

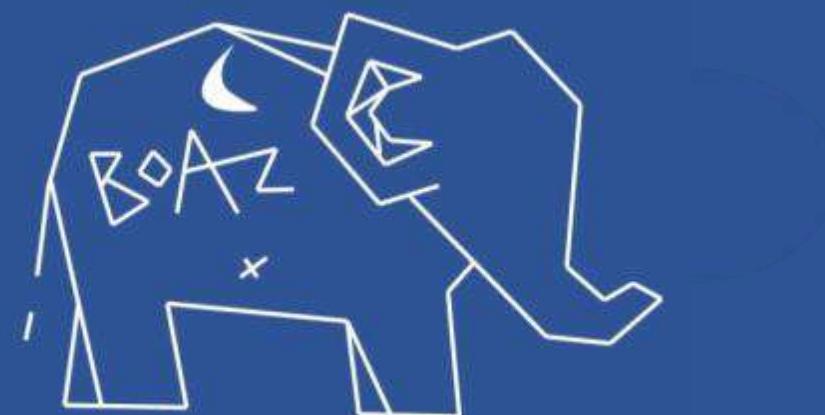
# 6주차 딥러닝 스터디

시각화 20기 노승혜  
시각화 20기 홍나연



8.1

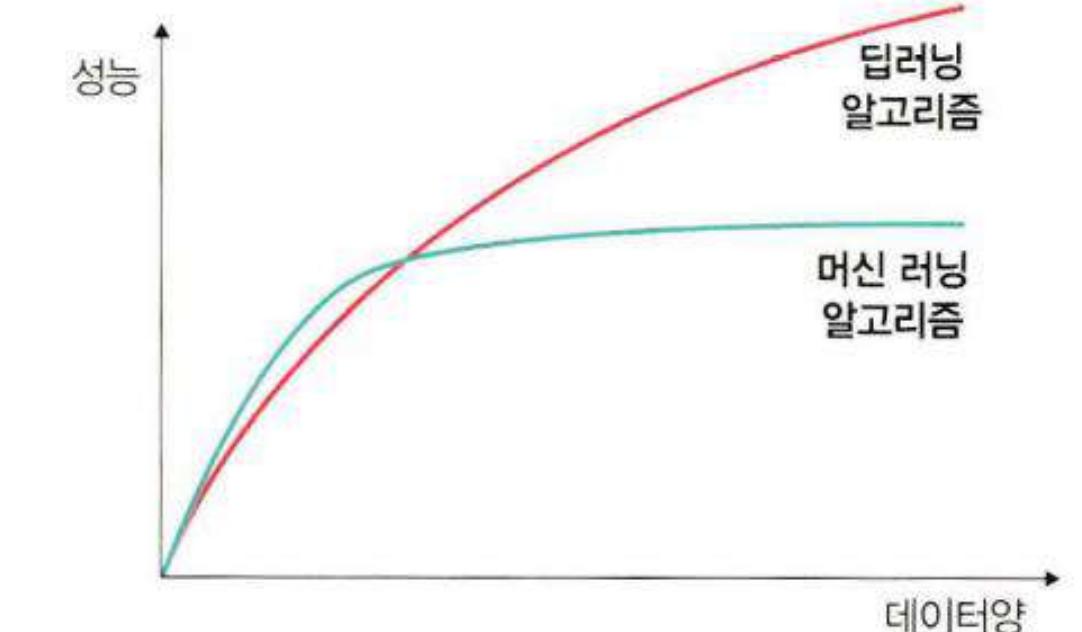
# 성능 최적화



### 8.1.1 데이터를 사용한 성능 최적화

## 데이터를 사용한 성능 최적화

- 최대한 많은 데이터 수집하기
  - 딥러닝 머신 러닝 알고리즘은 데이터양이 많을수록 성능이 좋음
- 데이터 생성하기
- 데이터 범위(scale) 조정하기
  - 활성화 함수로 시그모이드 사용 => 데이터셋 범위 0~1
  - 활성화 함수로 하이퍼볼릭 탄젠트 사용 : 데이터셋 범위 -1~1
  - 정규화, 규제화, 표준화도 성능 향상에 도움



<그림> 데이터와 딥러닝, 머신 러닝 알고리즘의 성능 비교

### 8.1.2 알고리즘을 이용한 성능 최적화

## 알고리즘을 이용한 성능 최적화

- 유사한 용도의 알고리즘들을 선택하여 모델을 훈련시켜 보고 최적의 성능을 보이는 알고리즘을 선택
- 데이터 분류
  - SVM, K-최근접 이웃 알고리즘들을 선택하여 훈련
- 시계열 데이터
  - RNN, LSTM, GRU 알고리즘 훈련

### 8.1.3 알고리즘 튜닝을 위한 성능 최적화

## 알고리즘 튜닝을 위한 성능 최적화 \_ 선택 하이퍼파라미터(1)

- 진단
  - 모델에 대한 평가를 사용해 문제 진단
  - Train 성능 >>> Test 성능 : 과적합 의심 => 규제화 진행
  - Train 성능 <<< Test 성능 : 과소적합 의심 => 네트워크 구조 변경 또는 에포크 수 조정
- 가중치
  - 초기값 : 작은 난수로 설정
  - 오토인코더 같은 비지도 학습을 이용하여 사전 훈련 후 지도 학습 진행

## 알고리즘 튜닝을 위한 성능 최적화 \_ 선택 하이퍼파라미터(2)

- 학습률
  - 초기값 : 매우 크거나 작은 난수
  - 네트워크 계층과 비례하게 설정 (ex. 네트워크 계층 多 => 학습률 높아야 함)
- 활성화 함수
  - 손실 함수도 함께 변경하는 경우 多
  - 시그모이드 / 하이퍼볼릭 탄젠트 사용한다면, 출력층에서는 소프트맥스 / 시그모이드 함수 많이 사용
- 배치와 에포크
  - 작은 배치와 큰 에포크 사용하는 게 트렌드

## 알고리즘 튜닝을 위한 성능 최적화 \_ 선택 하이퍼파라미터(3)

- 옵티마이저 및 손실 함수
  - 경사 하강법 많이 사용
  - 아담(Adam), RMSProp 도 좋은 성능 보임
- 네트워크 구성 (네트워크 토플로지)
  - 네트워크 구성을 변경해가며 성능 테스트 필요
  - 네트워크가 얕다 : 네트워크 계층 늘리되, 뉴런 개수
  - 네트워크가 넓다 : 하나의 은닉층에 여러 개의 뉴런
- 하이퍼파라미터에 대한 경우의 수를 모두 고려해야 하기 때문에 많은 모델 훈련이 필요

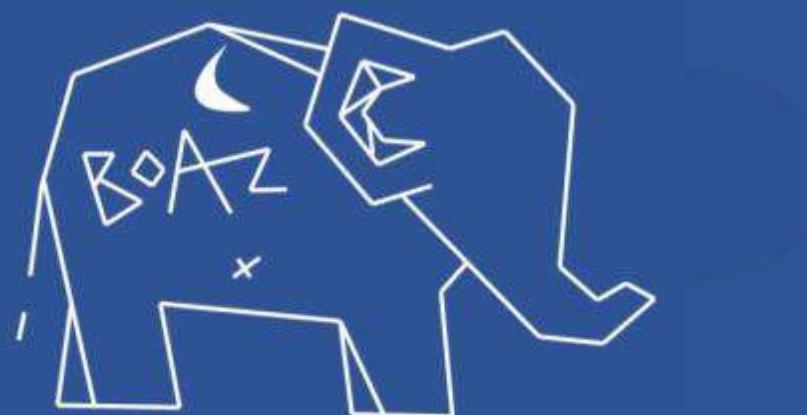
#### 8.1.4 양상블을 이용한 성능 최적화

## 양상블을 이용한 성능 최적화

- 양상블
- 두 개 이상의 모델을 섞어 사용
- ex. 랜덤 포레스트, 에이다부스트, 그래디언트 부스팅 ..
- 모델에 의해 발생하는 예측 오류를 감소할 수 있어 성능 최적화 방법 중 하나

