



12주차 발표

김예진 박보영 오수진

목차

#01 성능 최적화

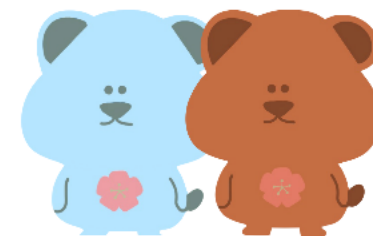
#02 배치 정규화

#03 드롭 아웃

#04 조기 종료



01. 성능 최적화



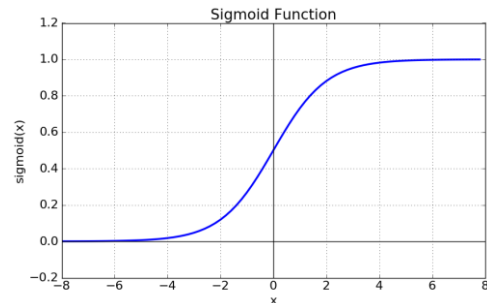
1. 성능 최적화

1. 데이터를 사용한 성능 최적화

- 최대한 많은 데이터 수집
- 데이터 생성(**5장 – RandomResizedCrop, RandomHorizontalFlip)
- 데이터 범위(scale) 조정

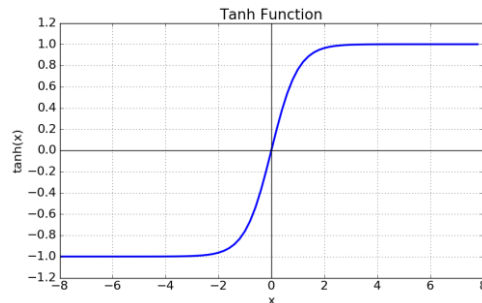
(특성별로 데이터의 스케일이 다르다면, 머신러닝이 잘 동작하지 않을 수 있음 → 모든 특성의 범위(또는 분포)를 같게 만들어줘야 함)

- 정규화(Normalization)
- 표준화(Standardization)
- 규제화(Regularization)
- 활성화 함수(시그모이드, 하이퍼볼릭 탄젠트)



시그모이드 함수

데이터셋의 범위를 0~1의 값을 갖도록 조정



하이퍼볼릭 탄젠트

데이터셋의 범위를 -1~1 값을 갖도록 조정

2. 알고리즘을 이용한 성능 최적화: 유사한 용도의 알고리즘을 선택하여 모델 훈련 시켜보고 최적의 성능 보이는 알고리즘 선택

1. 성능 최적화

3. 알고리즘 튜닝을 이용한 성능 최적화

진단	<ul style="list-style-type: none">모델 평가 결과를 바탕으로 성능 향상이 멈춘 원인 분석
	<ul style="list-style-type: none">과적합(훈련 성능이 검증보다 눈에 띄게 좋은 경우) → 규제화
	<ul style="list-style-type: none">과소적합(훈련과 검증 성능이 모두 좋지 않은 경우) → 네트워크 구조 변경, 에포크 수 조절(훈련 늘리기)
	<ul style="list-style-type: none">훈련 성능이 검증을 넘어서는 변곡점이 있는 경우 → 조기종료 고려
가중치	<ul style="list-style-type: none">가중치에 대한 초기값은 작은 난수 사용오토인코더를 이용하여 사전 훈련 후 지도학습 진행
학습률	<ul style="list-style-type: none">초기에 매우 크거나 작은 임의의 난수를 선택하여 학습 결과를 보면서 변경네트워크 계층 ↑ → 학습률 ↑, 네트워크 계층 ↓ → 학습률 ↓
활성화 함수	<ul style="list-style-type: none">활성화 함수 변경할 때 손실 함수도 함께 변경하는 경우 많음ex) 활성화 함수: 시그모이드, 하이퍼볼릭 탄젠트 → 출력층: 소프트 맥스, 시그모이드 함수 주로 선택
배치와 에포크	<ul style="list-style-type: none">일반적으로 큰 에포크와 작은 배치를 사용하는 것이 최근 딥러닝 트렌드 but 다양한 테스트 진행하는 것 좋음
옵티마이저 및 손실함수	<ul style="list-style-type: none">확률적 경사 하강법 일반적으로 많이 사용아담(Adam), 알엠에스프롭(RMSProp) 등도 좋은 성능 보임
네트워크 구성	<ul style="list-style-type: none">네트워크 구성 변경해가면서 성능 테스트(네트워크 넓게/깊게/결합 등등 시도)

