**딥러닝 이해 4주차 과제**

현재 MLP 모델 정의는 아래와 같다.

**- MLP 모델 코드**

1. # Define the MLP model

2. class MLP(nn.Module):

3. def \_\_init\_\_(self, input\_size, hidden\_size, num\_classes, dropout\_rate):

4. super(MLP, self).\_\_init\_\_()

5. self.fc1 = nn.Linear(input\_size, hidden\_size)

6. self.relu = nn.ReLU()

7. self.dropout = nn.Dropout(dropout\_rate)

8. self.fc2 = nn.Linear(hidden\_size, num\_classes)

9.

10. def forward(self, x):

11. x = x.view(-1, input\_size)

12. x = self.fc1(x)

13. x = self.relu(x)

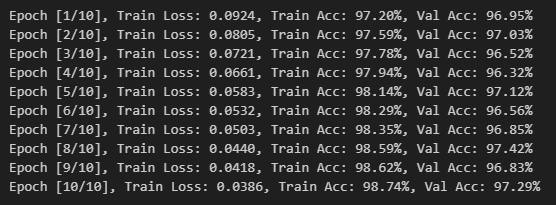
14. # x = self.dropout(x)

15. x = self.fc2(x)

16. return x

17.

10번의 학습을 거치고 난 이후, loss 값 변화



최종 정확도



이 보고서에서는 각각 정규화, batch 사이즈 변경, DROP OUT, running rate 조정을 적용했을 때 정확도가 어떻게 변화하는지를 보고, 하이퍼 파라미터 튜닝을 하려 한다.

## BATCH SIZE 변경

1. # Hyperparameters

2. input\_size = 28 \* 28 # MNIST images are 28x28 pixels

3. hidden\_size = 128

4. num\_classes = 10

5. num\_epochs = 10

6. batch\_size = 64

7. learning\_rate = 0.001

1. transform = transforms.Compose([

2. transforms.ToTensor(),

3. transforms.Normalize((0.5,), (0.5,))

4. ])

5.

6. train\_dataset = torchvision.datasets.MNIST(

7. root='./data', train=True, transform=transform, download=True)

8.

9. test\_dataset = torchvision.datasets.MNIST(

10. root='./data', train=False, transform=transform, download=True)

11.

12. print("Num of train Dataset: ", len(train\_dataset), "\tNum of test Dataset: ", len(test\_dataset))

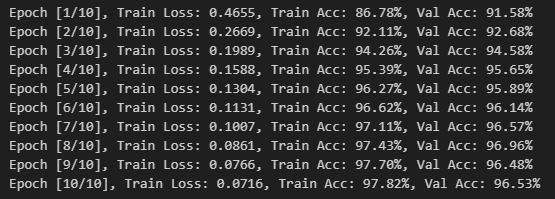
13.

현재 batch\_size는 64개이다. 그리고 dataset은 위처럼 구성되어 있다.

Batch는 학습을 할 때 노이즈를 제거해 조금 더 안정적인 결과가 나오도록 한다. 이 때, 기존의 batch\_size를 바꾸면 어떨까? Batch Size가 많고, 적어졌을 때 정확도가 어떻게 변화하는지 확인해보도록 하겠다.

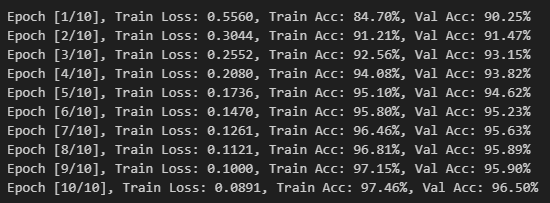
BATCH 사이즈를 증가시키는 경우

1. Batch\_Size = 128





2. Batch\_Size = 256

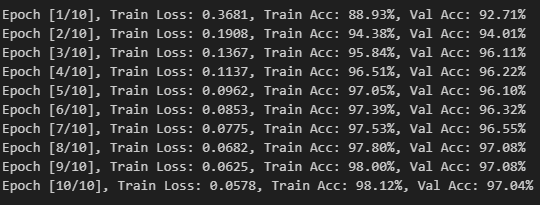




위 결과를 확인하였을 때에는 배치사이즈를 2의 제곱수로 증가시켜 적용했을 때, 오히려 정확도가 떨어지는 것을 볼 수 있다. 그렇다면 BATCH SIZE를 줄이면 어떨까?