## 8.2

# 하드웨어를 이용한 성능 최적화



### 8.2.1 CPU와 GPU 사용의 차이

데이터

알고리즘

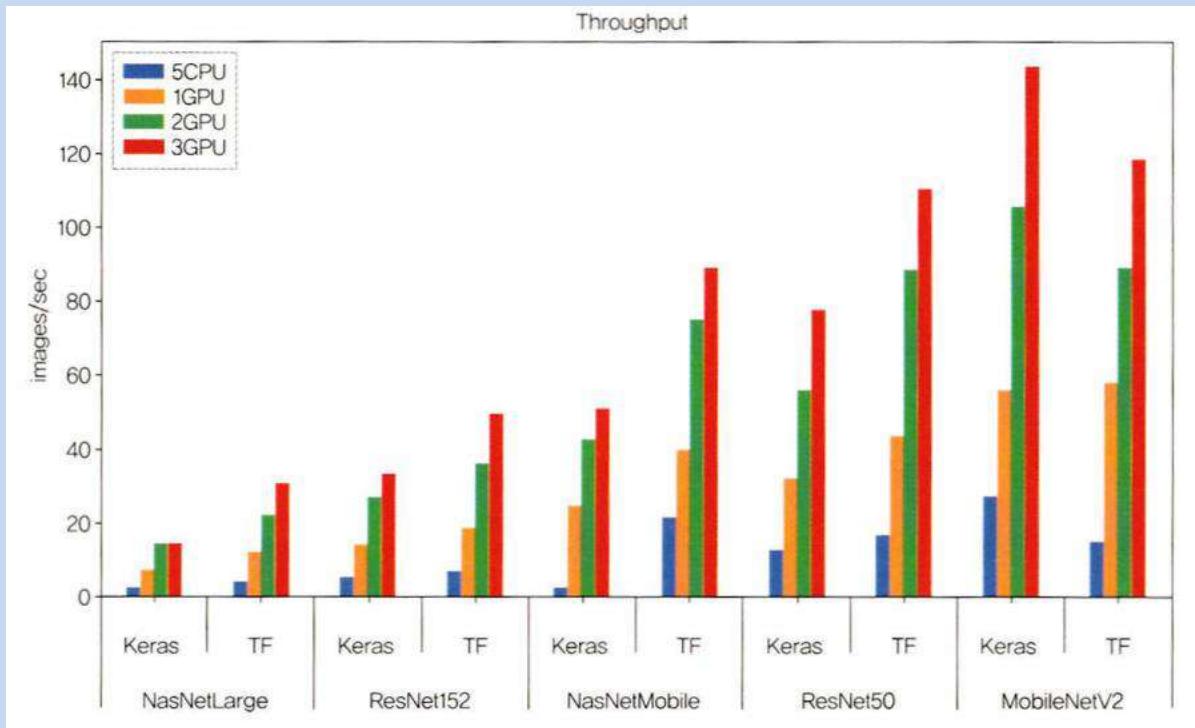
딥러닝에서의 성능 최적화

하드웨어

기존 CPU가 아닌 GPU를 이용

## 8.2.1 CPU와 GPU 사용의 차이

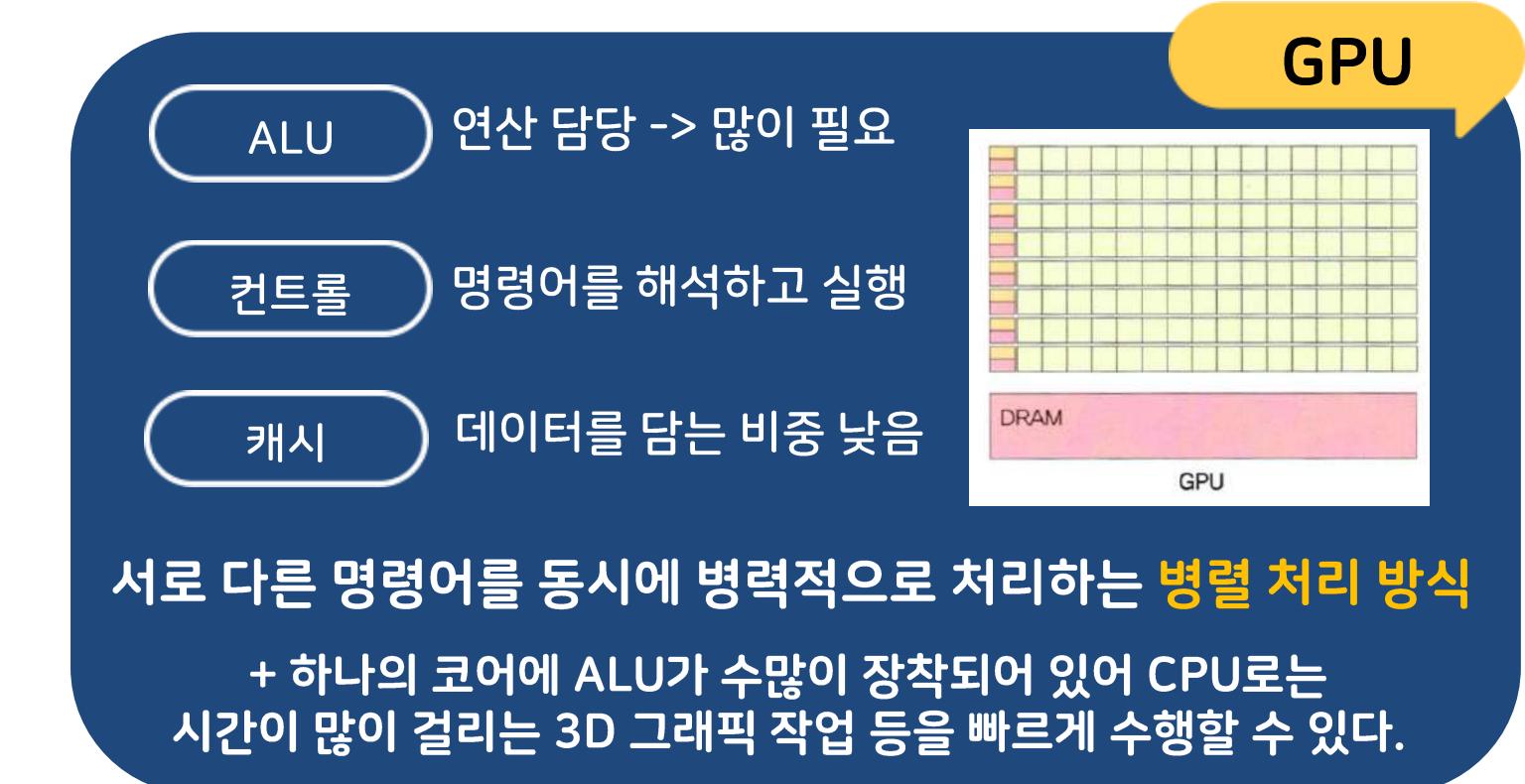
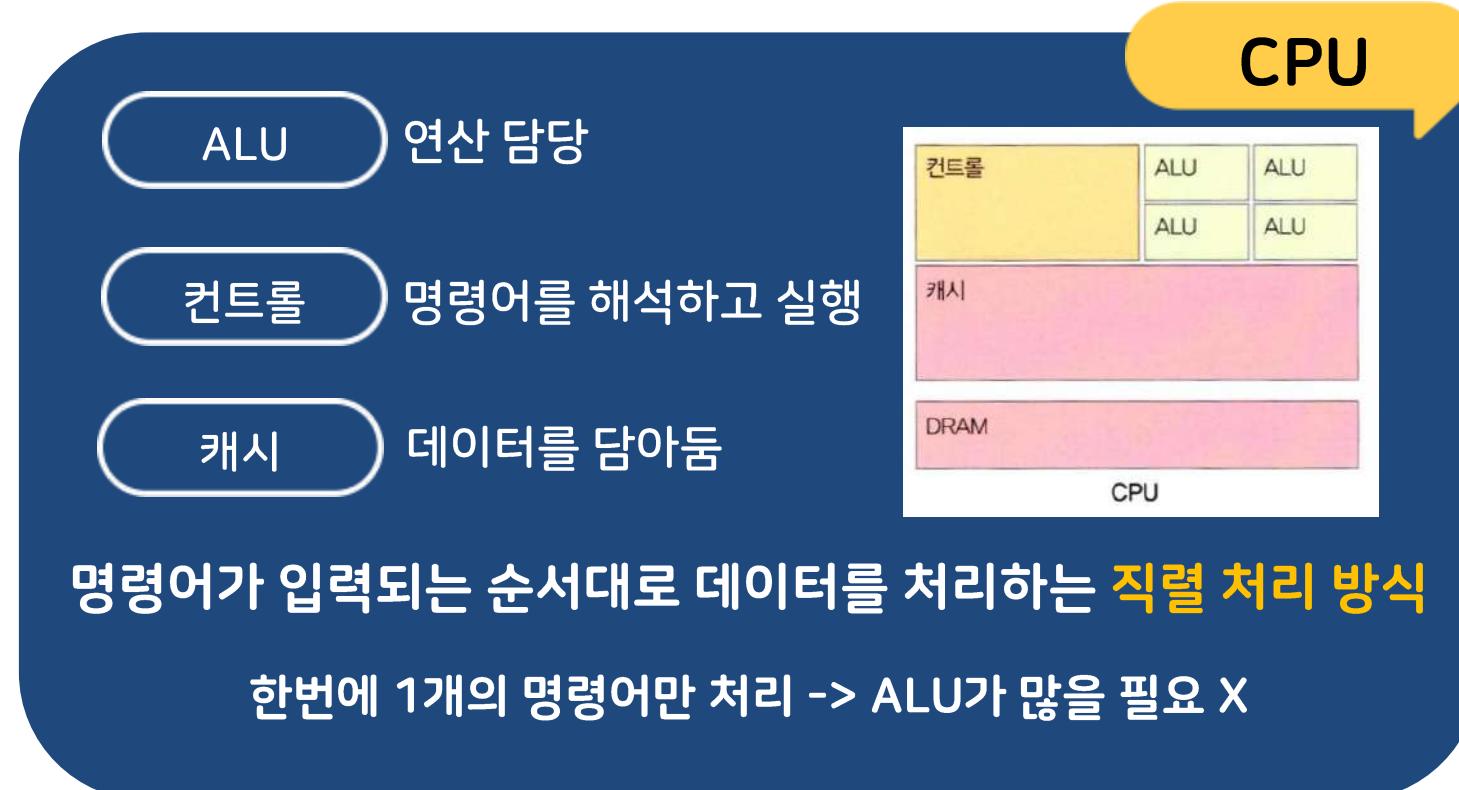
### GPU를 이용하는 이유



파란색 -> 5개의 CPU  
노란색 -> 1개의 GPU

### 차이가 왜 발생할까 ?

CPU와 GPU는 개발된 목적이 다르고, 그에 따라 내부 구조도 다르기 때문



### 8.2.1 CPU와 GPU 사용의 차이

## CPU, GPU 언제 사용하면 될까?

- 개별적 코어 속도 : CPU > GPU

CPU

파이썬이나 매트랩과 같이 행렬 연산을 많이 사용하는 재귀 연산

ex)  $3 \times 3$  행렬에서 A, B, C열이 있을 때 A열이 처리된 후에야  
B열이 처리되고 그 이후 C열이 처리되는 순차적 연산

GPU

역전파처럼 복잡한 미적분은 병렬 연산을 해야 속도가 빨라짐

-> A, B, C열을 얼마나 동시에 처리하느냐에 따라 계산속도가 달라지기 때문

병렬 처리는 복잡한 연산이 수반되는 딥러닝에서 속도와 성능을 높여주는 주요 요인

딥러닝은 데이터를 수백~수천만 건을 다루기 때문에  
(다룬다는 것은 데이터를 벡터로 변환 후 연산 수행)

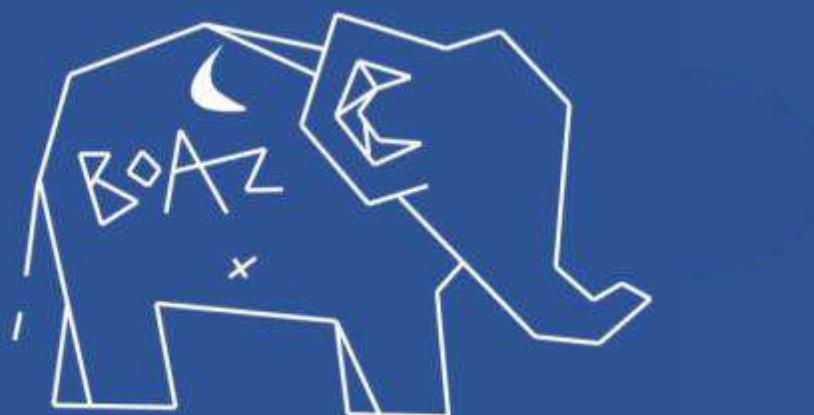
연산을 수행할 때 CPU에서 한번에 1개의 명령어만 처리한다면  
하나의 모델을 훈련시키는데 며칠~몇달까지도 걸릴 수 있음

-> GPU에서 병렬로 처리할 경우 모델 훈련 시간을 크게 단축시킬 수 있으므로

딥러닝에서 GPU 사용은 선택이 아닌 필수

8.3

## 하이퍼파라미터를 이용한 성능 최적화



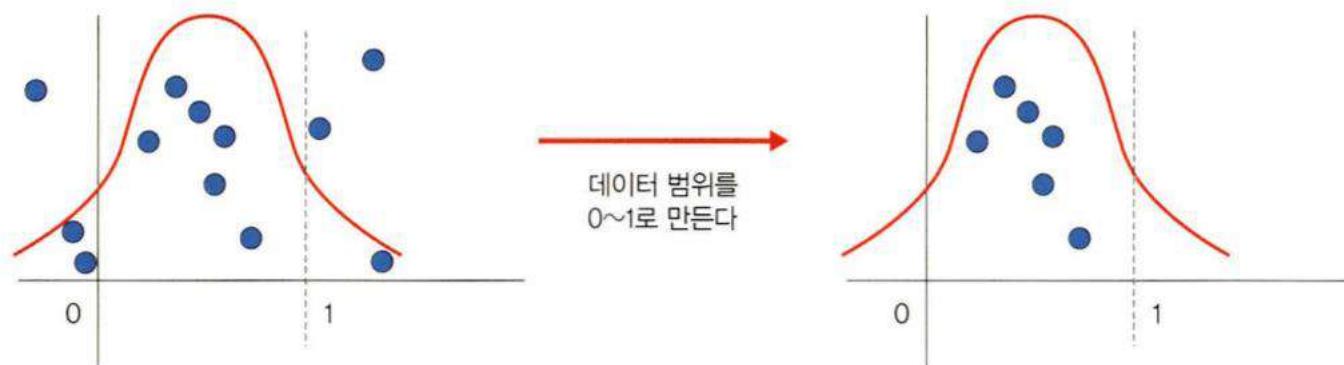
### 8.3.1 배치 정규화를 이용한 성능 최적화

#### 정규화

배치 정규화를 진행하기에 앞서 유사한 의미로 사용되는 용어

- 데이터 범위를 사용자가 원하는 범위로 제한하는 것을 의미

ex) 이미지 데이터의 픽셀 정보 = 0~255 사이의 값 -> 이를 255로 나누면 0~1 사이의 값을 갖게 된다.



- 각 특성 범위(스케일 | scale)를 조정한다는 의미로 특성 스케일링(feature scaling)이라고도 한다.  
-> 스케일 조정을 위한 기법 : MinMaxScaler()

$$\frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

( $x$ : 입력 데이터)

### 8.3.1 배치 정규화를 이용한 성능 최적화

#### 규제화

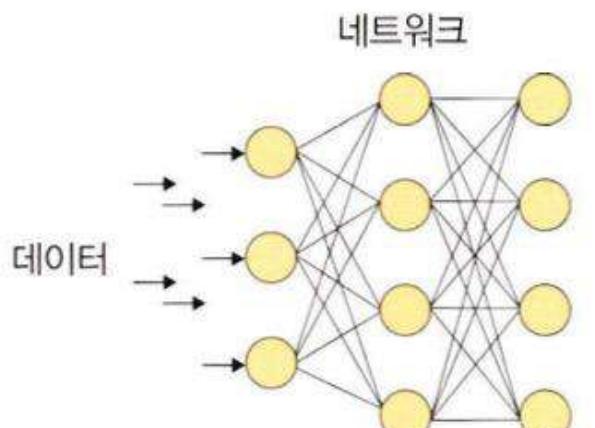
배치 정규화를 진행하기에 앞서 유사한 의미로 사용되는 용어

- 모델 복잡도를 줄이기 위해 제약을 두는 방법

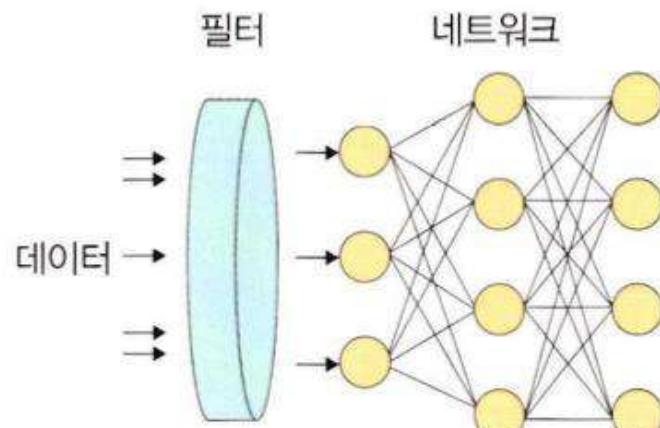
\* 제약 : 데이터가 네트워크에 들어가기 전에 필터를 적용한 것

ex) 왼쪽 그림은 필터가 적용되지 않은 경우 모든 데이터가 네트워크에 투입되는 것을 보여준다.

오른쪽 그림은 필터로 걸러진 데이터만 네트워크에 투입되어 빠르고 정확한 결과를 얻을 수 있다는 것을 보여준다.



