인공지능 프로그래밍

(LeNet-5를 이용한 필기체 숫자 인식)

인공지능전공
양 준 서
202576720

프로그램 개요

컴퓨터 비전 부문에서 현재 B2B, B2C를 막론하고 가장 활용도가 높은 분야 중 하나인 OCR(Optical Character Recognition) 기술과 관련하여, 필기체 숫자 인식을 LeNet-5 모델을 이용하여 구현하였습니다.

1. 캐글 데이터셋 불러오기 : kaggle API를 활용, 학습 및 테스트 용 데이터 수집
2. 시각화 확인 : 학습 데이터를 파이썬의 대표적인 시각화 라이브러리인 Matplotlib을 이용하여 확인
3. ML 모델 선정 : 초기 손으로 쓴 우편번호 인식을 위해 만들어졌고, ATM에서 수표를 읽는데 사용되었던 합성곱 신경망 아키텍처인 LeNet-5를 적용함
4. 데이터셋 분리 & 로드 : 캐글 데이터셋에서 가져올 때 이미 training set 60,000개와 test set 10,000로 분리되어져 있었으므로 분리 과정은 불필요하며, 각각에 대해 DataLoading 과정을 거침
5. 학습 실행 : 초기에는 최적화 과정 없이 학습을 실행
6. 평가 결과 확인 : 5. 번 과정으로 만들어진 모델에 대해 평가 결과 확인하였으며, Epoch 과정이 증가되더라도 정확도 변화 없음
7. 최적화 과정 포함하여 학습 재 실행 : 최적화 기법인 “확률적 경사 하강법(Stochastic Gradient Descent, SGD)”을 적용하여 학습 재 실행
8. 평가 결과 확인 : 7.번 과정으로 만들어진 모델에 대해 평가 결과 확인하였으며, Epoch 증가 시 정확도 증가
9. 최종 분석 결과 : 최적화 기법을 적용할 때 모델의 성능을 최대화하여 정확한 분류 작업을 수행할 수 있었음

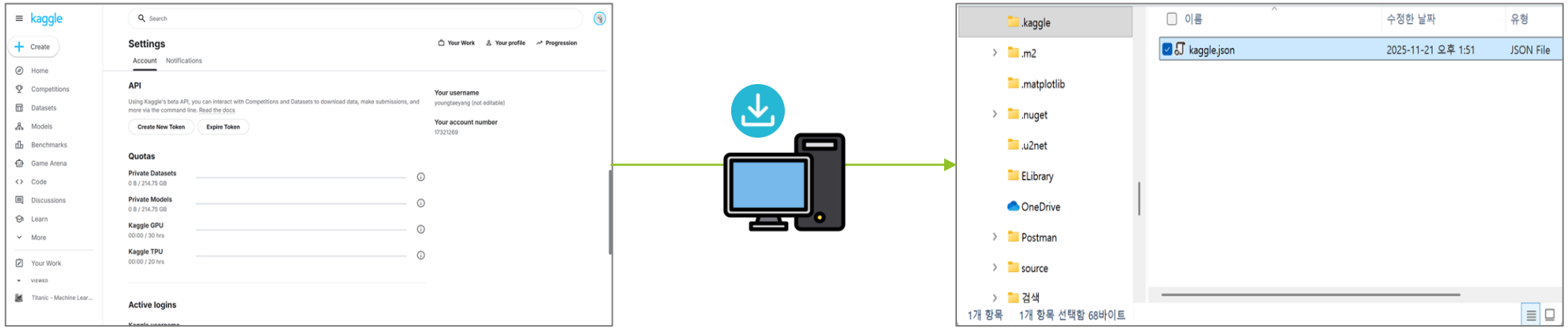
1. 캐글 데이터셋 불러오기

▶ Kaggle API를 이용하여 데이터를 가져옴

1) Kaggle 패키지 인스톨 : pip install kaggle

2) Kaggle TOKEN 받기 : Kaggle API를 이용하여 데이터를 받기 위해서는 Kaggle 인증 과정 필요 ➔ kaggle.api.authenticate()