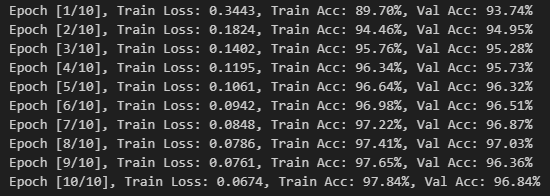
Batch Size를 감소시키는 경우

3. Batch Size = 32





4. Batch Size = 16





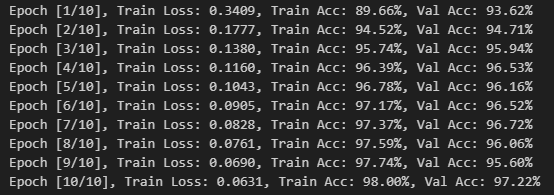
2 경우 모두를 살펴보았을 때, batch\_size가 너무 커도, 작아도 정확도가 오히려 떨어지는 것을 확인할 수 있다.

## DROP OUT

Drop out이란 모델이 학습으르 할 때, 편향이 너무 증가하는 것을 방지하기 위해, 랜덤으로 학습에 사용하는 노드들을 정해 사용하는 것을 의미한다.

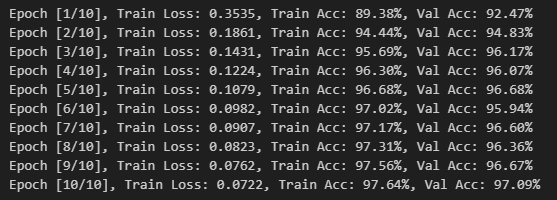
현재 학습에 사용되는 drop out은 0.2이다. 그렇다면 Batch\_size는 64로 고정한 상태에서, drop out의 크기가 0.2보다 크거나 작을 때 정확도가 어떻게 변화하는 지를 살펴보도록 한다.

DROP-OUT = 0 // 아예 적용하지 않음



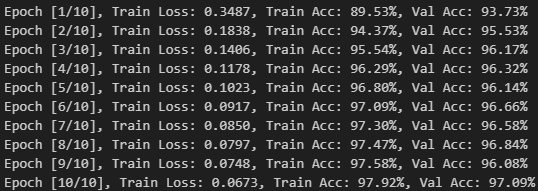


DROP-OUT = 0.4



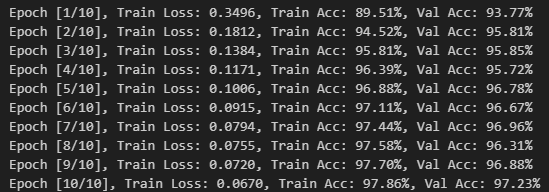


DROP-OUT = 0.5





DROP-OUT=0.8





DROP-OUT은 너무 낮을 경우, 동일한 학습을 반복하기 때문에 정확도가 낮게 나오고, 너무 높아도 학습이 제대로 이루어지지 않을 수 있어 좋지 않다. 예측하기에는 0.5 부근에서 가장 정확도가 높게 나올 것이라고 생각했는데 의외로 0.8에서 가장 정확도가 높게 나왔다.

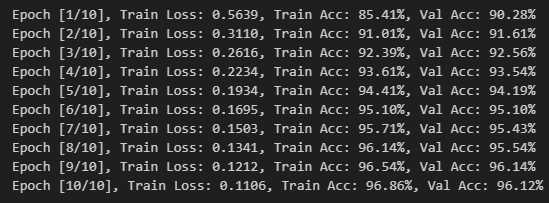
# Learning Rate

Learning rate란 모델이 가중치 및 편향치를 조절하는 속도를 의미한다. Learning\_rate는 모델에 따라 다르지만 통상적으로 0.001~0.0001 사이의 값을 많이 사용한다.

현재 모델에서는 0.001의 값을 가지고 있다. Learning\_rate 크기를 0.001 기준으로 크거나 작게 조절했을 때, 정확도가 어떻게 나오게 되는지를 알아보고자 한다.

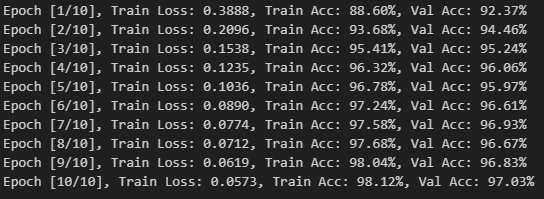
Learning\_rate를 낮췄을 때

Learning\_rate=0.0001 #기존 속도에서 1/10배





Learning\_rate = 0.0005 # 기존 속도에서 1/2배

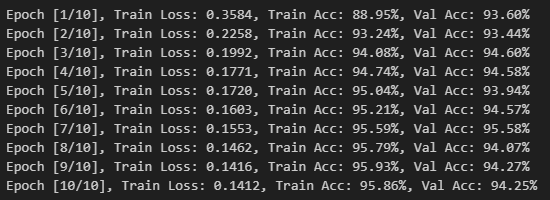




Learning\_rate가 낮을수록, w, b의 값 업데이트 속도가 느려진다. Train\_acc 및 val\_acc이 오르는 속도가 비교적 느린 것에서 그것을 확인 할 수 있다.