필터가 적용되지 않은 경우



필터가 적용된 경우

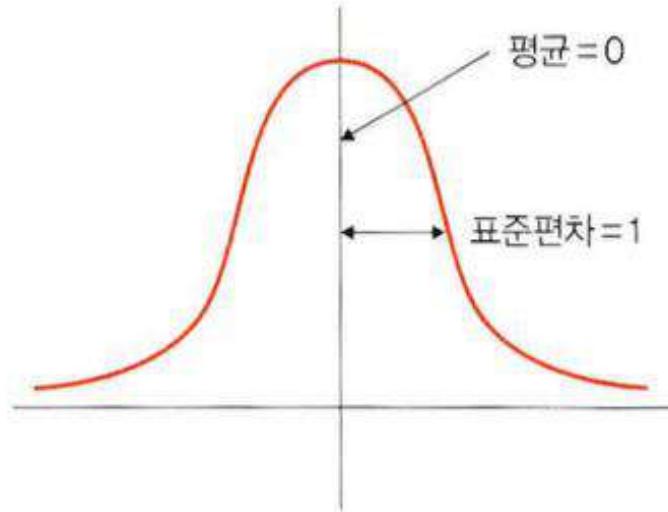
- 규제를 이용하여 모델 복잡도를 줄이는 방법
  1. 드롭아웃
  2. 조기 종료

### 8.3.1 배치 정규화를 이용한 성능 최적화

#### 표준화

배치 정규화를 진행하기에 앞서 유사한 의미로 사용되는 용어

- 기존 데이터를 평균 = 0, 표준편차 = 1인 형태의 데이터로 만드는 방법  
= 표준화 스칼라(standard scaler) = z - 스코어 정규화(z - score normalization)



- 평균을 기준으로 얼마나 떨어져 있는지 살펴볼 때 사용
- 보통 데이터 분포가 가우시안 분포를 따를 때 유용한 방법으로 다음과 같은 수식을 사용

$$\frac{x - m}{\sigma}$$

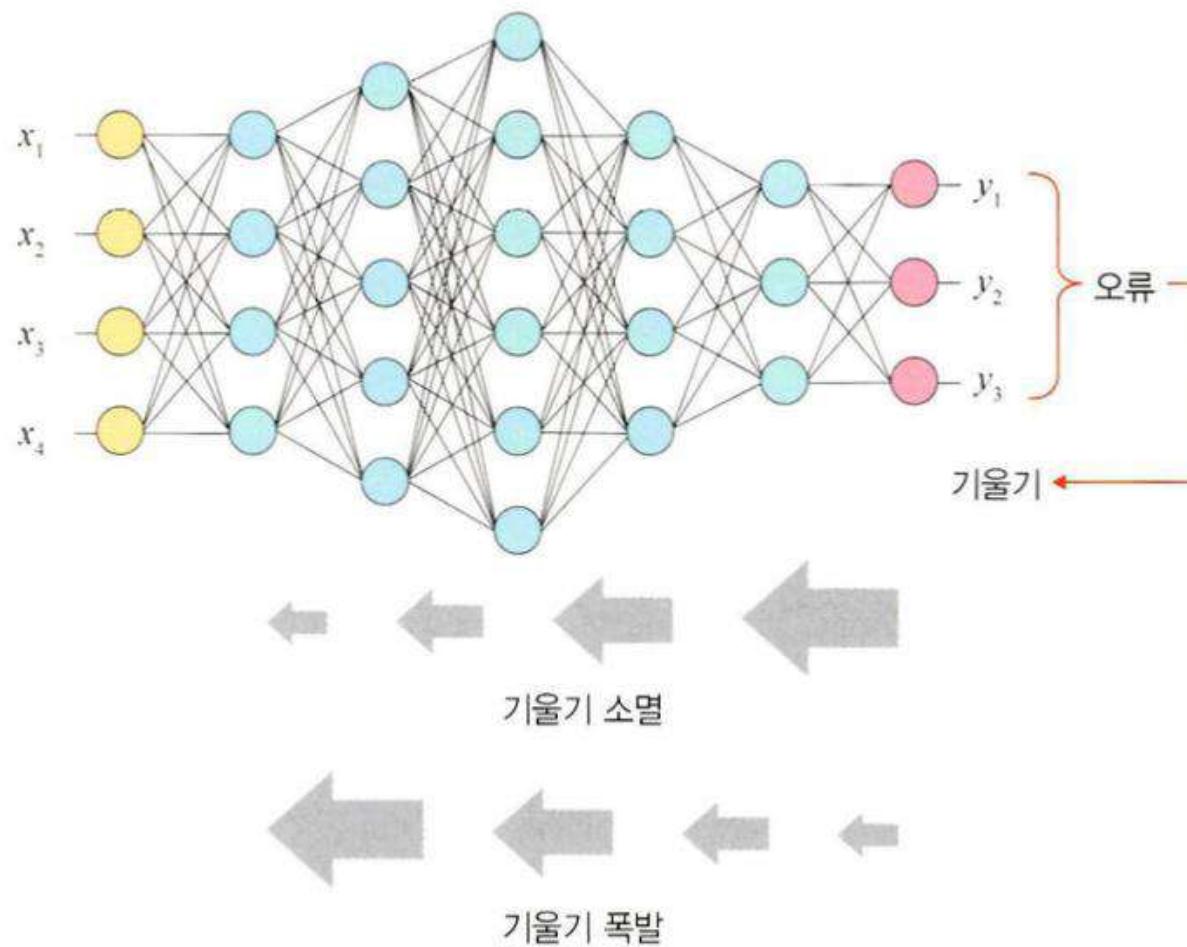
( $\sigma$ : 표준편차,  $x$ : 관측 값(입력 값),  $m$ : 평균)

### 8.3.1 배치 정규화를 이용한 성능 최적화

## 배치 정규화(batch normalization)

2015년 “Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift” 논문에 설명되어 있는  
기법

- 기울기 소멸(gradient vanishing) or 기울기 폭발(gradient exploding) 같은 문제를 해결하기 위한 방법
- 일반적으로 이러한 문제를 해결하기 위해 손실 함수로 렐루(ReLU)를 사용하거나 초깃값 튜닝, 학습률 등을 조정



기울기 소멸 : 오차 정보를 역전파시키는 과정에서 기울기가 급격히 0에 가까워져 학습이 되지 않는 현상

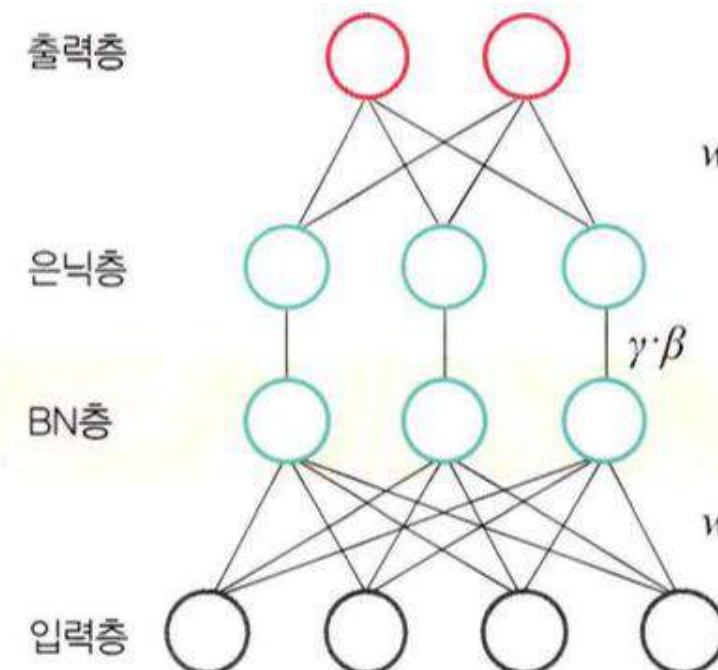
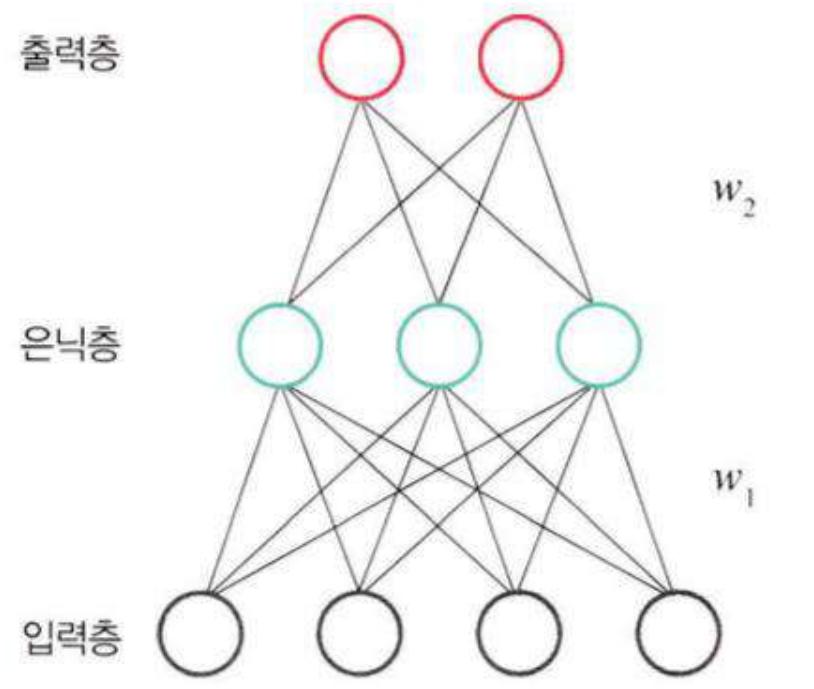
기울기 폭발 : 학습 과정에서 기울기가 급격히 커지는 현상

### 8.3.1 배치 정규화를 이용한 성능 최적화

## 배치 정규화(batch normalization)

2015년 “Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift” 논문에 설명되어 있는  
기법

- 기울기 소멸(or 폭발) 원인은 내부 공변량 변화  
-> 이것은 네트워크의 각 층마다 활성화 함수가 적용되면서 입력 값들의 분포가 계속 바뀌는 현상을 의미
- 분산된 분포를 정규분포로 만들기 위해 표준화와 유사한 방식으로 미니배치에 적용하여 평균 = 0, 표준편차 = 1로 유지하도록 한다.



$$\mu\beta \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad \dots \dots ①$$

$$\sigma^2 \beta \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu\beta)^2 \quad \dots \dots ②$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu\beta}{\sqrt{\sigma^2 \beta + \epsilon}} \quad \dots \dots ③$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \Leftrightarrow BN_{\gamma, \beta}(x_i) \quad \dots \dots ④$$

(  
입력:  $\beta = \{x_1, x_2, \dots, x_n\}$   
학습해야 할 하이퍼파라미터:  $\gamma, \beta$   
출력:  $y_i = BN_{\gamma, \beta}(x_i)$ )

① 미니 배치 평균을 구합니다.

② 미니 배치의 분산과 표준편차를 구합니다.

③ 정규화를 수행합니다.

④ 스케일(scale)을 조정(데이터 분포 조정)합니다.

### 8.3.1 배치 정규화를 이용한 성능 최적화

배치 정규화(batch normalization) 2015년 “Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift” 논문에 설명되어 있는 기법

- 매단계마다 활성화 함수를 거치면서 데이터셋 분포가 일정해지기 때문에 속도를 향상시킬 수 있지만 단점도 존재 !!

단점 1. 배치 크기가 작을 때는 정규화 값이 기존 값과 다른 방향으로 훈련될 가능성이 존재  
ex) 분산 = 0, 정규화 자체가 안되는 경우가 생길 수 있다.

단점 2. RNN은 네트워크 계층별로 미니 정규화를 적용해야 하기 때문에 모델이 더 복잡해지면서 비효율적일 수 있다.

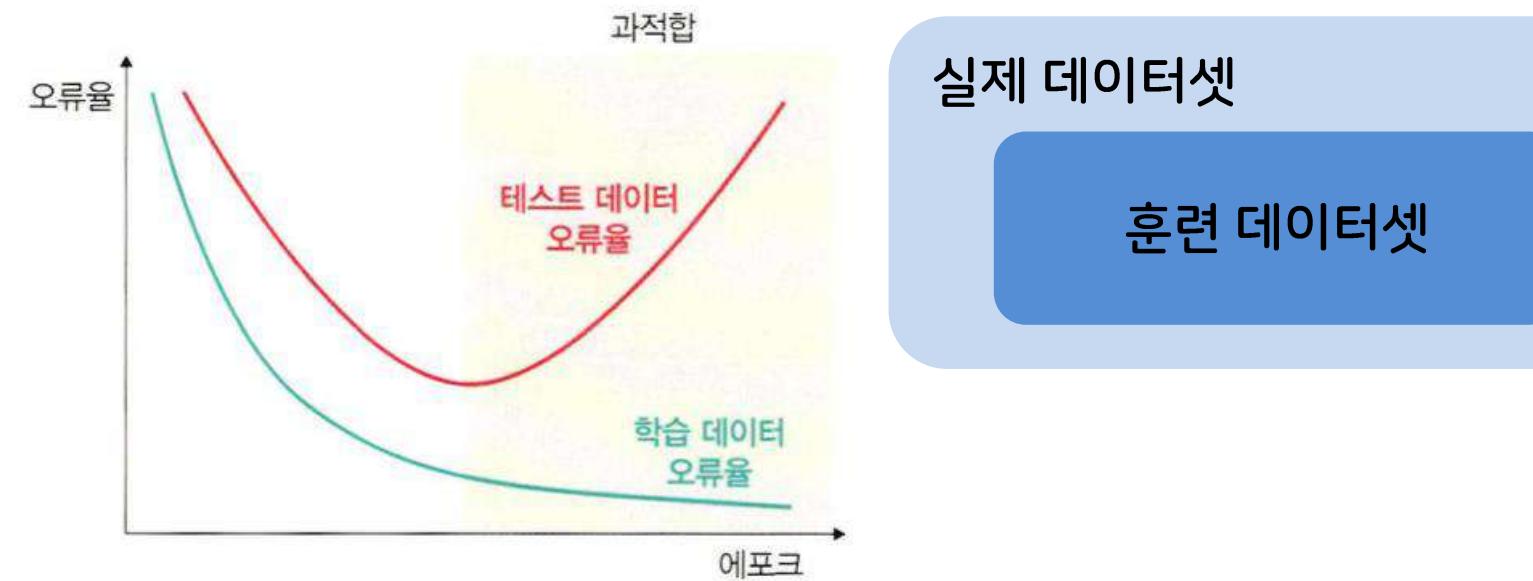
-> 이러한 문제를 해결하기 위해서는 가중치 수정, 네트워크 구성 변경 등을 수행

배치 정규화를 사용하면 적용 전보다 성능이 향상되어 많이 사용

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 과적합 문제

- 훈련 데이터셋을 과하게 학습하는 것을 의미 -> 왜 문제가 되나 ?