Kaggle에서 TOKEN을 받은 후, 로컬 컴퓨터 ("[사용자 홈 디렉토리]₩.kaggle")에 저장



3) Kaggle 데이터 불러오기 : csv 파일 형식의 MNIST 데이터 다운로드

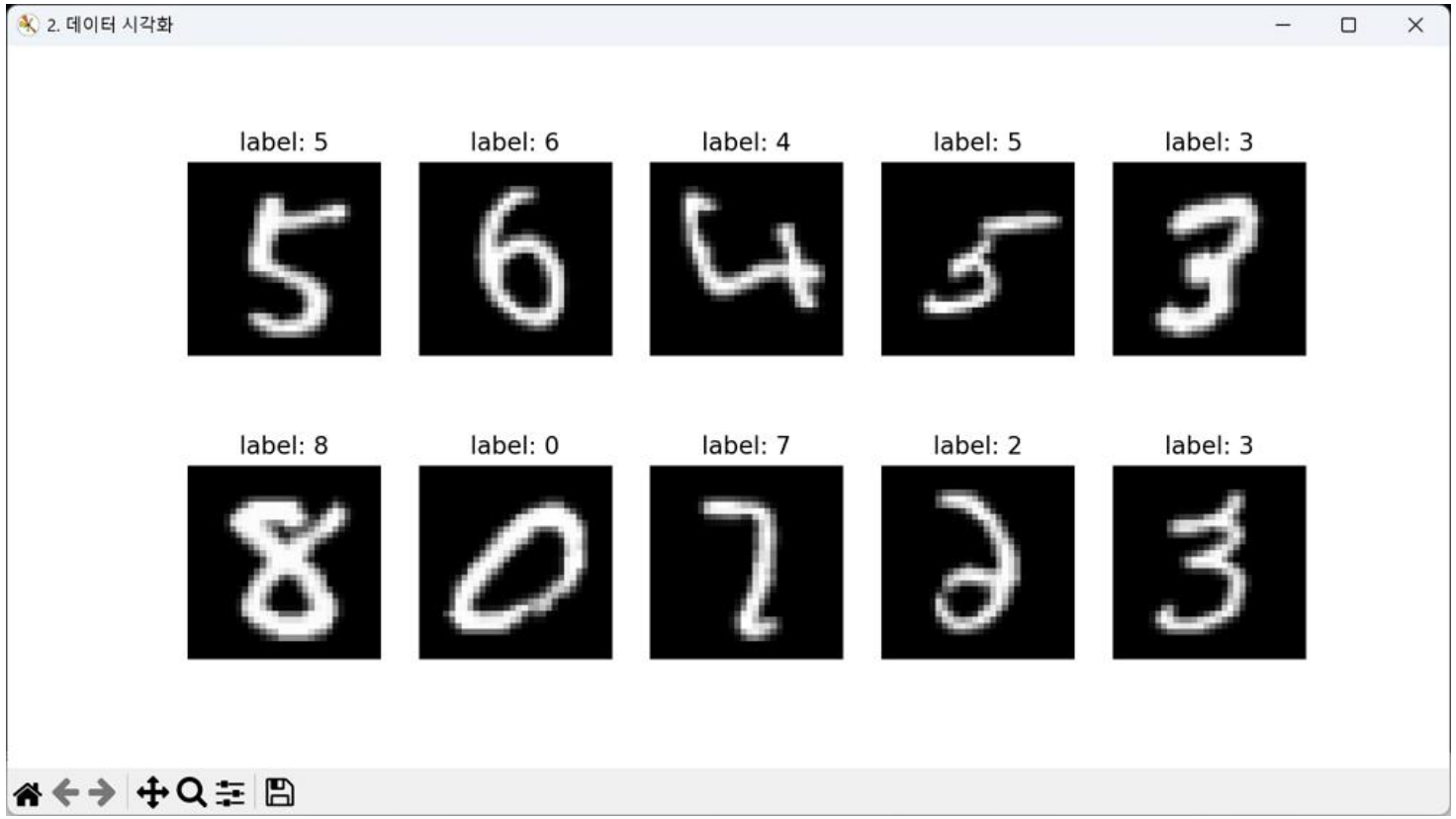
URL : [https://www.kaggle.com/datasets/quangphota/mnist-csv] 에서 [https://www.kaggle.com/datasets/] 뒷 부분인 "quangphota/mnist-csv" 이 데이터셋 식별자(dataset identifier) 임

