.11977, 1.12077, 1.12177, 1.12277, 1.12377, 1.12477, 1.12577, 1.12677,
1.12777, 1.12877, 1.12977, 1.13077, 1.13177, 1.13277, 1.13377, 1.13477, 1.13577, 1.13677,
1.13777, 1.13877, 1.13977, 1.14077, 1.14177, 1.14277, 1.14377, 1.14477, 1.14577, 1.14677,
1.14777, 1.14877, 1.14977, 1.15077, 1.15177, 1.15277, 1.15377, 1.15477, 1.15577, 1.15677,
1.15777, 1.15877, 1.15977, 1.16077, 1.16177, 1.16277, 1.16377, 1.16477, 1.16577, 1.16677,
1.16777, 1.16877, 1.16977, 1.17077, 1.17177, 1.17277, 1.17377, 1.17477, 1.17577, 1.17677,
1.17777, 1.17877, 1.17977, 1.18077, 1.18177, 1.18277, 1.18377, 1.18477, 1.18577, 1.18677,
1.18777, 1.18877, 1.18977, 1.19077, 1.19177, 1.19277, 1.19377, 1.19477, 1.19577, 1.19677,
1.19777, 1.19877, 1.19977, 1.20077, 1.20177, 1.20277, 1.20377, 1.20477, 1.20577, 1.20677,
1.20777, 1.20877, 1.20977, 1.21077, 1.21177, 1.21277, 1.21377, 1.21477, 1.21577, 1.21677,
1.21777, 1.21877, 1.21977, 1.22077, 1.22177, 1.22277, 1.22377, 1.22477, 1.22577, 1.22677,
1.22777, 1.22877, 1.22977, 1.23077, 1.23177, 1.23277, 1.23377, 1.23477, 1.23577, 1.23677,
1.23777, 1.23877, 1.23977, 1.24077, 1.24177, 1.24277, 1.24377, 1.24477, 1.24577, 1.24677,
1.24777, 1.24877, 1.24977, 1.25077, 1.25177, 1.25277, 1.25377, 1.25477, 1.25577, 1.25677,
1.25777, 1.25877, 1.25977, 1.26077, 1.26177, 1.26277, 1.26377, 1.26477, 1.26577, 1.26677,
1.26777, 1.26877, 1.26977, 1.27077, 1.27177, 1.27277, 1.27377, 1.27477, 1.27577, 1.27677,
1.27777, 1.27877, 1.27977, 1.28077, 1.28177, 1.28277, 1.28377, 1.28477, 1.28577, 1.28677,
1.28777, 1.28877, 1.28977, 1.29077, 1.29177, 1.29277, 1.29377, 1.29477, 1.29577, 1.29677,
1.29777, 1.29877, 1.29977, 1.30077, 1.30177, 1.30277, 1.30377, 1.30477, 1.30577, 1.30677,
1.30777, 1.30877, 1.30977, 1.31077, 1.31177, 1.31277, 1.31377, 1.31477, 1.31577, 1.31677,
1.31777, 1.31877, 1.31977, 1.32077, 1.32177, 1.32277, 1.32377, 1.32477, 1.32577, 1.32677,
1.32777, 1.32877, 1.32977, 1.33077, 1.33177, 1.33277, 1.33377, 1.33477, 1.33577, 1.33677,
1.33777, 1.33877, 1.33977, 1.34077, 1.34177, 1.34277, 1.34377, 1.34477, 1.34577, 1.34677,
1.34777, 1.34877, 1.34977, 1.35077, 1.35177, 1.35277, 1.35377, 1.35477, 1.35577, 1.35677,
1.35777, 1.35877, 1.35977, 1.36077, 1.36177, 1.36277, 1.36377, 1.36477, 1.36577, 1.36677,
1.36777, 1.36877, 1.36977, 1.37077, 1.37177, 1.37277, 1.37377, 1.37477, 1.37577, 1.37677,
1.37777, 1.37877, 1.37977, 1.38077, 1.38177, 1.38277, 1.38377, 1.38477, 1.38577, 1.38677,
1.38777, 1.38877, 1.38977, 1.39077, 1.39177, 1.39277, 1.39377, 1.39477, 1.39577, 1.39677,
1.39777, 1.39877, 1.39977, 1.40077, 1.40177, 1.40277, 1.40377, 1.40477, 1.40577, 1.40677,
1.40777, 1.40877, 1.40977, 1.41077, 1.41177, 1.41277, 1.41377, 1.41477, 1.41577, 1.41677,
1.41777, 1.41877, 1.41977, 1.42077, 1.42177, 1.42277, 1.42377, 1.42477, 1.42577, 1.42677,
1.42777, 1.42877, 1.42977, 1.43077, 1.43177, 1.43277, 1.43377, 1.43477, 1.43577, 1.43677,
1.43777, 1.43877, 1.43977, 1.44077, 1.44177, 1.44277, 1.44377, 1.44477, 1.44577, 1.44677,
1.44777, 1.44877, 1.44977, 1.45077, 1.45177, 1.45277, 1.45377, 1.45477, 1.45577, 1.45677,
1.45777, 1.45877, 1.45977, 1.46077, 1.46177, 1.46277, 1.46377, 1.46477, 1.46577, 1.46677,
1.46777, 1.46877, 1.46977, 1.47077, 1.47177, 1.47277, 1.47377, 1.47477, 1.47577, 1.47677,
1.47777, 1.47877, 1.47977, 1.48077, 1.48177, 1.48277, 1.48377, 1.48477, 1.48577, 1.48677,
1.48777, 1.48877, 1.48977, 1.49077, 1.49177, 1.4927
```

총 32 epoch 학습 후 학습이 조기종료되었습니다. 학습 결과 그래프는 아래와 같습니다.