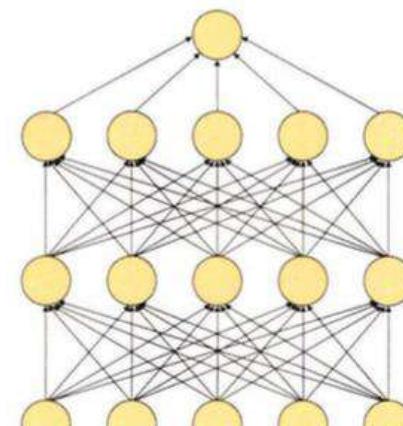


훈련 데이터셋에 대한 오류 감소 & 테스트 데이터셋에 대한 오류 증가  
즉, 훈련 데이터셋에 대해 훈련을 계속할 시 오류는 줄어들지만, 테스트 데이터셋에 대한 오류는 어느 순간부터 증가  
이러한 모델을 과적합되어있다고 한다.

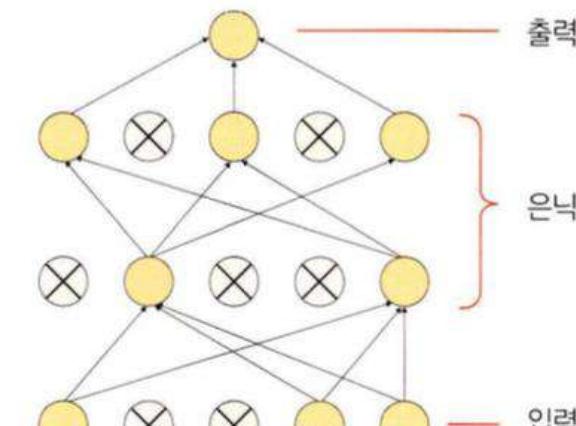
### 드롭 아웃

훈련할 때 일정 비율의 뉴런만 사용하고, 나머지 뉴런에 해당하는 가중치는 업데이트하지 않는 방법

노드를 임의로 끄면서 학습하는 방법으로, 은닉층에 배치된 노드 중 일부를 임의로 끄면서 학습  
꺼진 노드는 신호를 전달하지 않으므로 지나친 학습을 방지하는 효과 생성



일반적인 신경망



드롭아웃이  
적용된 신경망

- 일부 노드들은 비활성화 & 남은 노드들로 신호가 연결되는 신경망 형태
- 어떤 노드를 비활성화할지는 학습할 때마다 무작위로 선정
- 테스트 데이터로 평가 시 노드들을 모두 사용하여 출력하되 노드 삭제 비율(드롭아웃 비율)을 곱해서 성능 평가
- 단점 : 훈련 시간이 길어짐

그렇지만, 모델 성능을 향상하기 위해 상당히 자주 쓰는 방법

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 1. 먼저 필요한 라이브러리를 호출

#### 코드 8-1 라이브러리 호출

```
import torch
import matplotlib.pyplot as plt
import numpy as np

import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.optim as optim
```

### 2. 사용할 데이터셋 다운

#### 코드 8-2 데이터셋 내려받기

```
trainset = torchvision.datasets.FashionMNIST(
    root='..../chap08/data/', train=True,
    download=True,
    transform=transforms.ToTensor())
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 3. 내려받은 데이터셋을 메모리로 가져오기

#### 코드 8-3 데이터셋을 메모리로 가져오기

```
batch_size = 4  
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
```

batch\_size = 4 : 데이터를 메모리로 가져올 때 한 번에 네 개씩 쪼개서 가져온다.

### 4. 데이터셋을 이미지와 레이블로 분리 -> 학습을 위한 준비 단계

#### 코드 8-4 데이터셋 분리

```
dataiter = iter(trainloader)  
images, labels = dataiter.next()  
  
print(images.shape)  
print(images[0].shape)  
print(labels[0].item())
```

```
torch.Size([4, 1, 28, 28])  
torch.Size([1, 28, 28])  
9
```

**torch.Size[4, 1, 28, 28]**

4 : 한 번의 배치 크기로 몇 개의 데이터를 가져오는지 의미, batch size = 4 지정 -> 4 출력

1 : 채널을 의미, 흑백 이미지 = 1 / 컬러 이미지 = 3

28, 28 : 너비 X 높이로, 픽셀 크기의 이미지를 의미

### 8.3.2 드롭아웃을 이용한 성능 최적화

# 배치 정규화와 드롭아웃에 대한 파이토치 예제

## 5. 이미지 출력을 위해 데이터 형태를 바꿔주기 위한 전처리 함수 생성

### 코드 8-5 이미지 데이터를 출력하기 위한 전처리

```
def imshow(img, title):
    plt.figure(figsize=(batch_size * 4, 4)) ..... 출력할 개별 이미지의 크기 지정
    plt.axis('off')
    plt.imshow(np.transpose(img, (1, 2, 0))) ..... ① (1): 파이토치는
    plt.title(title)                                매플롯립(m
    plt.show()                                     + 변경을 하기 위
```

(1): 파이토치는 이미지 데이터셋을 [배치크기batch size, 채널channel, 너비width, 높이height] 순으로 저장  
     맷플롯립(matplotlib)으로 출력하기 위해선 이미지가 [너비, 높이, 채널]의 형태로 변경이 되어야 한다.  
+ 변경을 하기 위해 사용하는 것이 “ 넘파이 라이브러리의 transpose ”

## 6. 이미지 출력을 위한 그래프 방식 정의

#### 코드 8-6 이미지 데이터 출력 함수

### 8.3.2 드롭아웃을 이용한 성능 최적화

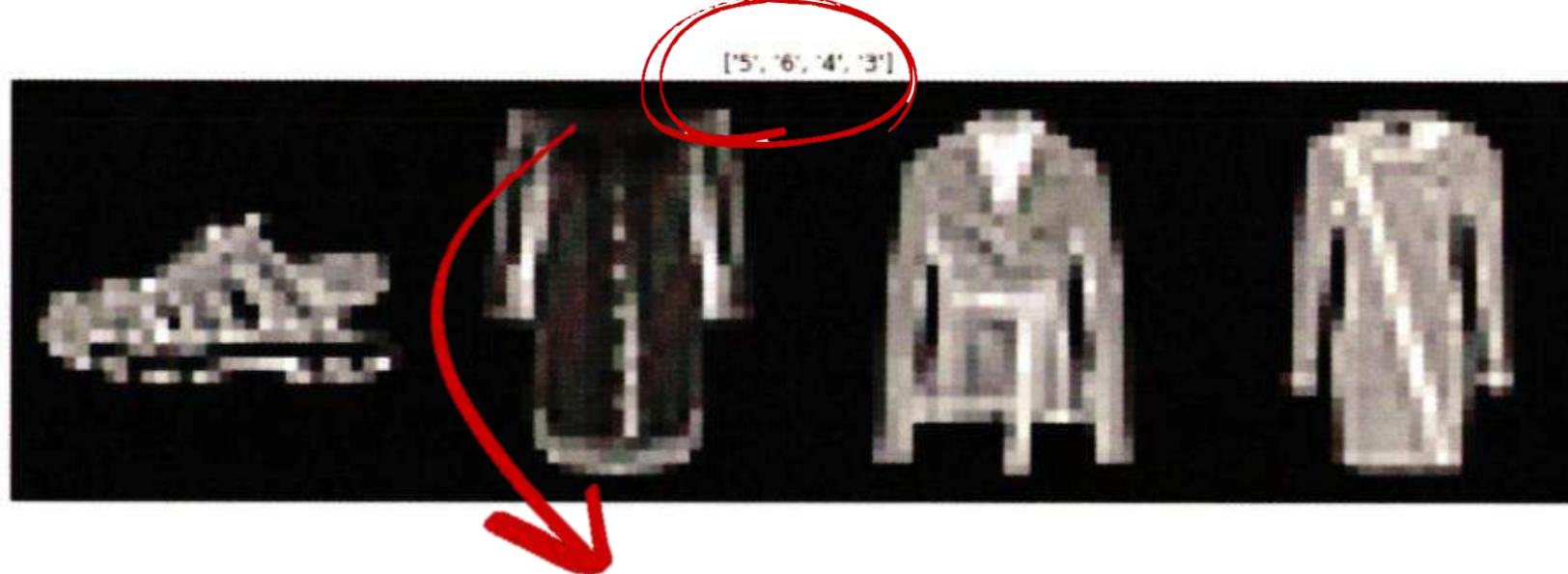
## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

## 7. 이미지 출력

코드 8-7 이미지 출력

```
images, labels = show_batch_images(trainloader)
```



클래스(레이블) 의미

즉, [‘5’, ‘6’, ‘4’, ‘3’]에서

5는 샌들, 6은 셔츠, 4는 코트, 3은 드레스를 의미

+ 4개의 이미지가 출력된 이유

: 한 번의 배치에서 4개의 이미지만 가져오도록 설정했기 때문 !!

```
classes = {  
    0: "T-Shirt/Top",  
    1: "Trouser",  
    2: "Pullover",  
    3: "Dress",  
    4: "Coat",  
    5: "Sandal",  
    6: "Shirt",  
    7: "Sneaker",  
    8: "Bag",  
    9: "Ankle Boot"  
}
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

## 8. 모델의 네트워크 구축

배치 정규화가 적용된 모델과 비교해보기 위해 배치 정규화가 적용되지 않는 모델을 먼저 생성

### 코드 8-8 배치 정규화가 적용되지 않은 네트워크

```
class NormalNet(nn.Module):
    def __init__(self):
        super(NormalNet, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(784, 48), .....(28, 28) 크기의 이미지로 입력은 784(28×28) 크기가 됩니다.
            nn.ReLU(),
            nn.Linear(48, 24),
            nn.ReLU(),
            nn.Linear(24, 10) .....FashionMNIST의 클래스는 총 열 개
        ) .....nn.Sequential을 사용하면 forward() 함수에서 계층(layer)별로 가독성 있게 코드 구현이 가능

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.classifier(x) .....nn.Sequential에서 정의한 계층 호출
        return x
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

## 9. 모델의 네트워크 구축

배치 정규화가 적용된 모델과 비교해보기 위해 이번에는 배치 정규화가 적용된 모델을 생성

코드 8-9 배치 정규화가 포함된 네트워크

```
class BNNet(nn.Module):
    def __init__(self):
        super(BNNet, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(784, 48),
            nn.BatchNorm1d(48), -----①
            nn.ReLU(),
            nn.Linear(48, 24),
            nn.BatchNorm1d(24),
            nn.ReLU(),
            nn.Linear(24, 10)
        )

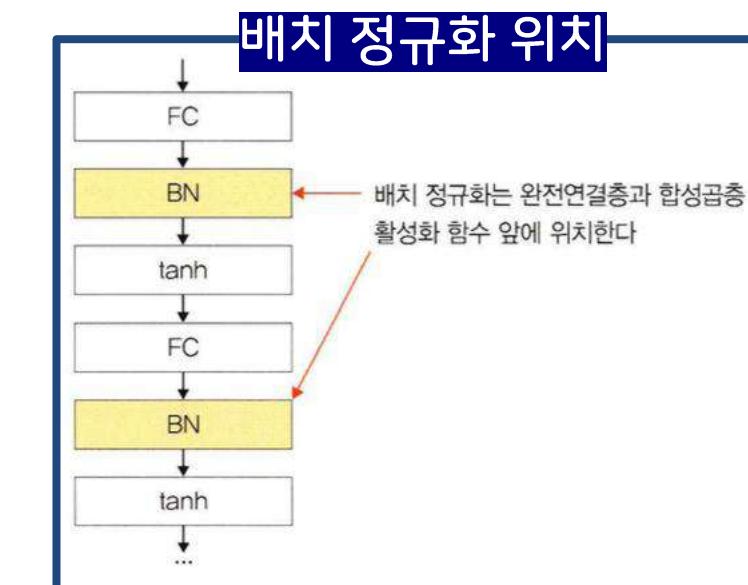
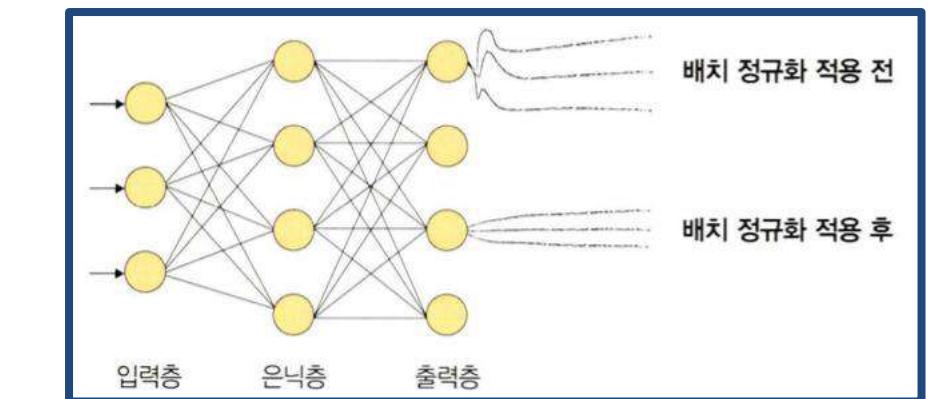
    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

### (1) : 배치 정규화가 적용되는 부분

BatchNorm1d에서 사용되는 파라미터 = 특성 개수로 이전 계층의 출력 채널

배치 정규화를 사용하는 이유 : 은닉층에서 학습이 진행될 때마다 입력 분포가 변하면서  
가중치가 엉뚱한 방향으로 갱신되는 문제가 종종 발생하기 때문

즉, 신경망의 층이 깊어질수록 학습할 때 가정했던 입력 분포가 변화하여 엉뚱한 학습이  
진행될 수 있는데 배치 정규화를 이용한다면 입력 분포를 고르게 맞출 수 있다.



### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 10. 배치 정규화가 적용되지 않은 모델 선언(객체화)

코드 8-10 배치 정규화가 적용되지 않은 모델 선언

```
model = NormalNet()  
print(model)
```

배치 정규화가 적용되지 않은 모델의 네트워크가 출력

```
NormalNet(  
    (classifier): Sequential(  
        (0): Linear(in_features=784, out_features=48, bias=True)  
        (1): ReLU()  
        (2): Linear(in_features=48, out_features=24, bias=True)  
        (3): ReLU()  
        (4): Linear(in_features=24, out_features=10, bias=True)  
    )  
)
```

### 11. 배치 정규화가 적용된 모델 선언(객체화)

코드 8-11 배치 정규화가 적용된 모델 선언

```
model_bn = BNNet()  
print(model_bn)
```

배치 정규화가 적용된 모델의 네트워크가 출력

```
BNNet(  
    (classifier): Sequential(  
        (0): Linear(in_features=784, out_features=48, bias=True)  
        (1): BatchNorm1d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (2): ReLU()  
        (3): Linear(in_features=48, out_features=24, bias=True)  
        (4): BatchNorm1d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (5): ReLU()  
        (6): Linear(in_features=24, out_features=10, bias=True)  
    )  
)
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 12. 데이터로더를 이용하여 데이터셋을 메모리로 불러올 준비

#### 코드 8-12 데이터셋 메모리로 불러오기

```
batch_size = 512  
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
```

- 이전(8-3)에 메모리로 불러오는 부분을 진행 / 이미지 출력을 위한 용도로 배치 크기 = 4 설정
- 이번에 메모리로 불러오는 것은 학습을 위한 용도 / 배치 크기 = 512 지정

### 13. 모델에서 사용할 옵티마이저와 손실 함수 지정

#### 코드 8-13 옵티마이저, 손실 함수 지정

```
loss_fn = nn.CrossEntropyLoss()  
opt = optim.SGD(model.parameters(), lr=0.01)  
opt_bn = optim.SGD(model_bn.parameters(), lr=0.01)
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 14. 모델 학습

코드 8-14 모델 학습

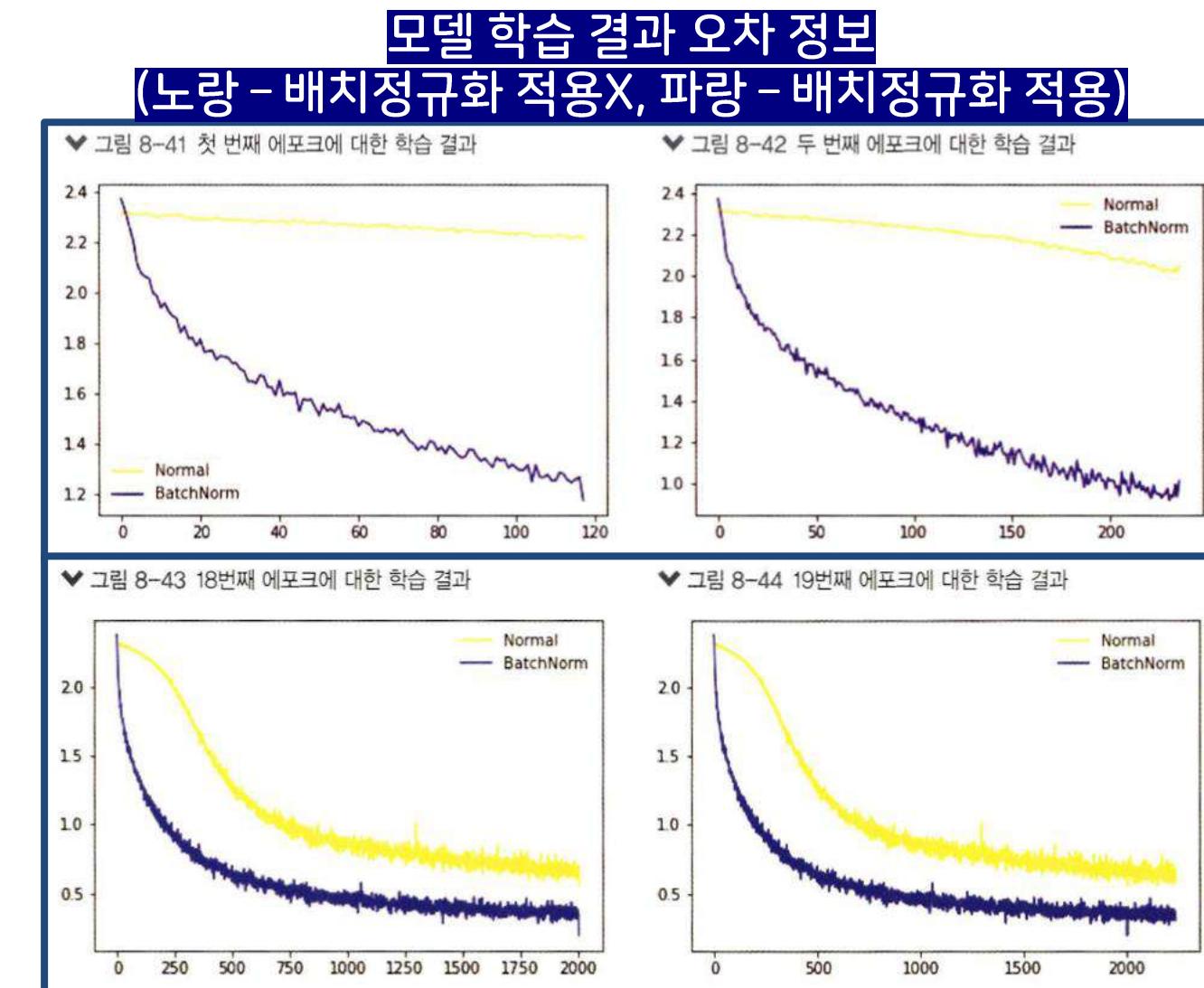
```
loss_arr = []
loss_bn_arr = []
max_epochs = 2

for epoch in range(max_epochs):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        opt.zero_grad() ..... 배치 정규화가 적용되지 않은 모델의 학습
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        opt.step()

        opt_bn.zero_grad() ..... 배치 정규화가 적용된 모델의 학습
        outputs_bn = model_bn(inputs)
        loss_bn = loss_fn(outputs_bn, labels)
        loss_bn.backward()
        opt_bn.step()

        loss_arr.append(loss.item())
        loss_bn_arr.append(loss_bn.item())

plt.plot(loss_arr, 'yellow', label='Normal')
plt.plot(loss_bn_arr, 'blue', label='BatchNorm')
plt.legend()
plt.show()
```



- 2개 모두 시간이 흐를수록 오차가 줄어드는 것을 확인
- 오차가 줄어드는 범위 &값의 차이는 명백히 발생
- 배치 정규화가 적용된 모델 : 낮은 값으로 안정적인 범위 내에서 감소

-> 배치 정규화가 적용된 모델은 에포크가 진행될수록 오차도 줄어들면서 안정적인 학습을 하고 있다.

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 1. 훈련과 테스트 데이터셋이 어떻게 분포하고 있는지 확인하는 단계

#### 코드 8-15 데이터셋의 분포를 출력하기 위한 전처리

```
N = 50
```

```
noise = 0.3
```

```
x_train = torch.unsqueeze(torch.linspace(-1, 1, N), 1) ..... ① 훈련 데이터셋이 -1~1의 값을 갖도록 조정
```

```
y_train = x_train + noise * torch.normal(torch.zeros(N, 1), torch.ones(N, 1)) ..... ② 훈련 데이터셋 값의 범위가 정규분포를 갖도록 조정
```

```
x_test = torch.unsqueeze(torch.linspace(-1, 1, N), 1)
```

```
y_test = x_test + noise * torch.normal(torch.zeros(N, 1), torch.ones(N, 1))
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 1. 훈련과 테스트 데이터셋이 어떻게 분포하고 있는지 확인하는 단계

#### (1) : 훈련 데이터셋이 -1 ~ 1의 값을 갖도록 조정

```
x_train = torch.unsqueeze(torch.linspace(-1, 1, N), 1)
```

ⓐ

ⓑ

(a) : torch.linspace: 주어진 범위에서 균등한 값을 갖는 텐서를 만들기 위해 사용

```
import torch
print(torch.linspace(0, 10)) ----- 0~10을 100으로 분할
print('-----')
print(torch.linspace(0, 10, 5)) ----- 0~10을 5로 분할
```

->

```
tensor([ 0.0000,  0.1010,  0.2020,  0.3030,  0.4040,  0.5051,  0.6061,  0.7071,  0.8081,
        0.9091,  1.0101,  1.1111,  1.2121,  1.3131,  1.4141,  1.5152,  1.6162,  1.7172,  1.8182,
        1.9192,  2.0202,  2.1212,  2.2222,  2.3232,  2.4242,  2.5253,  2.6263,  2.7273,  2.8283,
        2.9293,  3.0303,  3.1313,  3.2323,  3.3333,  3.4343,  3.5354,  3.6364,  3.7374,  3.8384,
        3.9394,  4.0404,  4.1414,  4.2424,  4.3434,  4.4444,  4.5455,  4.6465,  4.7475,  4.8485,
        4.9495,  5.0505,  5.1515,  5.2525,  5.3535,  5.4545,  5.5556,  5.6566,  5.7576,  5.8586,
        5.9596,  6.0606,  6.1616,  6.2626,  6.3636,  6.4646,  6.5657,  6.6667,  6.7677,  6.8687,
        6.9697,  7.0707,  7.1717,  7.2727,  7.3737,  7.4747,  7.5758,  7.6768,  7.7778,  7.8788,
        7.9798,  8.0808,  8.1818,  8.2828,  8.3838,  8.4848,  8.5859,  8.6869,  8.7879,  8.8889,
        8.9899,  9.0909,  9.1919,  9.2929,  9.3939,  9.4950,  9.5960,  9.6970,  9.7980,  9.8990,
        10.0000])
```

-----

```
tensor([ 0.0000,  2.5000,  5.0000,  7.5000, 10.0000])
```

torch.linspace(-1, 1, N) : -1 ~ 1의 범위에서 N개의 균등한 값을 갖는 텐서 생성

(b) : torch.unsqueeze : unsqueeze()는 차원을 늘리기 위해 사용

```
torch.unsqueeze(torch.linspace(-1, 1, N), 1)
: torch.linspace(-1, 1, N) 텐서의 첫 번째 자리에 차원을 증가시키겠다는 의미
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 1. 훈련과 테스트 데이터셋이 어떻게 분포하고 있는지 확인하는 단계

(2) : 훈련 데이터셋 값의 범위가 정규분포를 갖도록 조정

```
y_train = x_train + noise * torch.normal(torch.zeros(N, 1), torch.ones(N, 1))
```

ⓐ

ⓑ

ⓒ

(a) : `torch.normal` : 정규분포로부터 무작위 표본 추출을 위해 사용

`torch.Normal(평균, 표준편차)`를 의미

-> `torch.zeros()` = 평균, `torch.ones()` = 표준편차

기본값 : 평균 = 0, 표준편차 = 1

(b) : `torch.zeros()`: 0 값을 갖는 N X 1 텐서 생성

(c) : `torch.ones()`: 1 값을 갖는 N X 1 텐서 생성

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 2. 분포 확인

코드 8-16 데이터 분포를 그래프로 출력

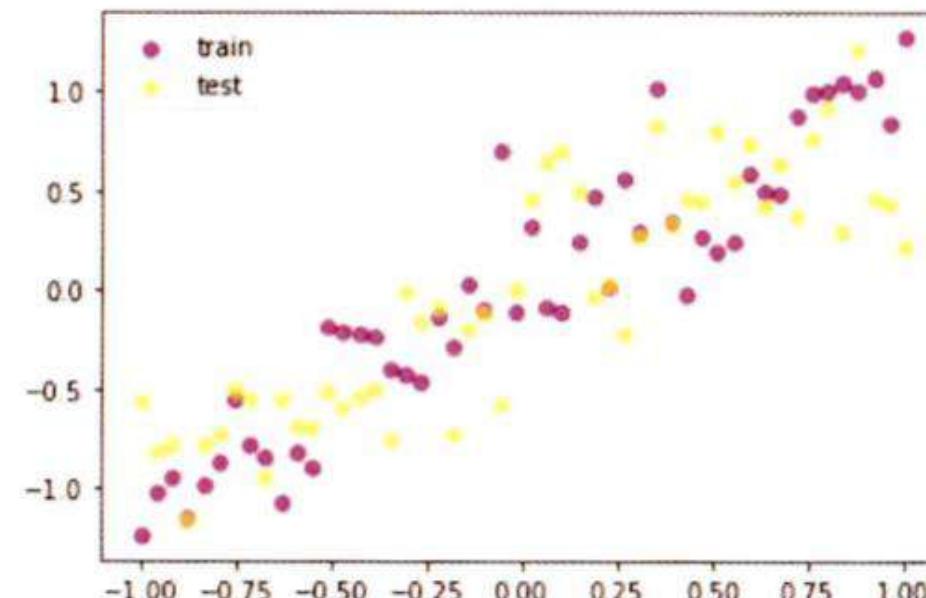
```
plt.scatter(x_train.data.numpy(), y_train.data.numpy(), c='purple',
            alpha=0.5, label='train') .....①
plt.scatter(x_test.data.numpy(), y_test.data.numpy(), c='yellow',
            alpha=0.5, label='test')
plt.legend()
plt.show()
```

```
plt.scatter(x_train.data.numpy(), y_train.data.numpy(), c='purple',
            alpha=0.5, label='train')
            ①           ②           ③
            ④           ⑤
```

(1) : plt.scatter()은 데이터를 그래프상에 점으로 출력해서 데이터 분포를 확인하고자 할 때 사용

- (a) : 첫번째 파라미터 = x축에 위치할 데이터
- (b) : 두번째 파라미터 = y축에 위치할 데이터
- (c) : C = 그래프로 출력되는 마커(그래프에서 작은 동그라미/점)의 색상
- (d) : Alpha = 마커에 대한 투명도를 조절하는 것으로 alpha=1 이면 완전 불투명 상태를 의미
- (e) : Label = 맷플롯립의 레전드와 같은 역할을 하지만 plt.legend()와 함께 사용되어야 한다.