1. 성능 최적화

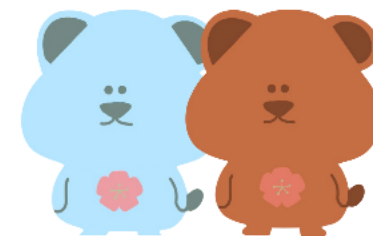
4. **양상블을 이용한 성능 최적화**: 모델을 두 개 이상 섞어서 사용

5. **하드웨어를 이용한 성능 최적화**

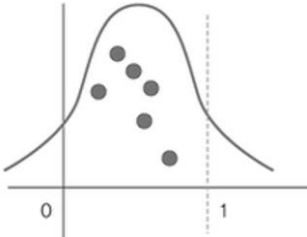
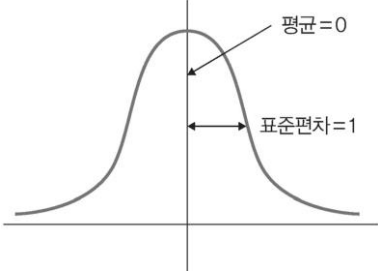
- 기존 CPU가 아닌 GPU 이용
- CPU와 GPU는 개발된 목적 다름 → 내부 구조 차이

CPU	GPU
직렬 처리 (명령어가 입력되는 순서대로 데이터 처리)	병렬 처리 (여러 명령 동시에 처리)
파이썬이나 매트랩(MATLAB)처럼 행렬 연산을 많이 사용하는 재귀 연산이 대표적인 '직렬' 연산 수행	역전파처럼 복잡한 미적분은 병렬 연산을 해야 속도가 빨라짐
	병렬 처리는 복잡한 연산이 수반되는 딥러닝에서 속도와 성능을 높여주는 주요 원인이 될 수 있음
	GPU용 파이토치 설치하려면 CUDA와 cuDNN 설치해야 함

02. 배치 정규화



1. 배치 정규화와 유사한 의미로 사용되는 용어

정규화(normalization)	표준화(Standardization)	규제화(regularization)
<ul style="list-style-type: none">데이터 범위를 사용자가 원하는 범위로 제한 (주로 [0,1]로 스케일링)각 특성 범위를 조정한다는 의미로 특성 스케일링이라고도 불리고, 스케일 조정을 위해 MinMaxScaler() 기법 사용	<ul style="list-style-type: none">기존 데이터를 평균은 0, 표준편차는 1인 형태의 데이터로 만드는 방법평균을 기준으로 얼마나 떨어져 있는지 살펴볼 때 사용보통 데이터 분포가 가우시안 분포를 따를 때 유용	<ul style="list-style-type: none">모델 복잡도를 줄이기 위한 제약을 두는 방법 (데이터가 네트워크에 들어가기 전에 필터를 적용한 것)드롭아웃, 조기종료
$\frac{x - x_{\min}}{x_{\max} - x_{\min}}$ 	$\frac{x - m}{\sigma}$ 	

2. 배치 정규화(Batch Normalization)

배치 정규화(Batch Normalization)

- 학습 과정에서 각 배치 단위 별로 데이터가 다양한 분포를 가지더라도, 각 미니배치의 평균과 분산을 이용해 정규화하는 것
- 분산된 분포를 정규분포로 만들기 위해 표준화와 유사한 방식을 미니 배치에 적용하여 평균은 0, 표준편차는 1로 유지하도록 함
- 데이터셋 분포가 일정해지기 때문에 속도 향상됨
- 평균과 분산을 조정하는 과정이 별도의 과정으로 떼어진 것이 아니라, 신경망 안에 포함되어 있음
- 기울기 소멸이나 기울기 폭발 문제를 해결하기 위한 방법
 - 기울기 소멸: 오차 정보를 역전파시키는 과정에서 기울기가 급격히 0에 가까워져 학습이 되지 않는 현상
 - 기울기 폭발: 학습 과정에서 기울기가 급격히 커지는 현상
 - 원인: 내부 공변량 변화