Learning\_rate를 높였을 때

Learning\_rate = 0.003 # 기존 속도에서 3배



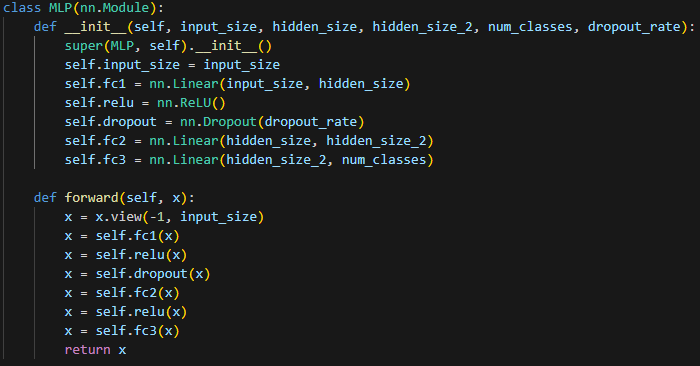


당장 0.003의 속도만 가져도 정확도가 낮아지는 것을 확인할 수 있다. Learning\_rate는 너무 낮을 경우에는 정확도를 높이는데 시간이 오래 걸리고, 너무 높을 경우에는 제대로 가중치 및 편향치를 업데이트 하지 못하는 모습을 볼 수 있다.

# 모델 수정

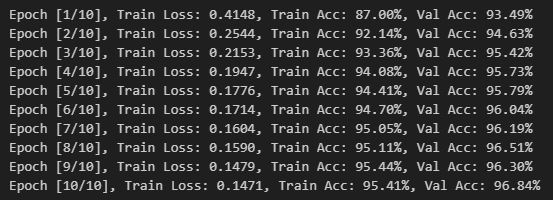
현재는 28\*28의 입력을 받아, 128개의 hidden\_node를 거쳐 10개의 출력값 중 하나로 결과를 도출하는 모델을 가지고 있다. 그렇다면 이 모델 구조에 새로운 hidden\_node를 추가해보면 어떨까? hidden\_layer를 1층만 가지고 있던 모델을 수정해 다음처럼 새로운 모델을 구성해보도록 하겠다.

모델 예시



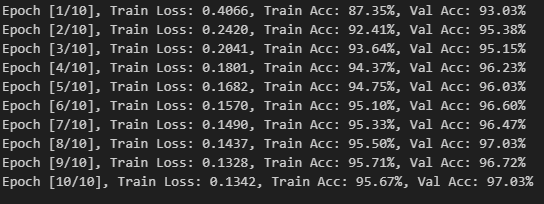
Hidden\_size\_2의 구성에 따라 정확도가 어떻게 변화하는지 살펴보겠다.

 Hidden\_size\_2=256



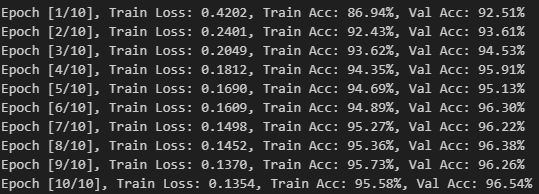


 Hidden\_size=128



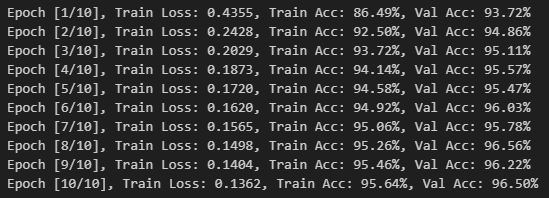


 Hidden\_size=64





 Hidden\_Size=32





Hidden\_layer를 추가해서 학습을 했을 때, 그닥 효과가 없는 것을 볼 수 있다.

# 초기화

마지막으로 볼 것은 가중치 및 편차 초기화이다. 현재는 아무 초기화를 거치지 않고, default 균등 분포를 통해 모델을 사용하고 있다. 그렇다면 여기에 he 혹은 xavier 초기화를 적용하면 어떨까? 해당 가정을 실험해 보기 위해

1. import torch.nn.init as init

2.

3. def init\_weights(m):

4.     if isinstance(m, nn.Linear):

5.         # Xavier 초기화 (tanh에 적합)

6.         # init.xavier\_uniform\_(m.weight)

7.