### FashionMNIST 데이터셋 분포



-> 훈련, 테스트 데이터가 고르게 분포되어 있는 것을 확인

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 3. 모델 생성

드롭아웃의 효과를 확인하기 위해 드롭아웃이 적용된 것과 그렇지 않은 것의 모델을 생성

#### 코드 8-17 드롭아웃을 위한 모델 생성

```
N_h = 100
model = torch.nn.Sequential(
    torch.nn.Linear(1, N_h),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, N_h),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, 1),
) .....드롭아웃이 적용되지 않은 모델
```

```
model_dropout = torch.nn.Sequential(
    torch.nn.Linear(1, N_h),
    torch.nn.Dropout(0.2), .....드롭아웃 적용
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, N_h),
    torch.nn.Dropout(0.2),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, 1),
) .....드롭아웃이 적용된 모델
```

### 4. 손실 함수 지정

#### 코드 8-18 옵티마이저와 손실 함수 지정

```
opt = torch.optim.Adam(model.parameters(), lr=0.01)
opt_dropout = torch.optim.Adam(model_dropout.parameters(), lr=0.01)
loss_fn = torch.nn.MSELoss()
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 5. 모델 학습

드롭아웃이 적용된 모델과 그렇지 않은 모델을 학습시키고 오차를 그래프로 출력

코드 8-19 모델 학습

```
max_epochs = 1000
for epoch in range(max_epochs):
    pred = model(x_train) ..... 드롭아웃이 적용되지 않은 모델 학습
    loss = loss_fn(pred, y_train)
    opt.zero_grad()
    loss.backward()
    opt.step()

    pred_dropout = model_dropout(x_train) ..... 드롭아웃이 적용된 모델 학습
    loss_dropout = loss_fn(pred_dropout, y_train)
    opt_dropout.zero_grad()
    loss_dropout.backward()
    opt_dropout.step()

    if epoch % 50 == 0: ..... epoch를 50으로 나눈 나머지가 0이면 다음 진행
        model.eval()
        model_dropout.eval()

        test_pred = model(x_test)
        test_loss = loss_fn(test_pred, y_test)

        test_pred_dropout = model_dropout(x_test)
        test_loss_dropout = loss_fn(test_pred_dropout, y_test)

        plt.scatter(x_train.data.numpy(), y_train.data.numpy(), c='purple',
                    alpha=0.5, label='train')
        plt.scatter(x_test.data.numpy(), y_test.data.numpy(), c='yellow',
                    alpha=0.5, label='test')
        plt.plot(x_test.data.numpy(), test_pred.data.numpy(), 'b-', lw=3,
                 label='normal') ..... 파란색 실선으로 x축은 테스트 데이터셋, y축은 드롭아웃이
                               적용되지 않은 모델의 결과를 그래프로 출력

        plt.plot(x_test.data.numpy(), test_pred_dropout.data.numpy(), 'g--', lw=3,
                 label='dropout') ..... 초록색 점선으로 x축은 테스트 데이터셋, y축은 드롭아웃이
                               적용된 모델의 결과를 그래프로 출력

        plt.title('Epoch %d, Loss=%0.4f, Loss with dropout=%0.4f' %
                  (epoch, test_loss, test_loss_dropout)) ..... 에포크, 드롭아웃이 적용되지 않은
                                                     모델의 오차, 드롭아웃이 적용된
                                                     모델의 오차를 타이틀로 출력
        plt.legend()
        model.train()
        model_dropout.train()
        plt.pause(0.05)
```

### 8.3.2 드롭아웃을 이용한 성능 최적화

## 배치 정규화와 드롭아웃에 대한 파이토치 예제

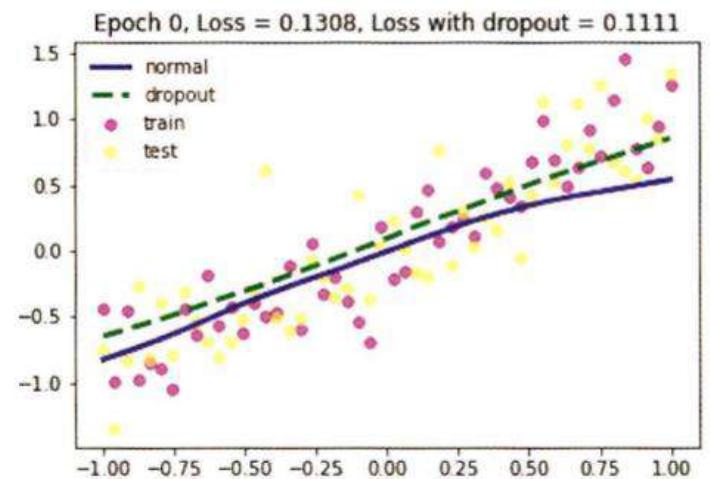
파이토치 torchvision.datasets에서 제공하는 FashionMNIST 데이터셋

### 5. 모델 학습

드롭아웃이 적용된 모델과 그렇지 않은 모델을 학습시키고 오차를 그래프로 출력

#### 드롭아웃과 관련된 모델 학습 결과

그림 8-46 첫 번째 드롭아웃에 대한 학습 결과



… 중간 생략 …

그림 8-47 두 번째 드롭아웃에 대한 학습 결과

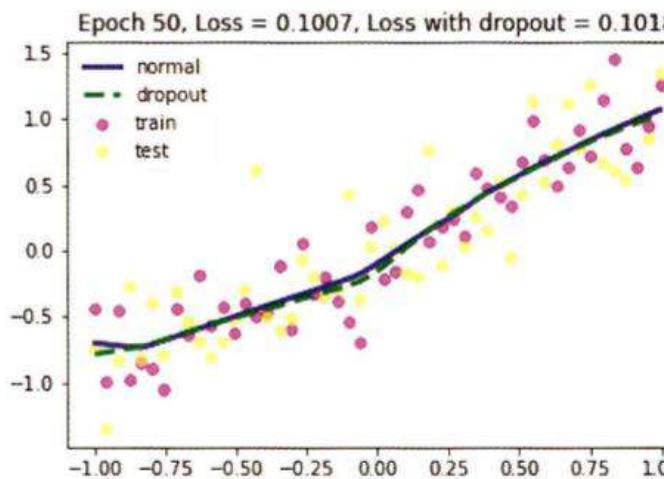


그림 8-48 19번째 드롭아웃에 대한 학습 결과

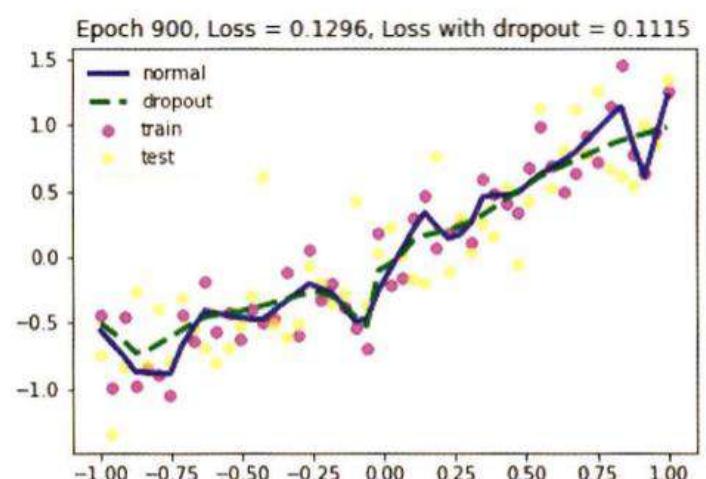
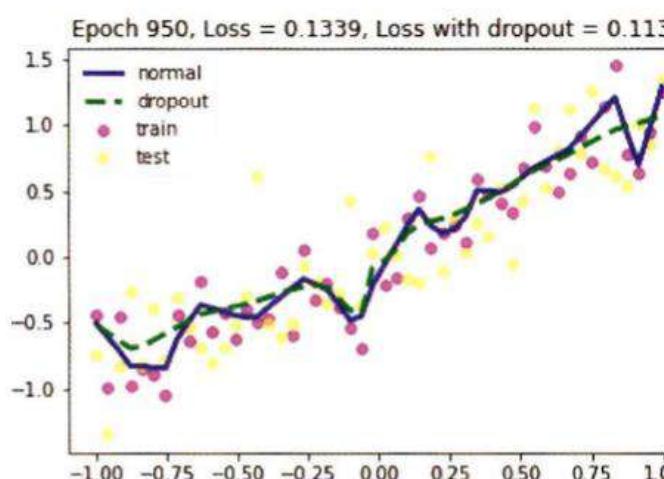


그림 8-49 20번째 드롭아웃에 대한 학습 결과

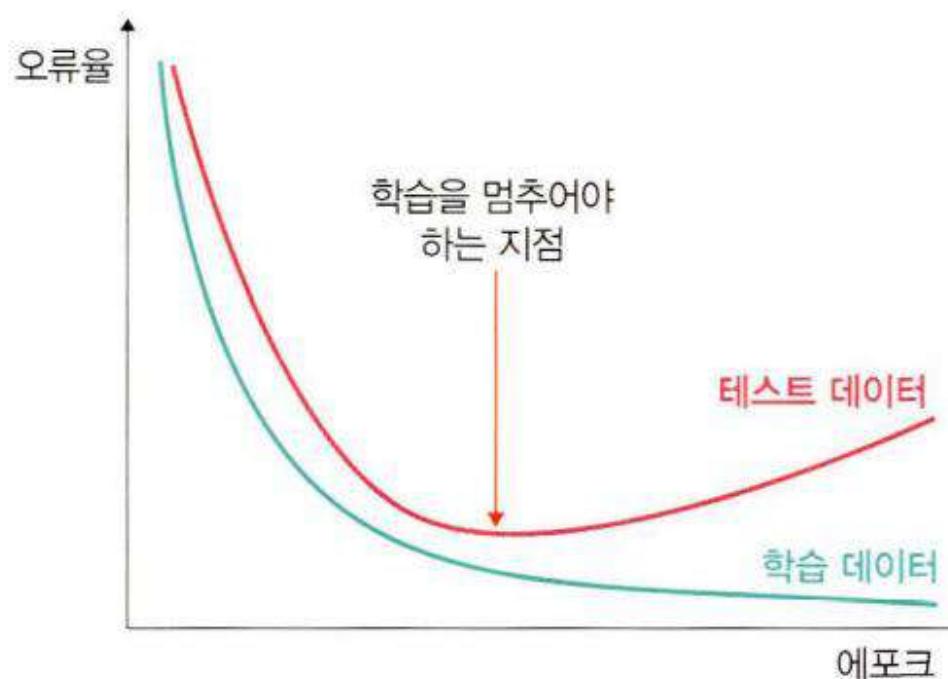


- 전반적으로 오차가 줄어드는 범위는 크지 않지만, 드롭아웃 적용 시 오차가 더 낮은 것을 확인
- 초록선(= 드롭아웃이 적용된 모델)과 파란선(= 드롭아웃이 적용되지 않은 모델)의 차이가 크지 않아 보일 수 있지만 사실 실제로는 큰 차이가 있는 상태.
- 훈련 횟수가 늘어날수록 파란선은 가장자리의 자주색 점들을 찾아가고 있다.  
→ 문제는 자주색 선이 훈련 데이터셋 / 즉, 과적합 현상이 나타나는 것으로 볼 수 있다.
- 과적합이 발생하는 모델 : 훈련 데이터에 대한 정확도는 높을 수 있지만, 새로운 데이터(검증 & 테스트 데이터)에 대해 잘 동작하지 않는 문제 발생  
→ 이와 같이 과적합 현상을 방지하기 위해 드롭아웃을 사용  
= 초록선에서는 과적합 현상이 발생하지 않는 것을 확인

### 8.3.3 조기 종료를 이용한 성능 최적화

## 조기 종료를 이용한 성능 최적화

- 조기 종료
  - 뉴럴 네트워크가 과적합을 회피하는 규제 기법
  - 매 에포크마다 검증 데이터에 대한 오차를 측정하여 모델의 종료 시점 제어 (최고의 성능 모델 보장 X)
  - 과적합 발생 시, 훈련 데이터셋에 대한 오차 감소 & 검증 데이터셋에 대한 오차 증가
  - 검증 데이터셋에 대한 오차가 증가하는 시점에 학습 종료



### 8.3.3 조기 종료를 이용한 성능 최적화

## 1단계 | 필요 라이브러리 호출

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models ..... 사전 학습된 모델을 이용하고자 할 때 사용하는 라이브러리
from torchvision import transforms, datasets

import matplotlib
import matplotlib.pyplot as plt
import time

import argparse
from tqdm import tqdm
matplotlib.style.use('ggplot') ..... 출력 그래프에서 격자로 숫자 범위가 눈에 잘 띄도록 하는 스타일
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

## 2단계 | 전처리를 위한 항목 정의

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])
```