학습 결과 그래프. validation box, cls, dfl loss가 나를 유의미하게 감소함을 확인

training result로 '**best.pt**' 파일이 생성되었습니다.(별첨 모델 파일) training 중 가장 성능이 우수했던 가중치 파일이 자동으로 저장된 파일이고 이를 검출 모델로 사용합니다.

이를 통해 기존 YOLOv8 모델(yolov8m.pt)을 기반으로 엠파이어 스테이트 빌딩 데이터에 대해 재학습하여 유의미한 검출성을 도출해내도록 하였습니다.

최종 엠파이어 스테이트 빌딩 검출 소스코드는 **OpenCV(cv2)** 및 **ultralytics 라이브러리(YOLOv8 사용)** 베이스로 작성하였습니다.

- `results = model.predict(source = img, save = False, conf = 0.60)`

confidence 값이 일정 수준을 넘는 객체만 엠파이어 스테이트 빌딩으로 검출하자는 아이디어를 이용하였습니다. 다양한 결과를 바탕으로 허리스틱하게 0.60(60%)이 넘는 객체들만 엠파이어 스테이트 빌딩으로 검출되도록 설정하였습니다.

대부분의 엠파이어 스테이트 빌딩 이미지에 대해 강건하게 검출을 잘 수행합니다. 또한 과제 명세 pdf에 나온 여러 가지 빌딩들을 꽤 좋은 성능으로 필터링했음을 확인했습니다. OpenCV를 사용하여 원본 이미지의 엠파이어 스테이트 빌딩 부분을 빨간색 경계 상자로 표시합니다.(이미지 크기에 따라 박스 두께가 얇게 나오는 경우도 있어 유심히 봄해야 하는 경우도 존재)

결론적으로 연관 데이터셋 다량 확보, YOLOv8의 효율적인 객체 검출 능력과 OpenCV의 유연한 이미지 처리 기능을 결합하여 엠파이어 스테이트 빌딩의 검출을 간단하게 수행하였습니다.

References

1. 강의 교안
2. https://www.youtube.com/watch?v=em_lOAp8DJE&t=875s → YOLOv8 파인튜닝 프로세스 이해를 위해 활용(동영상강의)
3. <https://universe.roboflow.com/project-y2pir/empire-state-building-09uwo> → 엠파이어 스테이트 빌딩 관련 데이터셋 1
4. <https://universe.roboflow.com/soongsil-lvhjq/empire-ecdms> → 엠파이어 스테이트 빌딩 관련 데이터셋 2

5. <https://universe.roboflow.com/project-d4ffr/emfire-building> → 엠파이어 스테이트 빌딩 관련 데이터셋 3
6. <https://universe.roboflow.com/182596/test-myvxo> → 엠파이어 스테이트 빌딩 관련 데이터셋 4
7. https://universe.roboflow.com/cv2023-r3b67/esb_full_box → 엠파이어 스테이트 빌딩 관련 데이터셋 5
8. https://universe.roboflow.com/esbdetection/esb_detect_merged → 위 데이터셋 5개를 merge하여 최종적으로 사용한 데이터셋
9. ChatGPT, Claude → 막하는 사항 해결 및 코드 가독성 향상을 위해 활용

감사합니다.