➔ kaggle.api.dataset_download_files("quangphota/mnist-csv" , path=DATASET_PATH, unzip=True)

2. 데이터 시각화

▶ Matplotlib 라이브러리를 이용하여 데이터 시각화

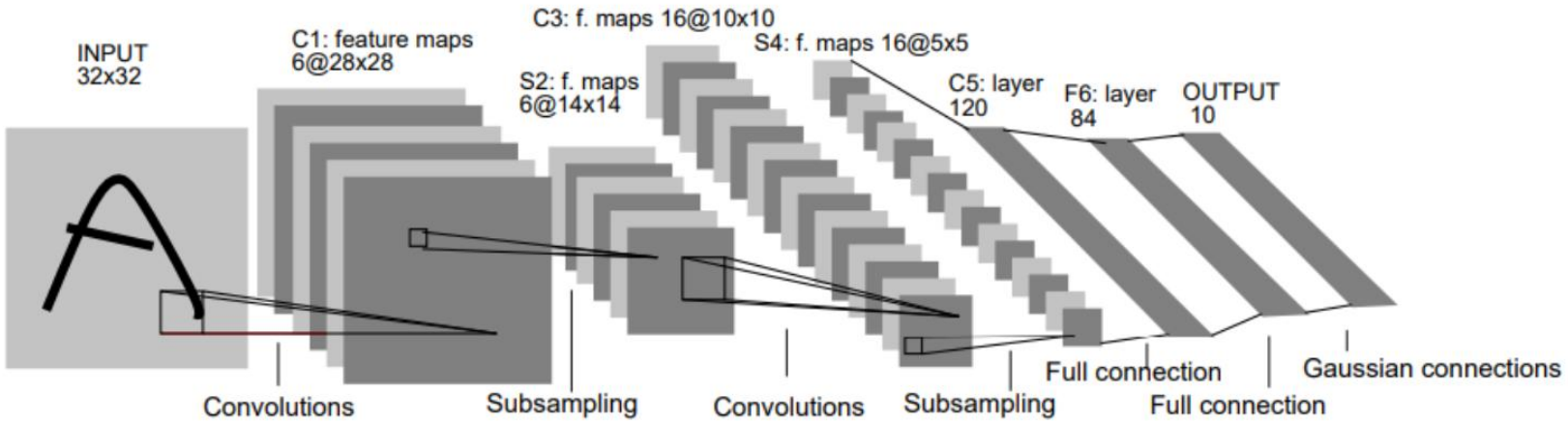
Matplotlib 라이브러리를 이용하여 받은 데이터 셋 중 무작위로 10개의 필기체 숫자를 보여줌



3. 모델 선정 & 4. 데이터 로드

▶ 모델 : LeNet-5

Lenet은 얀 르쿤이 제안한 합성곱 신경망 구조이며, 이미지 분류용 CNN의 초기 모델



[참고] Architecture of LeNet-5(논문 발췌)

[프로그램 소스]

```
class LeNet_5(nn.Module):
    def __init__(self):
        super(LeNet_5,self).__init__()
        self.c1 = nn.Conv2d(1,6,5)
        self.maxpool1 = nn.MaxPool2d(2)
        self.c2 = nn.Conv2d(6,16,5)
        self.maxpool2 = nn.MaxPool2d(2)
        self.c3 = nn.Conv2d(16,120,5)
        self.n1 = nn.Linear(120,84)
        self.relu = nn.ReLU()
        self.n2 = nn.Linear(84,10)

    def forward(self,x):
        x = self.c1(x)
        x = self.relu(x)
        x = self.maxpool1(x)
        x = self.c2(x)
        x = self.relu(x)
        x = self.maxpool2(x)
        x = self.c3(x)
        x = self.relu(x)
        x = torch.flatten(x,1)
        x = self.n1(x)
        x = self.relu(x)
        x = self.n2(x)
        return x
```