8.         # He 초기화 (ReLU에 적합)

9.         init.kaiming\_uniform\_(m.weight, nonlinearity='relu')

10.         if m.bias is not None:

11.             init.zeros\_(m.bias)

12.

1. dropout\_rate = 0.2

2. model = MLP(input\_size, hidden\_size, num\_classes, dropout\_rate)

3. model.to(device)

4.

5. # Model Init

6. model.apply(init\_weights)

7.

8. # Model summary

9. from torchsummary import summary

10. summary(model, (1, 28 \* 28))

11.

12. # Loss and optimizer

13. criterion = nn.CrossEntropyLoss()

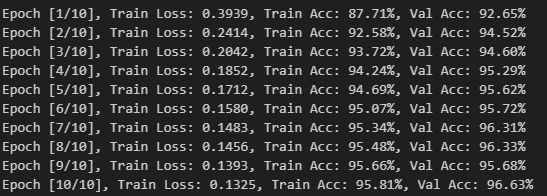
14. optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

15.

16.

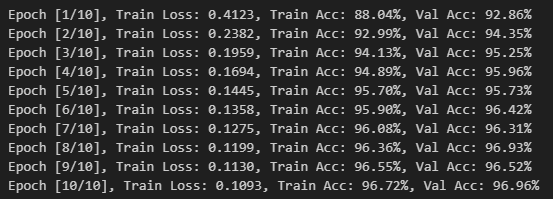
다음과 같은 코드를 추가하였다. 이제 각각의 경우에 따라 정확도가 어떻게 달라지는지 결과를 보도록 하겠다.

 HE초기화





 Xavier 초기화





## 4-1. 과적합(Overfitting) 현상 관찰하기

- 작은 학습 데이터셋 사용, epoch 수 증가, 모델복잡도(레이어 수) 증가, dropout 사용 x  
- 학습 정확도(손실)과 검증 정확도(손실)을 텐서 보드를 통해 관찰하여 과적합 현상 확인

우선 데이터 셋을 축소하기 위해 train\_dataset의 크기를 조정했다.

기존에는 train\_dataset 중 80%을 train으로 20%을 val로 사용하였다. 이번에는 이를 수정해 train\_dataset 크기를 1/4으로 줄이고, 그 중 80%를 train, 20%를 val로 사용하였다.

1. half\_size = len(train\_dataset) // 4 # 데이터 1/4만 사용하도록 크기 지정

2. half\_size\_train\_dataset, \_ = random\_split(train\_dataset, [half\_size, len(train\_dataset) - half\_size])

1. # Split the train set into train and validation

2. train\_size = int(0.8 \* len(half\_size\_train\_dataset))

3. val\_size = len(half\_size\_train\_dataset) - train\_size

4. train\_dataset, val\_dataset = random\_split(half\_size\_train\_dataset, [train\_size, val\_size])

1. train\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)

2. val\_loader = DataLoader(dataset=val\_dataset, batch\_size=batch\_size, shuffle=False)

3. test\_loader = DataLoader(dataset=test\_dataset, batch\_size=batch\_size, shuffle=False)

그 다음은 MLP 모델을 수정한다.

레이어를 많이 쌓으면 위에서 도출된 결과처럼 정확도가 떨어지는 것을 확인할 수 있다. 따라서, self.fc1 → self.fc2 → self.fc3 → self.fc4 → self.fc5 과정을 거치도록 모델을 구성한다.

1. # Define the MLP model

2. class MLP(nn.Module):

3.     def \_\_init\_\_(self, input\_size, hidden\_size, num\_classes, dropout\_rate):

4.         super(MLP, self).\_\_init\_\_()

5.         self.input\_size = input\_size

6.         self.fc1 = nn.Linear(input\_size, hidden\_size)

7.         self.sigmoid = nn.Sigmoid()

8.         self.dropout = nn.Dropout(dropout\_rate)

9.         self.fc2 = nn.Linear(hidden\_size, hidden\_size)

10.         self.fc3 = nn.Linear(hidden\_size, hidden\_size)

11.         self.fc4 = nn.Linear(hidden\_size, hidden\_size)

12.         self.fc5 = nn.Linear(hidden\_size, num\_classes)

13.

14.     def forward(self, x):

15.         x = x.view(-1, input\_size)

16.         x = self.fc1(x)

17.         x = self.sigmoid(x)

18.         x = self.fc2(x)

19.         x = self.sigmoid(x)

20.         x = self.fc3(x)

21.         x = self.sigmoid(x)

22.         x = self.fc4(x)

23.         x = self.sigmoid(x)

24.         x = self.fc5(x)

25.         return x

26.

torchSummary 사용 결과는 아래와 같다.