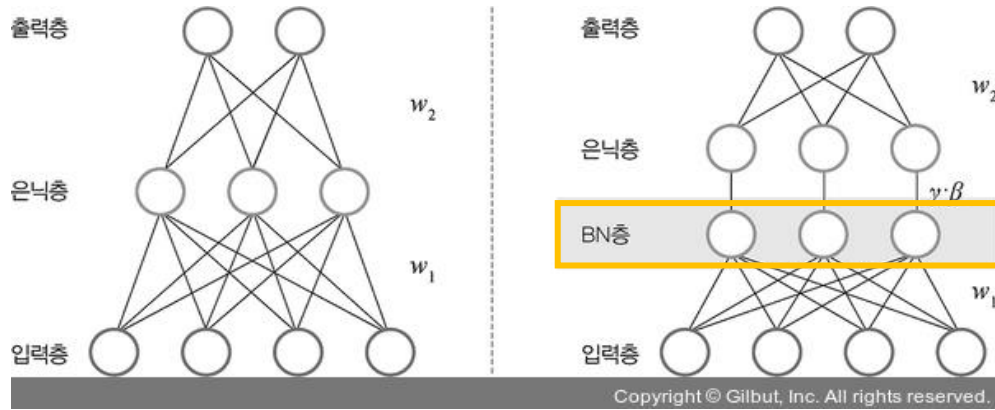
내부 공변량 변화(Internal Covariance Shift): 학습 과정에서 계층 별로 입력 데이터의 분포가 달라지는 현상

- Gradient 업데이트 할 때, 한 번에 모든 학습 데이터를 넣어서 구하는 것이 아닌, 데이터를 batch 단위로 나눠서 학습하는 방법을 사용하는 것이 일반적
- 그러나 Batch 단위로 학습하게 되면 내부 공변량 변화 문제가 발생함
(각 layer의 Input feature가 조금씩 변해서, Hidden layer에서 Input feature의 변동량이 누적되게 되면 각 layer에서는 입력되는 값이 전혀 다른 유형의 데이터라고 받아들일 수 있기 때문)
- 즉, batch 단위 간의 데이터가 상이한 문제가 발생함
 - 손실 함수로 Relu 사용, 초깃값 튜닝, 학습률 조정
 - 간접적인 방법보다는 학습하는 과정 자체를 전체적으로 안정화하여 학습 속도를 가속시킬 수 있는 근본적인 방법인 배치 정규화를 사용하는 것이 좋음

2. 배치 정규화(Batch Normalization)

배치 정규화(Batch Normalization)

- 평균과 분산을 조정하는 과정이 별도의 과정으로 떼어진 것이 아니라, 신경망 안에 포함되어 있음



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

---- ① 미니 배치 평균 구함

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

---- ② 미니 배치의 분산과 표준편차 구함

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

---- ③ 정규화 수행

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

---- ④ 스케일 조정

배치 정규화(Batch Normalization) 단점

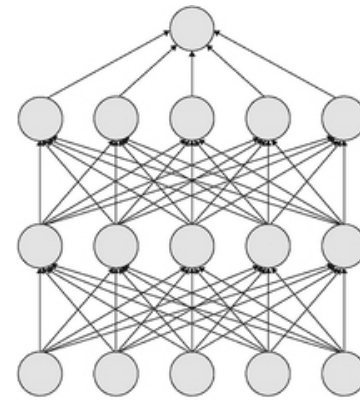
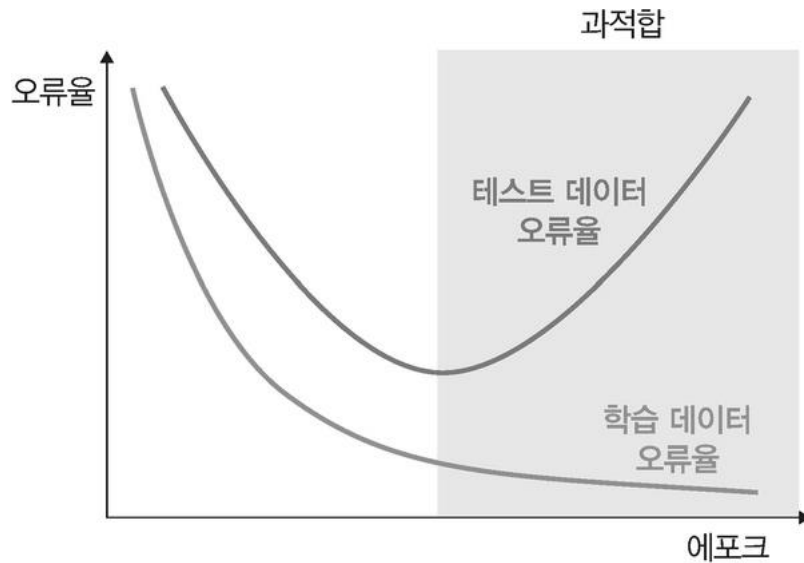
- 배치 크기가 작을 때는 정규화 값이 기존 값과 다른 방향으로 훈련될 수 있음.
(예를 들어 분산이 0이면 정규화 자체가 안 되는 경우가 생길 수 있음)
- RNN은 네트워크 계층별로 미니 정규화를 적용해야 하기 때문에 모델이 더 복잡해지면서 비효율적일 수 있음