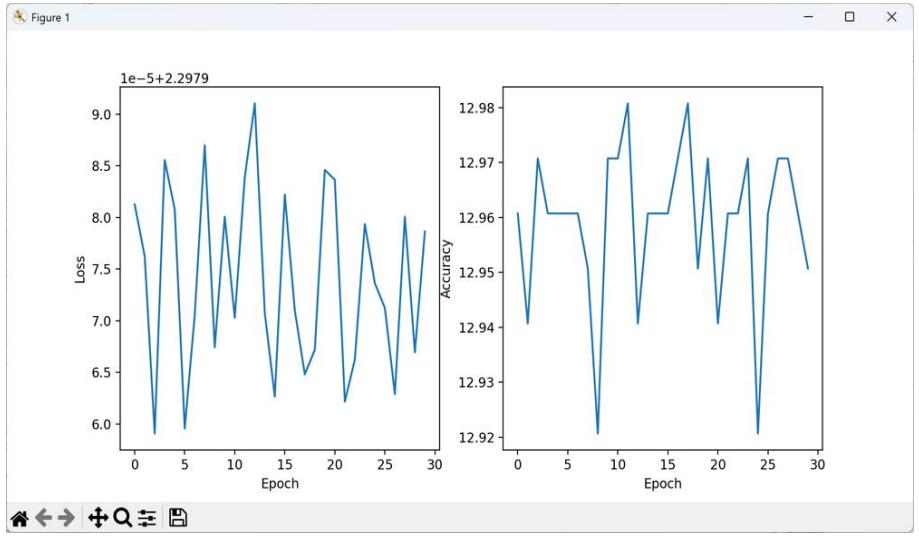
캐글에서 데이터 다운로드 시 이미 training set과 test set으로 분리되어 있었으므로, 데이터 분리 과정은 생략