1. ----------------------------------------------------------------

2. Layer (type) Output Shape Param #

3. ================================================================

4. Linear-1 [-1, 256] 200,960

5. Sigmoid-2 [-1, 256] 0

6. Linear-3 [-1, 256] 65,792

7. Sigmoid-4 [-1, 256] 0

8. Linear-5 [-1, 256] 65,792

9. Sigmoid-6 [-1, 256] 0

10. Linear-7 [-1, 256] 65,792

11. Sigmoid-8 [-1, 256] 0

12. Linear-9 [-1, 10] 2,570

13. ================================================================

14. Total params: 400,906

15. Trainable params: 400,906

16. Non-trainable params: 0

17. ----------------------------------------------------------------

18. Input size (MB): 0.00

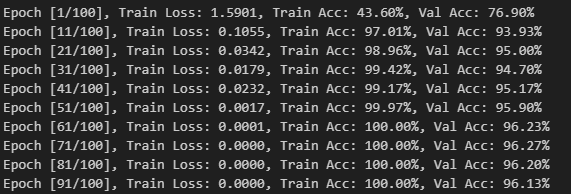
19. Forward/backward pass size (MB): 0.02

20. Params size (MB): 1.53

21. Estimated Total Size (MB): 1.55

22. ----------------------------------------------------------------

학습을 진행했을 때 학습 정확도는 아래와 같이 변화한다.





위 결과를 보면 train loss가 감소하면서 train\_acc는 100에 근접할 정도로 train이 이루어진 것을 확인할 수 있다. 하지만 val\_acc는 증가하는 추세를 보이다 정확도가 떨어지는 모습을 볼 수 있다. Train에 비하면 정확도도 그다지 높지 않다.

따라서 여기서 모델이 train에 과적합된 상태가 되었음을 확인할 수 있다.

TensorBoard 결과는 아래와 같다.

|  |  |
| --- | --- |
| 기존 모델 | 과적합 모델 |
|  |  |
|  | |

 대부분의 class에서 정확도가 떨어진 것을 확인할 수 있다.

-클래스 2, 7은 과적합 모델의 정확도가 조금 향상된 것을 볼 수 있다.

-클래스 3, 4, 6, 9는 성능이 하락하였으며, 특히 클래스 9는 클래스 4(22건)와의 혼동이  
크게 증가한 점을 확인할 수 있다.

과적합 모델은 기존과 다르게 오히려 일부 클래스 간 오분류가 증가하는 경향이 나타나는 것을 확인할 수 있다.

## 4-2. LMS에 있는 MLP 예제를 참고하여 기존 모델 성능을 개선

Hyper\_parameter setting

1. input\_size = 28 \* 28  # MNIST images are 28x28 pixels

2. batch\_size = 64

3. hidden\_size = 256

4. hidden\_size\_2 = 256

5. num\_classes = 10

6. dropout\_rate = 0.3

7. num\_epochs = 25

8. learning\_rate = 0.001

9.

위와 같이 파라미터 세팅을 한다.

MLP(Multi-Layer Perceptron) 모델은 다음과 같은 구조로 수정하였다.

1. class MLP(nn.Module):

2. def \_\_init\_\_(self, input\_size, hidden\_size, num\_classes, dropout\_rate):

3. super(MLP, self).\_\_init\_\_()

4. self.input\_size = input\_size

5. self.fc1 = nn.Linear(input\_size, hidden\_size)

6. self.sigmoid = nn.Sigmoid()

7. self.dropout = nn.Dropout(dropout\_rate)

8. self.fc2 = nn.Linear(hidden\_size, num\_classes)

9.

10. def forward(self, x):

11. x = x.view(-1, input\_size)

12. x = self.fc1(x)

13. x = self.sigmoid(x)

14. x = self.dropout(x)

15. x = self.fc2(x)

16. return x

17.

기존 MLP 구조에서 Fully Connected Layer(FC)를 세 개까지 늘려 학습을 시도했으나, 오히려 위의 결과처럼 정확도가 감소하는 현상이 발생하였다. 이에 따라 불필요한 복잡도를 줄이고 과적합을 방지하기 위해 FC 레이어의 수는 두 개(FC1, FC2)만 사용하였다.

또한 활성화 함수로는 기존의 ReLU 대신 출력의 범위를 [0, 1]로 제한하여 모델의 안정적인 학습을 유도하는 **Sigmoid** 함수를 적용하였다. 더불어 과적합을 방지하기 위해 첫 번째 FC 레이어 이후에 **Dropout**을 삽입하여 정규화를 수행하였다.