3. 드롭아웃을 이용한 성능 최적화

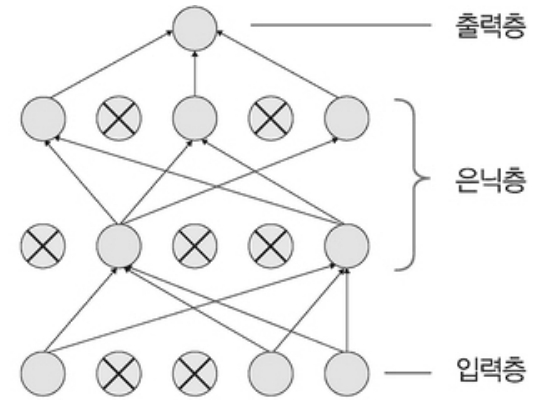


0.1. 드롭아웃을 이용한 성능 최적화

드롭아웃(dropout) 훈련 시 일부 뉴런만 사용하고, 나머지 뉴런에 해당하는 가중치는 업데이트하지 않는 방법이다. 꺼진 뉴런에 해당하는 신호가 없으므로 **과적합을 방지**할 수 있다.



일반적인 신경망



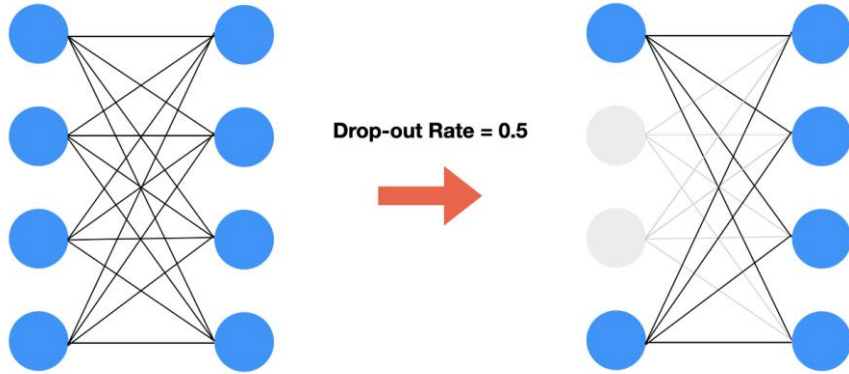
드롭아웃이 적용된 신경망

단점: 훈련시간 길어짐

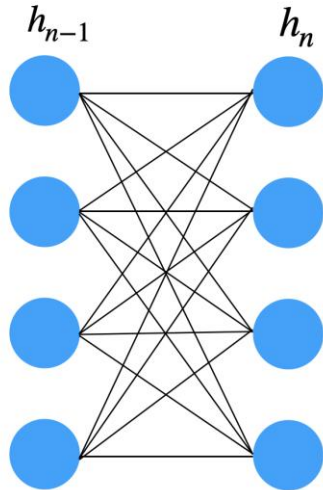
장점: 과적합 방지, 모델 성능 향상

0.1. 드롭아웃을 이용한 성능 최적화

EX



4개의 뉴런 각각은 0.5의 확률로 제거될지 말지 랜덤하게 결정된다.



$$h_n = a(\alpha W_n h_{n-1} + b_n)$$

테스트 데이터로 평가할 때는 **노드들을 모두 사용하여** 출력하되,
드롭아웃 비율을 곱해서 성능을 평가한다.