- 데이터 크기 조정 및 데이터 정규화 (분포 조정)
- 데이터셋은 핫도그와 핫도그가 아닌 이미지 이용

### 8.3.3 조기 종료를 이용한 성능 최적화

#### 3단계 | 데이터셋 가져오기

```
train_dataset = datasets.ImageFolder(  
    root=r'../chap08/data/archive/train',  
    transform=train_transform  
)  
  
train_dataloader = torch.utils.data.DataLoader(  
    train_dataset, batch_size=32, shuffle=True,  
)  
  
val_dataset = datasets.ImageFolder(  
    root=r'../chap08/data/archive/test',  
    transform=val_transform  
)  
  
val_dataloader = torch.utils.data.DataLoader(  
    val_dataset, batch_size=32, shuffle=False,  
)
```

- 데이터셋을 배치 크기로 메모리로 가져올 준비
- 일반적으로 PC에서 사용하는 메모리 용량(16~24GB) 고려
- 한 번에 32개의 이미지 가져오도록 batch\_size = 32로 설정

#### 4단계 | 모델 생성

```
def resnet50(pretrained=True, requires_grad=False):  
    model = models.resnet50(progress=True, pretrained=pretrained)  
    if requires_grad == False: ----- 파라미터를 고정하여 backward() 중에 기울기가 계산되지 않도록 합니다.  
        for param in model.parameters(): requires_grad=False를 파라미터로 받았기 때문에 해당 구문이 실행됩니다.  
            param.requires_grad = False  
    elif requires_grad == True: ----- 파라미터 값이 backward() 중에 기울기 계산에 반영됩니다.  
        for param in model.parameters():  
            param.requires_grad = True  
    model.fc = nn.Linear(2048, 2) ----- 마지막 분류를 위한 계층은 학습을 진행합니다.  
    return model
```

- 사전 학습된 ResNet50을 사용하여 네트워크 구축

### 8.3.3 조기 종료를 이용한 성능 최적화

## 5단계 | 학습률 감소

```
class LRScheduler():
    def __init__(  
        self, optimizer, patience=5, min_lr=1e-6, factor=0.5  
    ):  
        self.optimizer = optimizer  
        self.patience = patience  
        self.min_lr = min_lr  
        self.factor = factor  
        self.lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(  
            self.optimizer,  
            mode='min',  
            patience=self.patience,  
            factor=self.factor,  
            min_lr=self.min_lr,  
            verbose=True  
        ) .....①  
  
    def __call__(self, val_loss):  
        self.lr_scheduler.step(val_loss) .....②
```

- 학습률 감소

- 학습이 진행되는 과정에서 학습률을 조금씩 낮추는 성능 튜닝 기법
- 'patience' 횟수 만큼 검증 데이터셋에 대한 오차 감소가 없으면 'factor' 만큼 학습률 감소

학습률 감소

모델 성능 개선이 없을 경우,  
학습률 값 조절

실제로 학습률 업데이트

### 8.3.3 조기 종료를 이용한 성능 최적화

## 5단계 | 학습률 감소

```
torch.optim.lr_scheduler.ReduceLROnPlateau(self.optimizer, mode='min', patience=self.patience, factor=self.factor, min_lr=self.min_lr, verbose=True)
```

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨

- ⓐ ReduceLROnPlateau는 검증 데이터셋에 대한 오차의 변동이 없으면 학습률을 factor배로 감소
- ⓑ 파라미터(가중치)를 갱신시키는 부분, 여기에서는 아담(optim.Adam)을 사용
- ⓒ 언제 학습률을 조정할 지에 대한 기준값 : min과 max, auto를 값으로 가짐
  - 학습률 조정의 기준이 정확도(val\_acc) : 값이 클수록 좋으므로 max로 지정
  - 학습률 조정의 기준이 오차(val\_loss) : 값이 작을수록 좋으므로 min으로 지정
  - 값이 더 이상 증/감소하지 않을 때, 학습률 조정
- ⓓ 학습률을 업데이트 하기 전에 몇 번의 에포크를 기다려야 하는지 결정 (여기서 self.patience = 5)

### 8.3.3 조기 종료를 이용한 성능 최적화

## 5단계 | 학습률 감소

```
torch.optim.lr_scheduler.ReduceLROnPlateau(self.optimizer, mode='min', patience=self.patience, factor=self.factor, min_lr=self.min_lr, verbose=True)
```

(a) (b) (c) (d) (e) (f) (g)

### ⑤ 학습률을 얼마나 감소시킬 지 지정

- 새로운 학습률 = 기존 학습률 \* factor
- ex. 현재 학습률 0.01이고, factor가 0.5이면 콜백 함수 실행 시, 다음 학습률은  $0.01 * 0.5$

### ⑥ 학습률의 하한선 지정

- 하한선 =  $\text{min\_lr} * \text{새로운 학습률}$
- ex. 학습률 0.1, factor 0.5, min\_lr 0.03 이라면, 하한선은  $0.03 * (0.1 * 0.5)$

### ⑦ 조기 종료의 시작과 끝 출력 여부 설정

### 8.3.3 조기 종료를 이용한 성능 최적화

## 6단계 | 조기 종료

```
class EarlyStopping():
    def __init__(self, patience=5, verbose=False, delta=0,
                 path='../chap08/data/checkpoint.pt'):
        self.patience = patience .....①
        self.verbose = verbose
        self.counter = 0
        self.best_score = None .....검증 데이터셋에 대한 오차 최적화 값(오차가 가장 낮은 값)
        self.early_stop = False .....조기 종료를 의미하며 초기값은 False로 설정
        self.val_loss_min = np.Inf .....np.Inf(infinity)는 넘파이에서 무한대를 표현
        self.delta = delta .....②
        self.path = path .....모델이 저장될 경로
```

### ① 오차 개선이 없는 에포크를 얼마나 기다릴 지 지정

- 해당 예제의 경우, patience = 5이므로  
개선 없는 에포크가 5번 지속될 경우 학습 종료

### ② 오차가 개선되고 있다고 판단하기 위한 최소 변화량

- 변화량이 delta 보다 적으면 개선이 없다고 판단

▶ keras의 callbacks 이용할 수 있음

(ModelCheckPoint, EarlyStopping 함수)

### 8.3.3 조기 종료를 이용한 성능 최적화

## 7단계 | 인수 값 지정

```
parser = argparse.ArgumentParser() ..... 인수 값을 받을 수 있는 인스턴스 생성  
parser.add_argument('--lr-scheduler', dest='lr_scheduler', action='store_true') ..... ①  
parser.add_argument('--early-stopping', dest='early_stopping', action='store_true') .....  
args = vars(parser.parse_args()) ..... ②
```

조기 종료에 대한 인수

① 원하는 인수 추가

② 입력 받은 인수 값 변수에 저장

```
parser.add_argument('--lr-scheduler', dest='lr_scheduler', action='store_true')
```

(a)

(b)

(c)

ⓐ 옵션 문자열의 이름, 명령 실행 시 사용 (ex. python main.py --lr-scheduler)

ⓑ 입력 값이 저장되는 변수 (lr-scheduler에 입력 값이 저장됨)

ⓒ action을 'store\_true'로 지정하면, 입력 값을 dest 파라미터에 의해 생성된 변수에 저장

### 8.3.3 조기 종료를 이용한 성능 최적화

## 8단계 | 사전 훈련된 모델의 파라미터 확인

\* ipywidgets 라이브러리 설치 필요

```
print(f"Computation device: {device}\n") ..... CPU를 사용하는지 GPU를 사용하는지 검사
model = models.resnet50(pretrained=True).to(device) ..... 사전 훈련된 ResNet50 사용
total_params = sum(p.numel() for p in model.parameters()) ..... 총 파라미터 수
print(f"{total_params:,} total parameters.")
total_trainable_params = sum(
    p.numel() for p in model.parameters() if p.requires_grad) ..... 학습 가능한 파라미터 수
print(f"{total_trainable_params:,} training parameters.")
```

\* 출력 결과

Computation device: cpu

25,557,032 total parameters.

25,557,032 training parameters.

CPU를 사용할 것이며,

전체 파라미터와 학습 가능한 파라미터의 수는 25,557,032 임을 의미

### 8.3.3 조기 종료를 이용한 성능 최적화

## 9단계 | 옵티마이저와 손실 함수 지정

```
lr = 0.001
epochs = 100
optimizer = optim.Adam(model.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()
```

## 10단계 | 오차, 정확도 및 모델의 이름에 대한 문자열 지정

```
loss_plot_name = 'loss' ..... 오차 출력에 대한 문자열
acc_plot_name = 'accuracy' ..... 정확도 출력에 대한 문자열
model_name = 'model' ..... 모델을 저장하기 위한 문자열

if args['lr_scheduler']:
    print('INFO: Initializing learning rate scheduler')
    lr_scheduler = LRScheduler(optimizer)

    loss_plot_name = 'lrs_loss' ..... 학습률 감소를 적용했을 때의 오차에 대한 문자열
    acc_plot_name = 'lrs_accuracy' ..... 학습률 감소를 적용했을 때의 정확도에 대한 문자열
    model_name = 'lrs_model' ..... 학습률 감소를 적용했을 때의 모델에 대한 문자열

if args['early_stopping']:
    print('INFO: Initializing early stopping')
    early_stopping = EarlyStopping()

    loss_plot_name = 'es_loss' ..... 조기 종료를 적용했을 때의 오차에 대한 문자열
    acc_plot_name = 'es_accuracy' ..... 조기 종료를 적용했을 때의 정확도에 대한 문자열
    model_name = 'es_model' ..... 조기 종료를 적용했을 때의 모델에 대한 문자열
```

- --lr-scheduler 또는 early-stopping 인수를 사용할 경우, (7단계)  
오차, 정확도 및 모델의 이름으로 사용할 문자열을 지정

### 8.3.3 조기 종료를 이용한 성능 최적화

## 11단계 | 모델 학습 함수 정의

```
def training(model, train_dataloader, train_dataset, optimizer, criterion):
    print('Training')
    model.train()
    train_running_loss = 0.0
    train_running_correct = 0
    counter = 0
    total = 0
    prog_bar = tqdm(enumerate(train_dataloader), total=int(len(train_dataset)/
        train_dataloader.batch_size)) ..... 훈련 진행 과정을 시각적으로 표현
    for i, data in prog_bar:
        counter += 1
        data, target = data[0].to(device), data[1].to(device)
        total += target.size(0)
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        train_running_loss += loss.item()
        _, preds = torch.max(outputs.data, 1)
        train_running_correct += (preds == target).sum().item()
        loss.backward()
        optimizer.step()

    train_loss = train_running_loss / counter
    train_accuracy = 100. * train_running_correct / total
    return train_loss, train_accuracy
```

## 12단계 | 모델 성능 검증 함수 정의

```
def validate(model, test_dataloader, val_dataset, criterion):
    print('Validating')
    model.eval()
    val_running_loss = 0.0
    val_running_correct = 0
    counter = 0
    total = 0
    prog_bar = tqdm(enumerate(test_dataloader), total=int(len(val_dataset)/
        test_dataloader.batch_size)) ..... 모델 검증 과정을 시각적으로 표현
    with torch.no_grad():
        for i, data in prog_bar:
            counter += 1
            data, target = data[0].to(device), data[1].to(device)
            total += target.size(0)
            outputs = model(data)
            loss = criterion(outputs, target)

            val_running_loss += loss.item()
            _, preds = torch.max(outputs.data, 1)
            val_running_correct += (preds == target).sum().item()

    val_loss = val_running_loss / counter
    val_accuracy = 100. * val_running_correct / total
    return val_loss, val_accuracy
```

### 8.3.3 조기 종료를 이용한 성능 최적화

## 13단계 | 모델 학습

```
train_loss, train_accuracy = [], [] ..... 훈련 데이터셋을 이용한 모델 학습 결과(오차, 정확도)를  
저장하기 위한 변수(리스트 형태를 갖습니다)  
  
val_loss, val_accuracy = [], [] ..... 검증 데이터셋을 이용한 모델 성능 결과(오차, 정확도)를  
저장하기 위한 변수(리스트 형태를 갖습니다)  
  
start = time.time()  
for epoch in range(epochs):  
    print(f"Epoch {epoch+1} of {epochs}")  
    train_epoch_loss, train_epoch_accuracy = training(  
        model, train_dataloader, train_dataset, optimizer, criterion  
    )  
    val_epoch_loss, val_epoch_accuracy = validate(  
        model, val_dataloader, val_dataset, criterion  
    )  
    train_loss.append(train_epoch_loss)  
    train_accuracy.append(train_epoch_accuracy)  
    val_loss.append(val_epoch_loss)  
    val_accuracy.append(val_epoch_accuracy)  
    if args['lr_scheduler']: ..... 인수 값이 lr_scheduler이면 다음을 실행  
        lr_scheduler(val_epoch_loss)  
    if args['early_stopping']: ..... 인수 값이 early_stopping이면 다음을 실행  
        early_stopping(val_epoch_loss, model)  
        if early_stopping.early_stop:  
            break  
    print(f"Train Loss: {train_epoch_loss:.4f}, Train Acc: {train_epoch_accuracy:.2f}")  
    print(f"Val Loss: {val_epoch_loss:.4f}, Val Acc: {val_epoch_accuracy:.2f}")  
end = time.time()  
print(f"Training time: {(end-start)/60:.3f} minutes")
```