따라서 torchSummary에 따라 나오는 모델 구조는 아래와 같다.

----------------------------------------------------------------

Layer (type) Output Shape Param #

================================================================

Linear-1 [-1, 256] 200,960

Sigmoid-2 [-1, 256] 0

Dropout-3 [-1, 256] 0

Linear-4 [-1, 10] 2,570

================================================================

Total params: 203,530

Trainable params: 203,530

Non-trainable params: 0

----------------------------------------------------------------

Input size (MB): 0.00

Forward/backward pass size (MB): 0.01

Params size (MB): 0.78

Estimated Total Size (MB): 0.79

기존 epoch와 다르게 epoch를 증가시켰기 때문에, epoch%5==0일때에만 정확도를 출력한다.

1. # Track the best model

2. best\_val\_acc = 0

3. best\_model = None

4.

5. # Training loop

6. for epoch in range(num\_epochs):

7. model.train() #gradient 변경

8. #eval 모드에서는 가중치 변경 x

9. running\_loss = 0.0

10. correct = 0

11. total = 0

12.

13. for i, (images, labels) in enumerate(train\_loader):

14. # Forward pass

15. outputs = model(images.to(device))

16. loss = criterion(outputs, labels.to(device))

17.

18. # Backward pass and optimization

19. optimizer.zero\_grad() # 가중치 초기화

20. loss.backward() # 가중치 업데이트

21. optimizer.step()

22.

23. # Track statistics : 중간 점검

24. running\_loss += loss.item()

25. \_, predicted = torch.max(outputs.data, 1)

26. total += labels.size(0)

27. correct += (predicted == labels.to(device)).sum().item()

28.

29. #Add a few random images to TensorBoard along with their labels (every 100 steps)

30. if i % 100 == 0:

31. images\_to\_tensorboard(writer, images, labels, step=epoch \* len(train\_loader) + i)

32.

33. train\_acc = 100 \* correct / total

34. val\_acc = 0

35. val\_loss = 0.0

36. total = 0

37. correct = 0

38.

39. # Validation step

40. model.eval()

41. with torch.no\_grad():

42. for images, labels in val\_loader:

43. outputs = model(images.to(device))

44. loss = criterion(outputs, labels.to(device))

45. val\_loss += loss.item()

46. \_, predicted = torch.max(outputs.data, 1)

47. total += labels.size(0)

48. correct += (predicted == labels.to(device)).sum().item()

49.

50. val\_acc = 100 \* correct / total

51.

52. # Log to TensorBoard

53. writer.add\_scalar('Loss/train', running\_loss / len(train\_loader), epoch)

54. writer.add\_scalar('Accuracy/train', train\_acc, epoch)

55. writer.add\_scalar('Loss/validation', val\_loss / len(val\_loader), epoch)

56. writer.add\_scalar('Accuracy/validation', val\_acc, epoch)

57.

58. if (epoch%5 == 0):

59. print(f'Epoch [{epoch+1}/{num\_epochs}], Train Loss: {running\_loss / len(train\_loader):.4f}, '

60. f'Train Acc: {train\_acc:.2f}%, Val Acc: {val\_acc:.2f}%')

61.

62. # Save the best model

63. if val\_acc > best\_val\_acc:

64. best\_val\_acc = val\_acc

65. best\_model = model.state\_dict()

66.

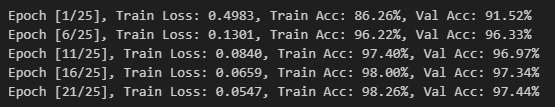
67. # Save the best model

68. torch.save(best\_model, 'best\_mlp\_model.pth')

69. writer.close()

70

 위 코드를 실행한 결과



  최종 정확도



위처럼 기존 97.37%에서 97.83%까지 정확도를 상승시킨 모습을 확인 할 수 있다.

 Tensorboard 결과

|  |  |
| --- | --- |
| 기존 모델 | 개선된 모델 |
|  |  |

위 그림은 기존 모델(왼쪽)과 개선된 모델(오른쪽)에 대한 Confusion Matrix를 나타낸 것이다.