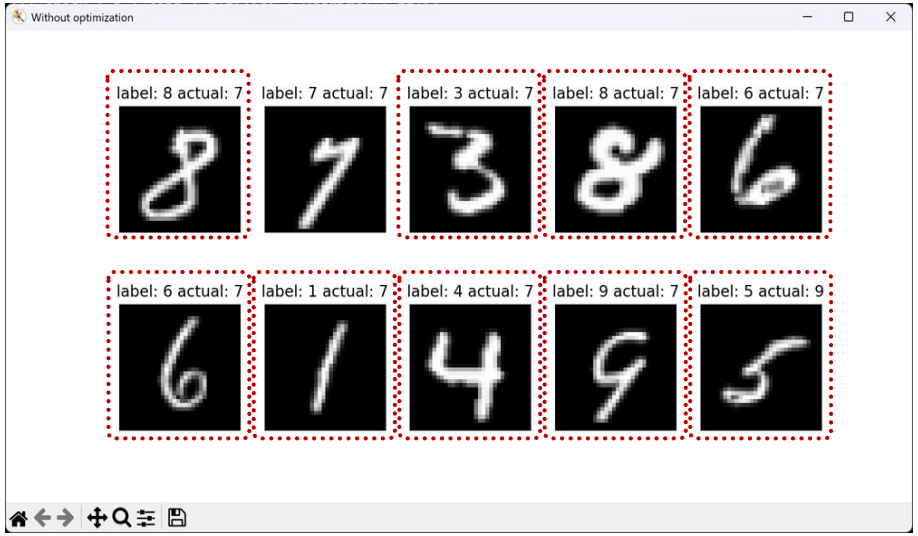
5. 학습 실행 & 6. 평가 결과 확인

▶ 최적화 과정 없이 학습 실행. Batch Size=128, Epoch=30

최적화 과정 없이 학습을 수행한 경우, 아래 결과와 같이 손실 값과 정확도가 학습을 지속하여도 향상되지 않음

```
5) 학습실행
+++ 디바이스 => [cuda]
... epoch : 1 | loss : 2.297981 | Accuracy : 12.96
... epoch : 2 | loss : 2.297976 | Accuracy : 12.94
... epoch : 3 | loss : 2.297959 | Accuracy : 12.97
... epoch : 4 | loss : 2.297986 | Accuracy : 12.96
... epoch : 5 | loss : 2.297981 | Accuracy : 12.96
... epoch : 6 | loss : 2.297960 | Accuracy : 12.96
... epoch : 7 | loss : 2.297971 | Accuracy : 12.96
... epoch : 8 | loss : 2.297987 | Accuracy : 12.95
... epoch : 9 | loss : 2.297967 | Accuracy : 12.92
... epoch : 10 | loss : 2.297980 | Accuracy : 12.97
... epoch : 11 | loss : 2.297970 | Accuracy : 12.97
... epoch : 12 | loss : 2.297984 | Accuracy : 12.98
... epoch : 13 | loss : 2.297991 | Accuracy : 12.94
... epoch : 14 | loss : 2.297971 | Accuracy : 12.96
... epoch : 15 | loss : 2.297963 | Accuracy : 12.96
... epoch : 16 | loss : 2.297982 | Accuracy : 12.96
... epoch : 17 | loss : 2.297971 | Accuracy : 12.97
... epoch : 18 | loss : 2.297965 | Accuracy : 12.98
... epoch : 19 | loss : 2.297967 | Accuracy : 12.95
... epoch : 20 | loss : 2.297985 | Accuracy : 12.97
... epoch : 21 | loss : 2.297984 | Accuracy : 12.94
... epoch : 22 | loss : 2.297962 | Accuracy : 12.96
... epoch : 23 | loss : 2.297966 | Accuracy : 12.96
... epoch : 24 | loss : 2.297979 | Accuracy : 12.97
... epoch : 25 | loss : 2.297974 | Accuracy : 12.92
... epoch : 26 | loss : 2.297971 | Accuracy : 12.96
... epoch : 27 | loss : 2.297963 | Accuracy : 12.97
... epoch : 28 | loss : 2.297980 | Accuracy : 12.97
... epoch : 29 | loss : 2.297967 | Accuracy : 12.96
... epoch : 30 | loss : 2.297979 | Accuracy : 12.95
```