## 14단계 | 모델 학습 결과 출력

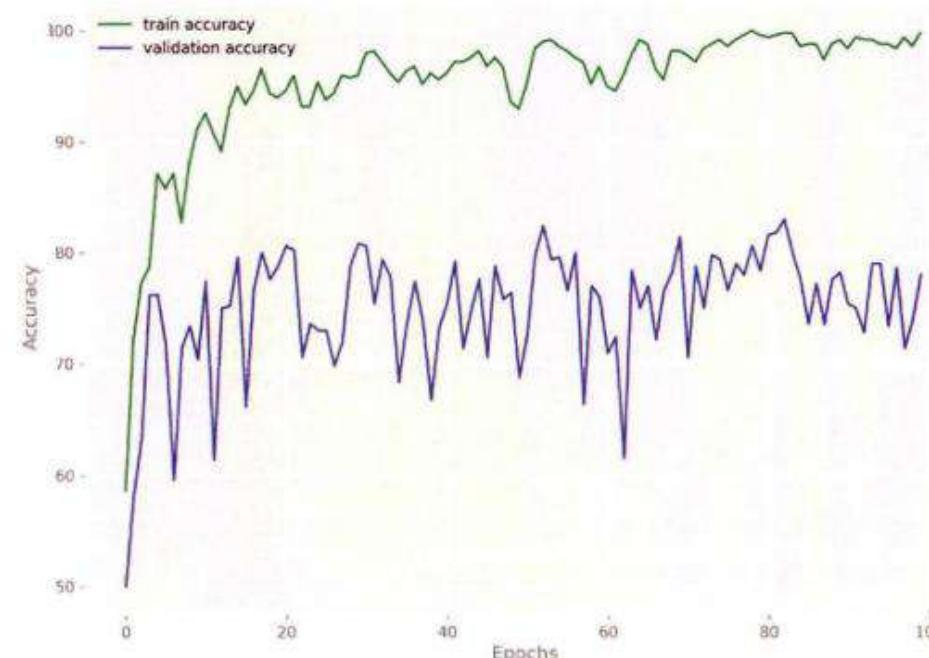
```
print('Saving loss and accuracy plots...')  
plt.figure(figsize=(10, 7))  
plt.plot(train_accuracy, color='green', label='train accuracy') ..... 훈련 데이터셋에 대한 정확도를  
                                                                그래프로 출력  
plt.plot(val_accuracy, color='blue', label='validation accuracy') ..... 검증 데이터셋에 대한 정확도를 그래프로 출력  
plt.xlabel('Epochs')  
plt.ylabel('Accuracy')  
plt.legend()  
plt.savefig(f'../chap08/img/{acc_plot_name}.png')  
plt.show()  
  
plt.figure(figsize=(10, 7))  
plt.plot(train_loss, color='orange', label='train loss') ..... 훈련 데이터셋에 대한 오차를 그래프로 출력  
plt.plot(val_loss, color='red', label='validation loss') ..... 검증 데이터셋에 대한 오차를  
                                                                그래프로 출력  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
plt.savefig(f'../chap08/img/{loss_plot_name}.png')  
plt.show()  
  
print('Saving model...')  
torch.save(model.state_dict(), f'../chap08/img/{model_name}.pth') ..... 모델을 저장  
print('TRAINING COMPLETE')
```

\* 어떤 인수도 사용되지 않는 모델

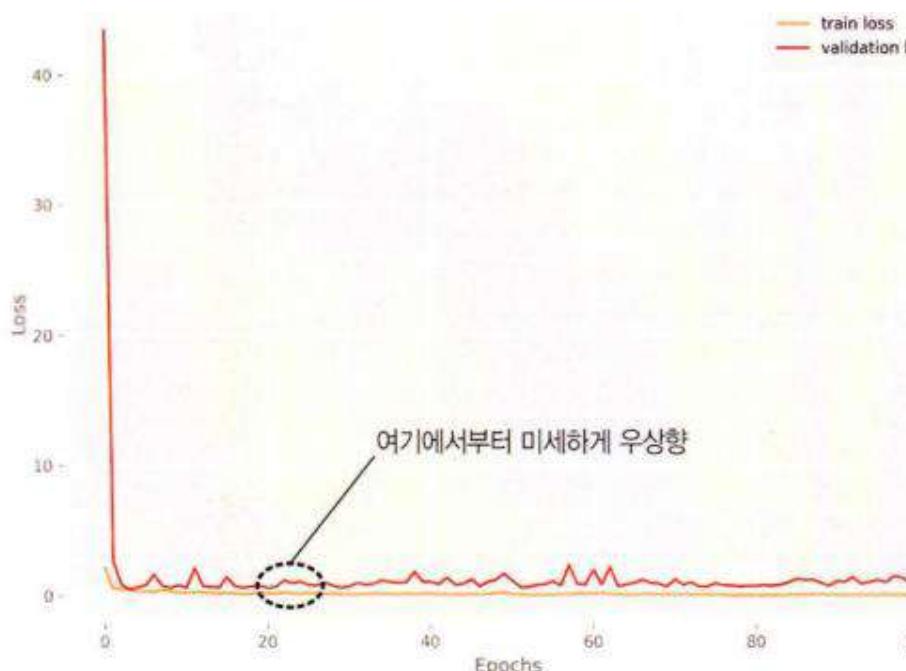
### 8.3.3 조기 종료를 이용한 성능 최적화

## 15단계 | 모델 정확도 및 오차(1)

\* 어떤 인수도 적용되지 않은 경우



정확도



오차

- 정확도
  - 위 아래로 변동이 많음
  - 일부 에포크 사이의 기복 심함
- 오차
  - 에포크 10 이후부터 오차 우상향

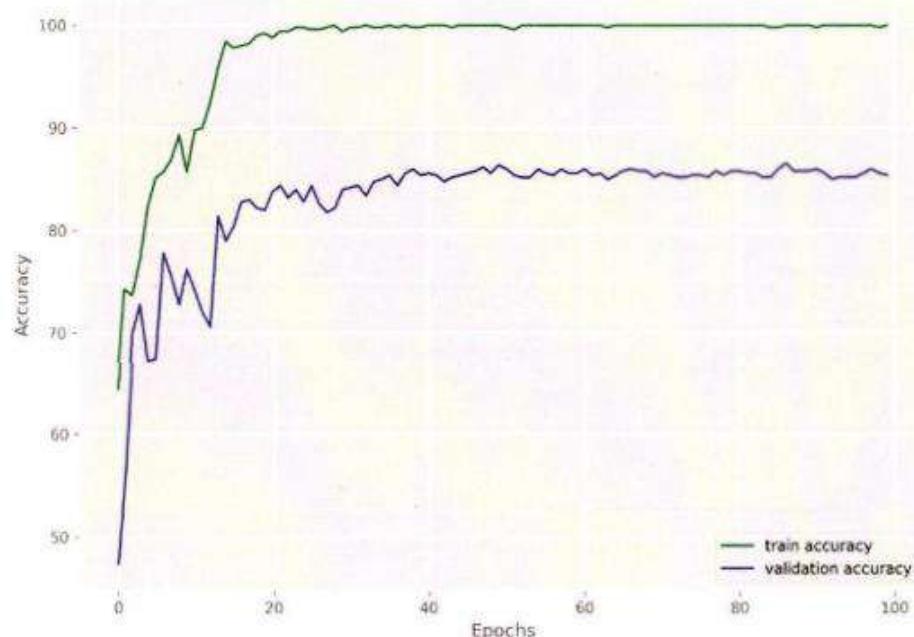
=> 모델의 과적합

따라서 계속 모델 학습하려면 학습률 줄여야 함

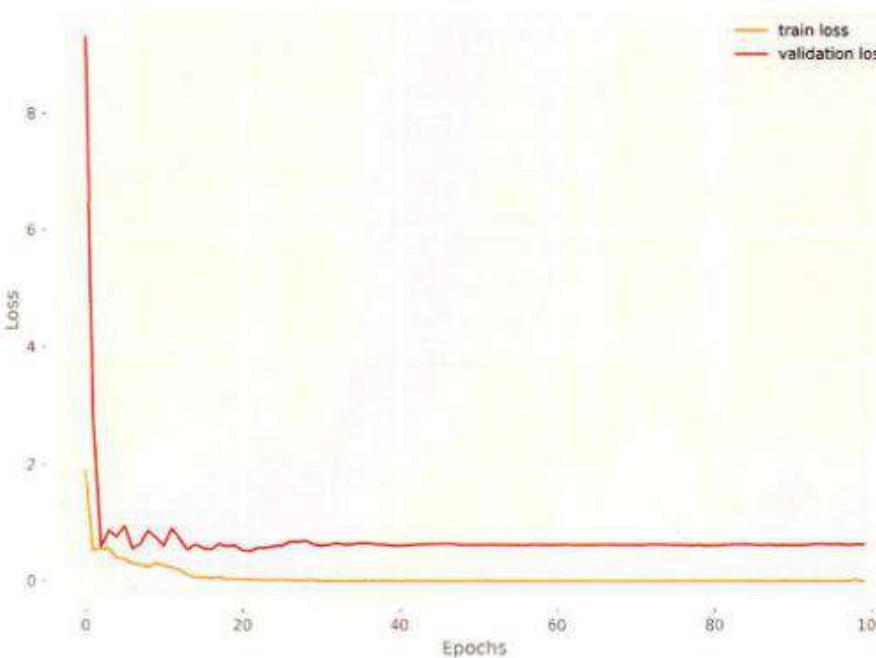
### 8.3.3 조기 종료를 이용한 성능 최적화

## 15단계 | 모델 정확도 및 오차(2)

\* 학습률 감소에 대한 인수 적용한 경우



정확도



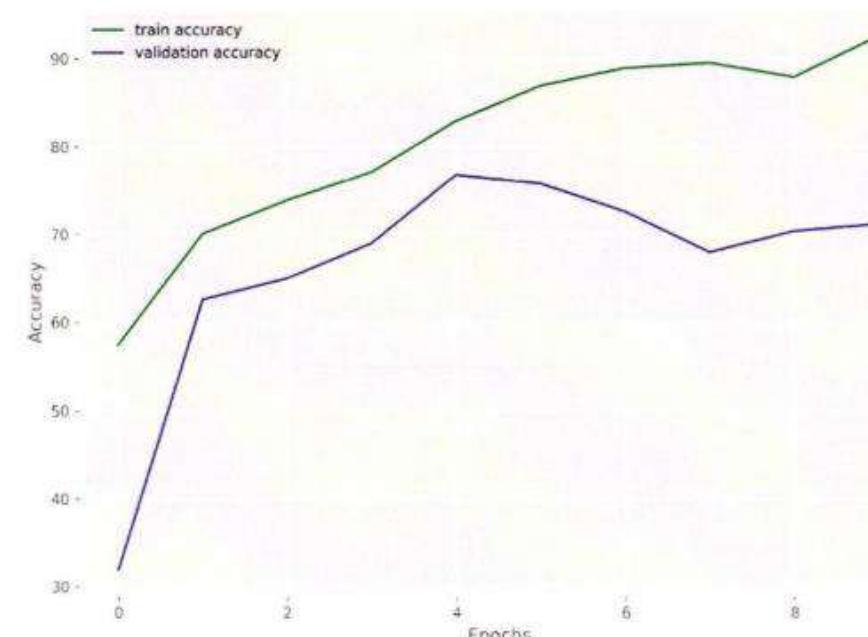
오차

- 정확도
  - 완만한 곡선 형태
  - 훈련이 종료된 시점의 정확도 높음
- 오차
  - 그래프의 우상향 현상 없어짐
  - 학습률 스케줄러가 성능 향상에 어느 정도 기여했음을 의미

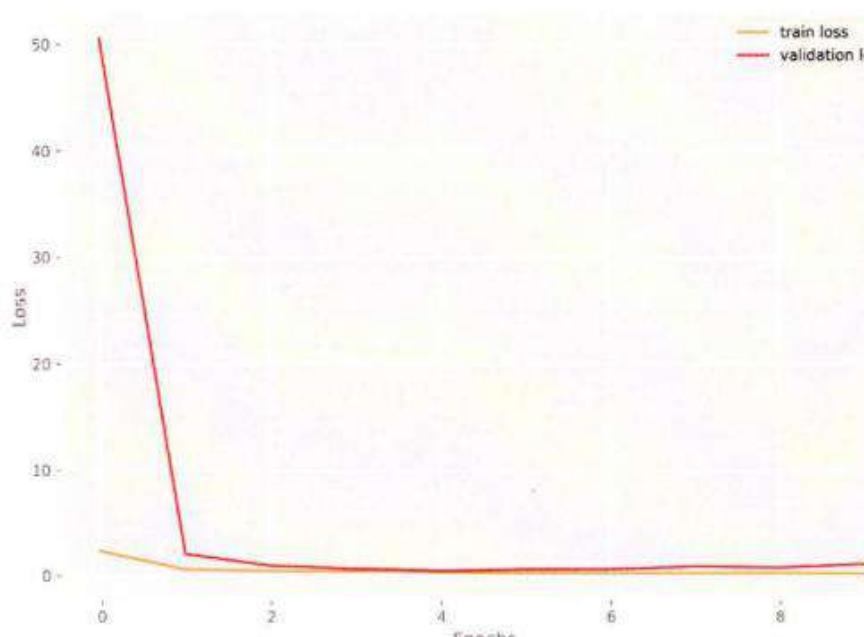
### 8.3.3 조기 종료를 이용한 성능 최적화

## 15단계 | 모델 정확도 및 오차(3)

\* 조기 종료 인수 적용한 경우



정확도



오차

- 정확도 : 불안정한 결과 출력
- 오차 : 많이 낮아짐
- 조기 종료가 항상 성능에 좋은 영향을 미치는 것은 아님
- 조기 종료의 경우에는 성능 향상보다는 자원(CPU/메모리)의 효율화
- 그래프가 의미하는 내용을 잘 이해하고 적절한 성능 향상 방안을 적용하는 것이 중요

다섯 번째 에포크부터 조기 종료가 수행되고 있기 때문에 실제로 학습은 네 번째 에포크까지만 수행

오늘 발제 끝!