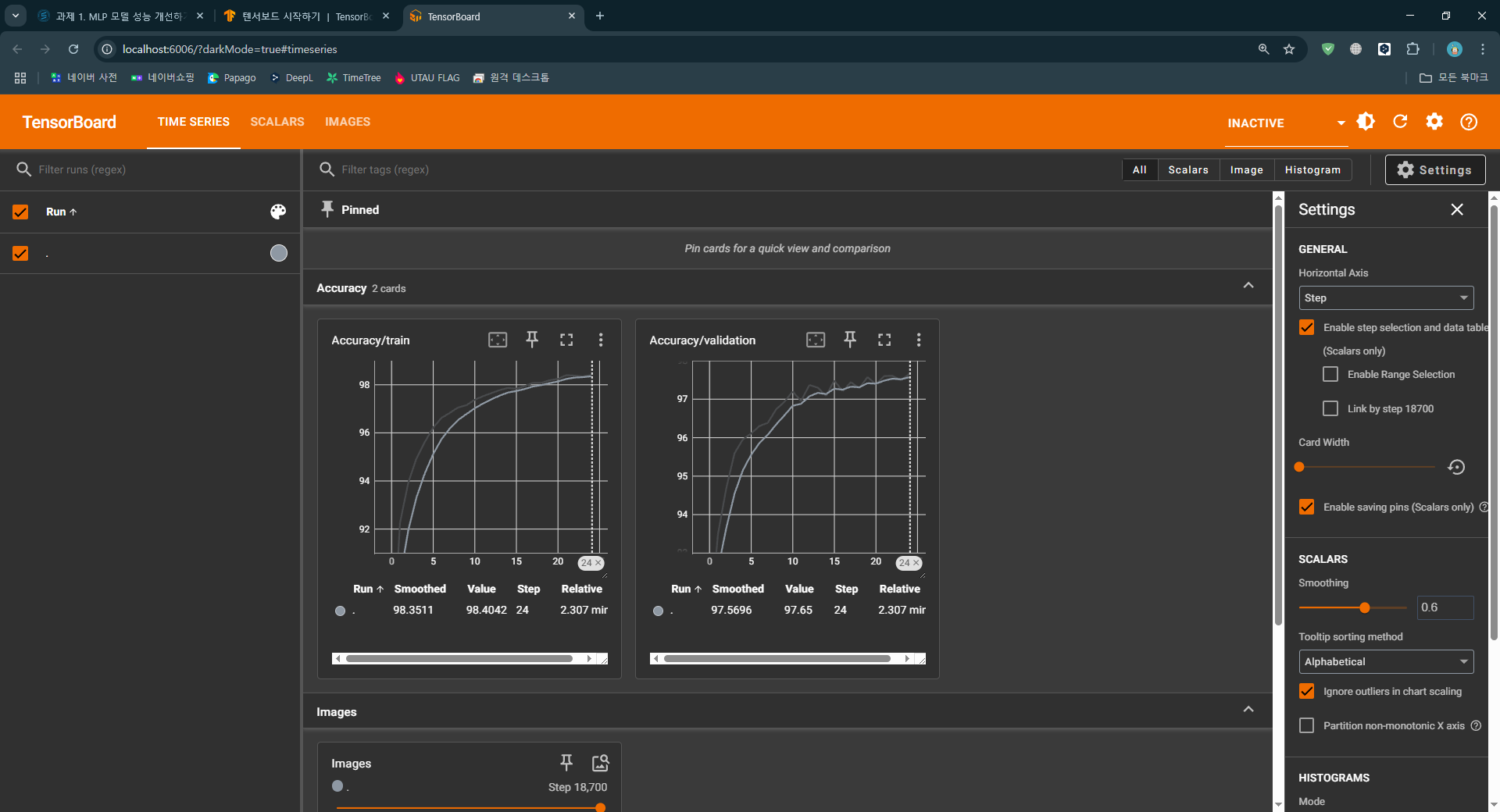
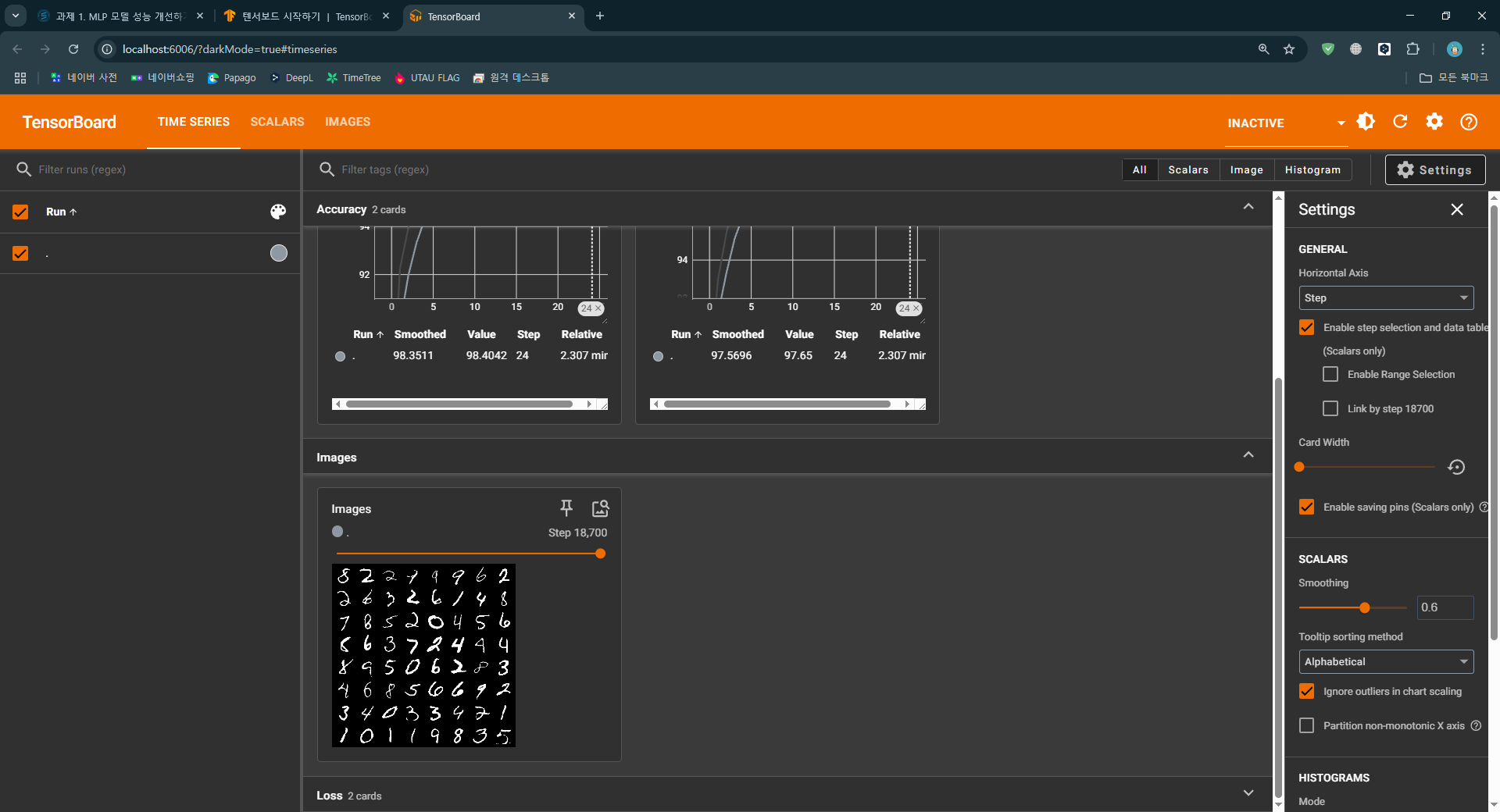
기존 모델 대비 개선된 모델은 전반적으로 클래스 간 예측 정확도가 향상된 양상을 보였다.