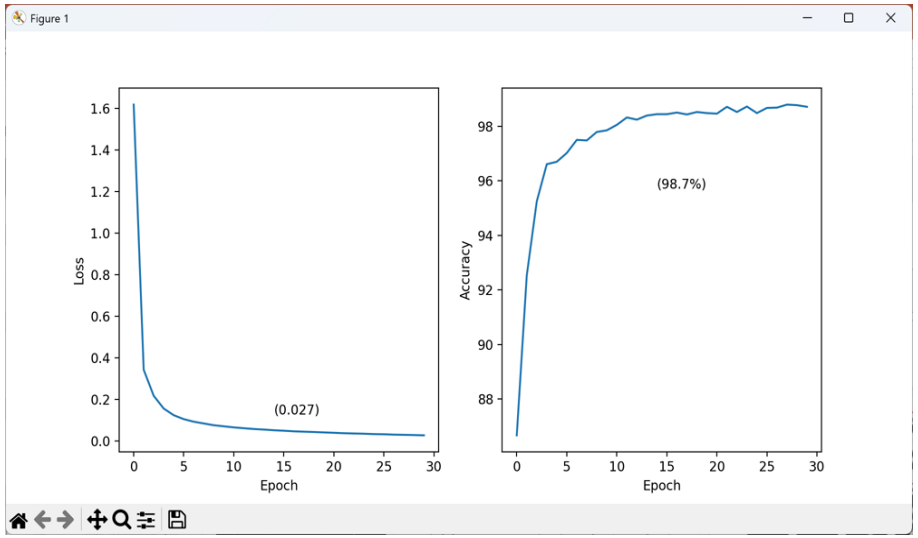


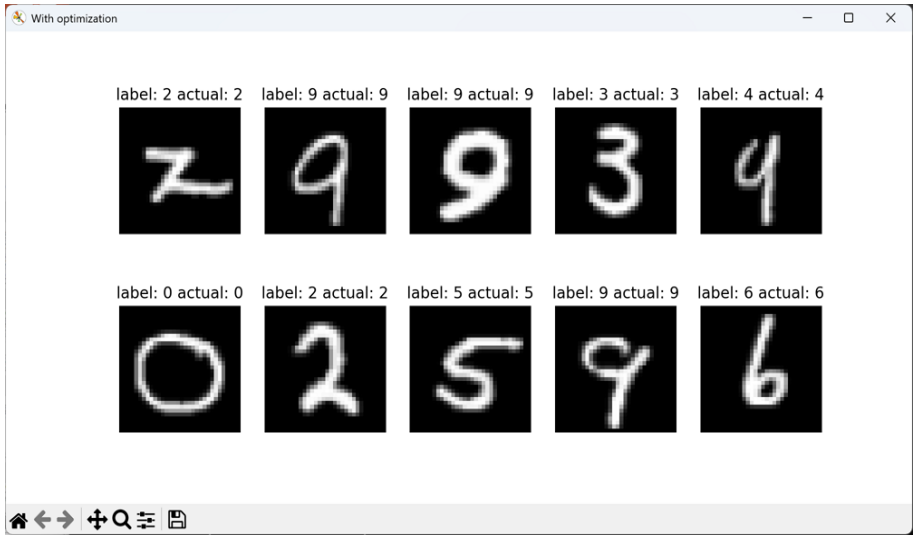
7. 최적화 포함 학습 실행 & 8. 평가 결과 확인

▶ 최적화 포함하여 학습 실행. Batch Size=128, Epoch=30

Epoch가 증가되면서 정확도가 향상됨

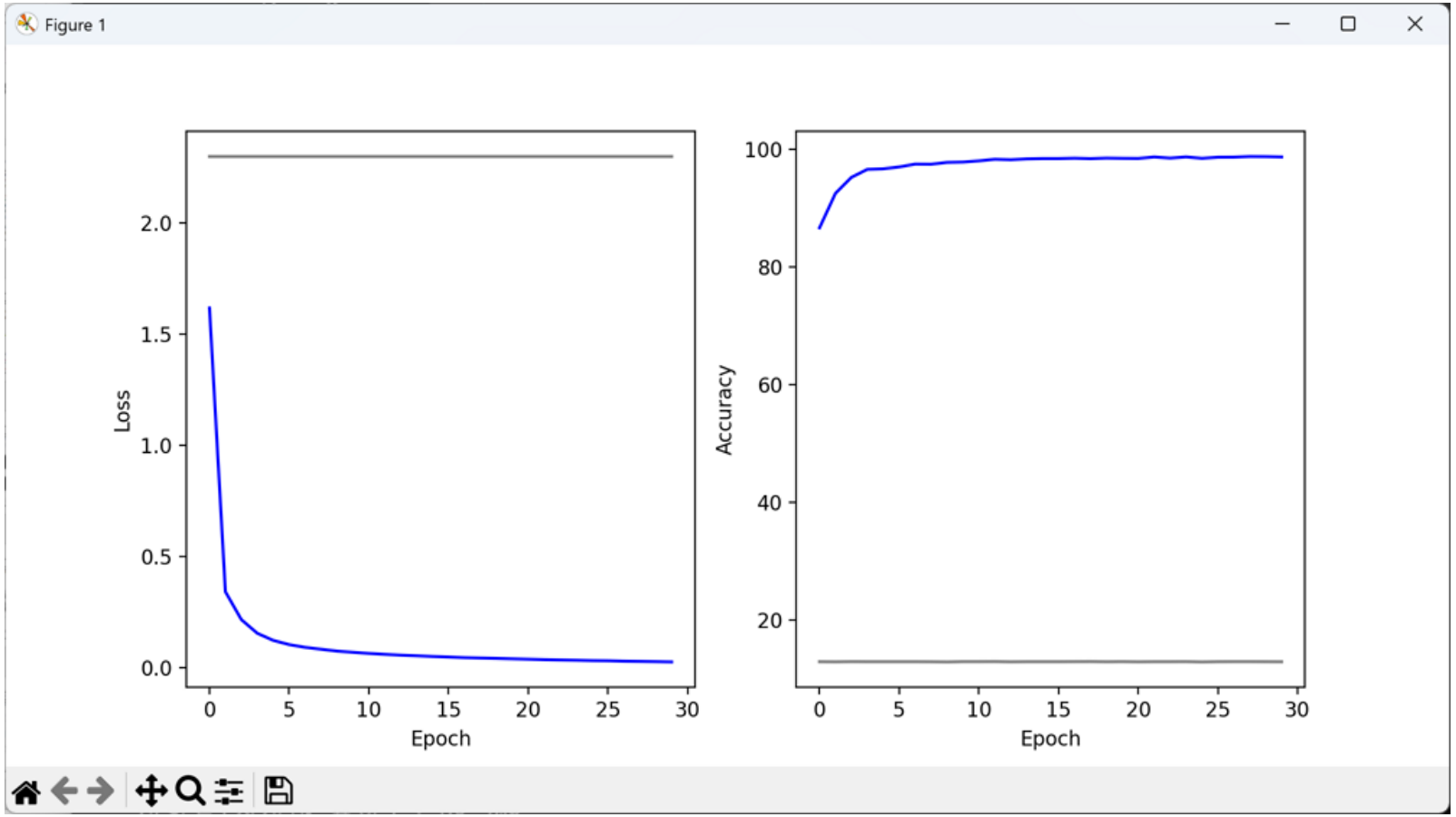
```
7) 최적화 실행하여 학습 재실행
... epoch : 1 | loss : 1.617321 | Accuracy : 86.67
... epoch : 2 | loss : 0.342366 | Accuracy : 92.53
... epoch : 3 | loss : 0.217146 | Accuracy : 95.24
... epoch : 4 | loss : 0.155716 | Accuracy : 96.60
... epoch : 5 | loss : 0.123730 | Accuracy : 96.69
... epoch : 6 | loss : 0.104813 | Accuracy : 97.02
... epoch : 7 | loss : 0.092526 | Accuracy : 97.50
... epoch : 8 | loss : 0.083997 | Accuracy : 97.48
... epoch : 9 | loss : 0.075511 | Accuracy : 97.79
... epoch : 10 | loss : 0.070271 | Accuracy : 97.85
... epoch : 11 | loss : 0.065125 | Accuracy : 98.05
... epoch : 12 | loss : 0.061143 | Accuracy : 98.32
... epoch : 13 | loss : 0.057376 | Accuracy : 98.24