드롭아웃 비율을 활용해 scaling 하는 이유는 기존에 모델 학습 시,
드롭아웃의 확률로 각 뉴런이 꺼져 있었다는 점을 고려하기 위해서이다.

a 는 activation function, α 는 drop-out rate

0.2. 성능 최적화 코드 실습

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

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

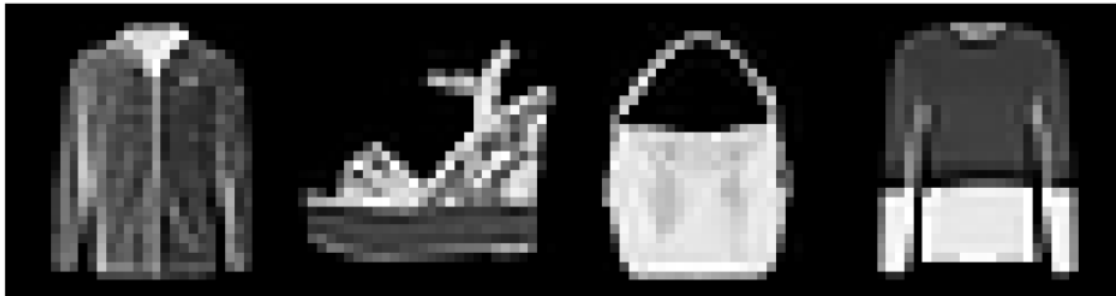
```
print(images.shape)
print(images[0].shape)
print(labels[0].item())
```

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

Batch_size=4, 한번에 네 개씩 데이터 가져옴
Torch.size([batch_size, 채널, 너비, 높이])

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

['4', '5', '8', '2']



이미지 출력해봄

0.2.1. 배치 정규화를 이용한 성능 최적화

배치 정규화

#배치 정규화 x

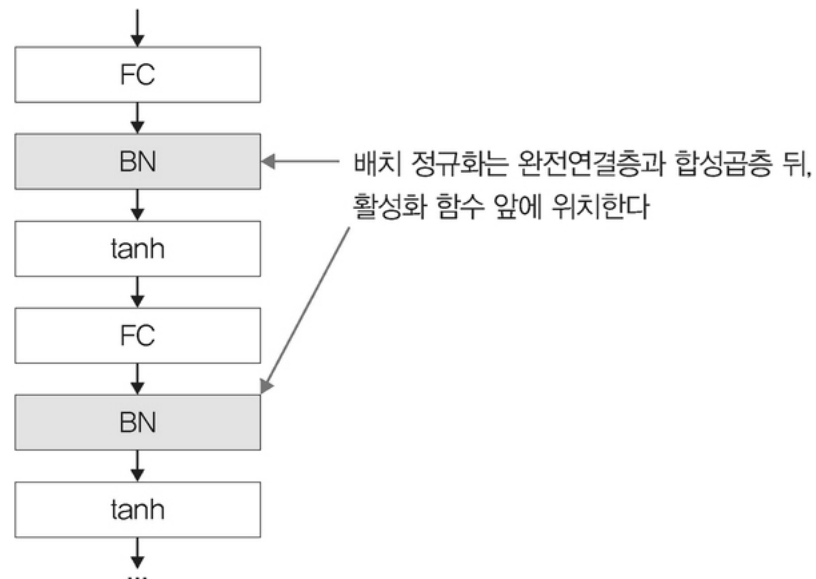
```
class NormalNet(nn.Module):
    def __init__(self):
        super(NormalNet, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(784, 48), # 28 x 28 = 784
            nn.ReLU(),
            nn.Linear(48, 24),
            nn.ReLU(),
            nn.Linear(24, 10) #클래스 10개
        )

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

배치 정규화 0

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



Linear -> BatchNorm1d -> ReLu -> Linear -> BatchNorm1d -> ReLu

0.2.1. 배치 정규화를 이용한 성능 최적화

```
model = NormalNet().to(device)
print(model)
```

```
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)
  )
)
```

```
model_bn = BNNet().to(device)
print(model_bn)
```

```
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)
  )
)
```

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

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

모델 선언

데이터셋 불러오기

옵티마이저, 손실함수 설정

0.2.1. 배치 정규화를 이용한 성능 최적화

모델 학습