- 클래스 3, 4, 5, 8, 9번에서 True Positive 수치가 증가 => 정확한 분류 가능

- 10개 미만의 개수에서 차이를 보이는 class 또한 성능이 비슷한 수준을 유지하고 있어 과적합 없이 안정적으로  
 학습이 이루어졌음을 확인할 수 있다.

- 일부 클래스에서 정확도 저하가 나타나기도 했다.

약간 정확도가 저하된 클래스가 존재함에도, 대부분의 클래스를 제대로 분류해 분류 정확도 및 성능이 향상되었음을 확인할 수 있었다.

1, 2번 과제를 수행하며, 데이터 과적합 문제를 완화하고 모델의 정확도를 개선하기 위한 다양한 실험을 진행하였다.

초기 실험에서는 다양한 조건을 적용했음에도 기존 모델의 정확도인 97.37%를 넘기기 어려운 경우가 많았으나, 여러 튜닝을 거쳤을 때 최종적으로 정확도가 상승하는 것을 확인할 수 있었다. 특히, Epoch 수를 증가시키고, Dropout 비율을 적절히 조정하며, Optimizer를 변경하는 등의 튜닝을 거치니 정확도가 상승하는 것을 확인할 수 있었다.

여러 hyper parameter가 서로 상호 영향을 미치는 경우가 많기에 최적화를 하는 것이 상당히 어려운 일이라는 것을 알게 되었다. Hyperparameter Tuning이라고 부르는 활동을 해보면서  
튜닝을 더 체계적이고 효과적으로 수행하는 구체적인 기법에 대해 학습하고 싶다는 생각을 하게 되었다.