... epoch : 14 | loss : 0.054674 | Accuracy : 98.39
... epoch : 15 | loss : 0.051461 | Accuracy : 98.44
... epoch : 16 | loss : 0.049134 | Accuracy : 98.44
... epoch : 17 | loss : 0.046227 | Accuracy : 98.50
... epoch : 18 | loss : 0.044484 | Accuracy : 98.43
... epoch : 19 | loss : 0.042940 | Accuracy : 98.52
... epoch : 20 | loss : 0.041009 | Accuracy : 98.48
... epoch : 21 | loss : 0.039113 | Accuracy : 98.46
... epoch : 22 | loss : 0.037144 | Accuracy : 98.71
... epoch : 23 | loss : 0.035832 | Accuracy : 98.52
... epoch : 24 | loss : 0.034657 | Accuracy : 98.72
... epoch : 25 | loss : 0.032897 | Accuracy : 98.48
... epoch : 26 | loss : 0.032130 | Accuracy : 98.67
... epoch : 27 | loss : 0.030261 | Accuracy : 98.68
... epoch : 28 | loss : 0.029409 | Accuracy : 98.79
... epoch : 29 | loss : 0.028366 | Accuracy : 98.77
... epoch : 30 | loss : 0.027001 | Accuracy : 98.71
```





9. 최종분석결과

▶ 최적화는 모델의 성능과 효율성을 향상시키기 위한 필수 과정임



감사합니다