```
loss_arr = []
loss_bn_arr = []
max_epochs = 20

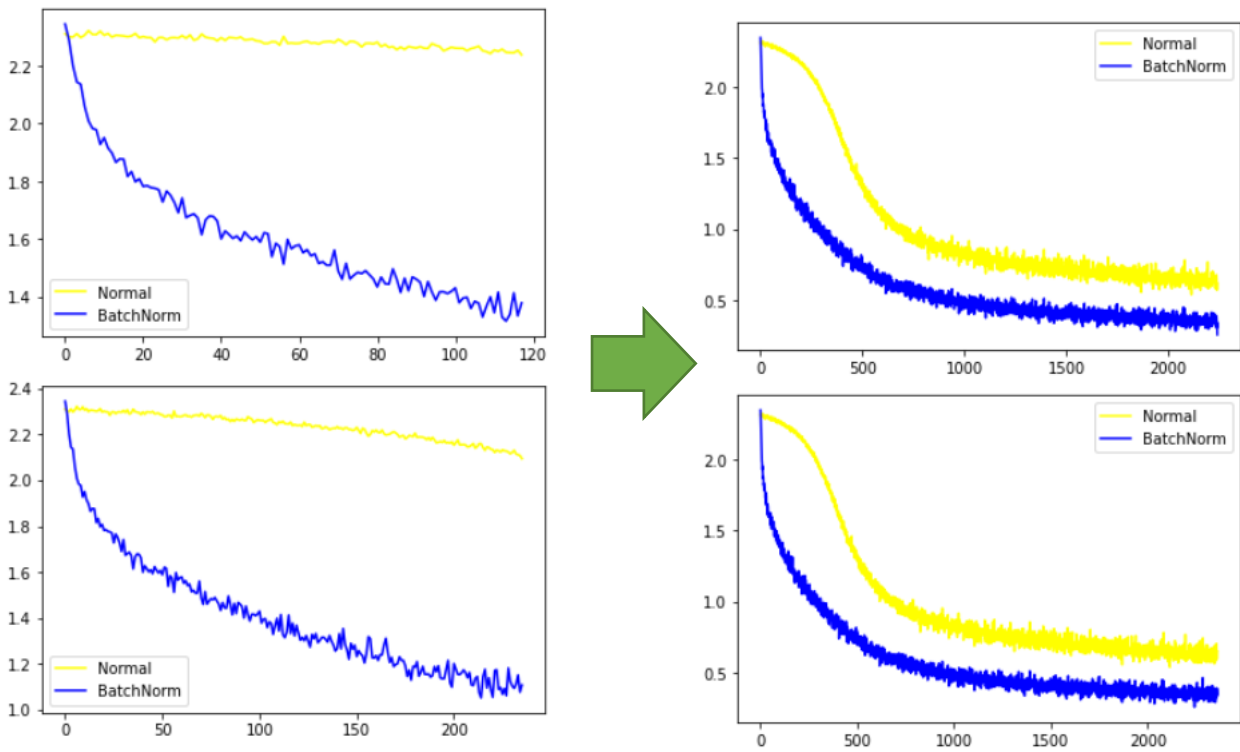
for epoch in range(max_epochs):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        opt.zero_grad()
        outputs = model(inputs).to(device)
        loss = loss_fn(outputs, labels)
        loss.backward()
        opt.step()

        opt_bn.zero_grad()
        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()
```

배치정규화 O (파랑) / 배치정규화 X (노랑) Loss FNC



시간이 흐를 수록, 파랑노랑 모두 오차가 줄어들지만, 그 줄어드는 범위와 값의 차이를 봤을 때, 배치정규화를 적용한 모델이 에포크가 진행될 수록 오차도 줄어들면서 안정적인 학습을 하고 있다

0.2.2. 드롭아웃을 이용한 성능 최적화

드롭아웃

```
N = 50
noise = 0.3

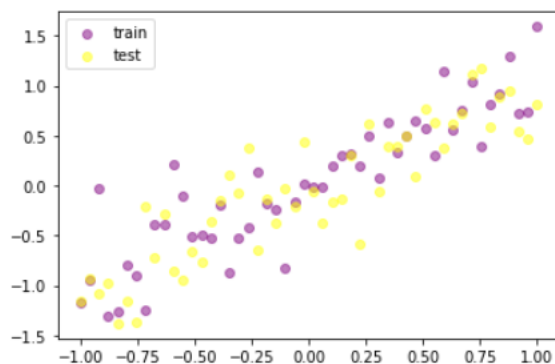
x_train = torch.unsqueeze(torch.linspace(-1, 1, N), 1) # 차원 늘리기
y_train = x_train + noise * torch.normal(torch.zeros(N, 1), torch.ones(N, 1)) #정규화

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))
```

데이터셋 분포를 위한 전처리

```
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.legend()
plt.show()
```

데이터셋 분포



0.2.2. 드롭아웃을 이용한 성능 최적화

#드롭아웃 0

```
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),
)
```

#드롭아웃 X

```
model_dropout = torch.nn.Sequential(
    torch.nn.Linear(1, N_h),
    torch.nn.Dropout(0.2),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, N_h),
    torch.nn.Dropout(0.2),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, 1),
)
```

```
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().to(device)
```

드롭아웃 모델 생성

Linear -> Dropout -> ReLu -> Linear-> Dropout -> ReLu

옵티마이저, 손실함수 설정

0.2.2. 드롭아웃을 이용한 성능 최적화

#모델학습

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

    pred_dropout = model_dropout(x_train)
    loss_dropout = loss_fn(pred_dropout, y_train)
    opt_dropout.zero_grad()
    loss_dropout.backward()
    opt_dropout.step()

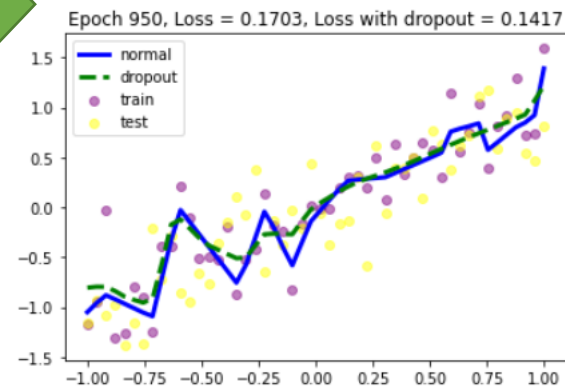
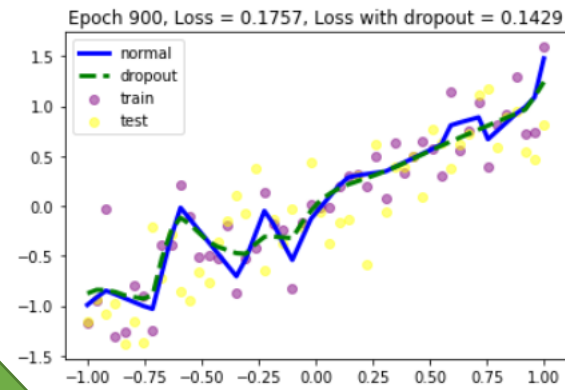
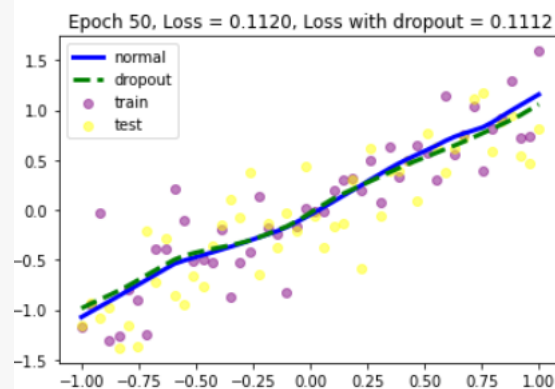
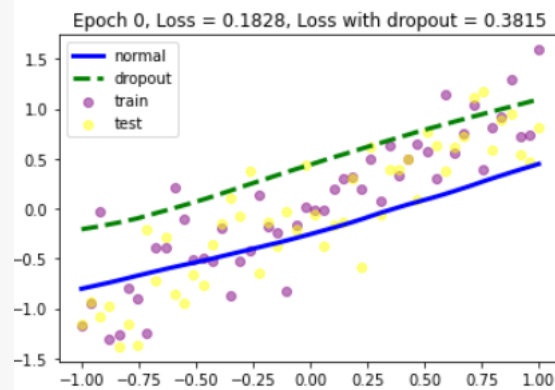
    if epoch % 50 == 0:
        model.eval()
        model_dropout.eval()

        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)

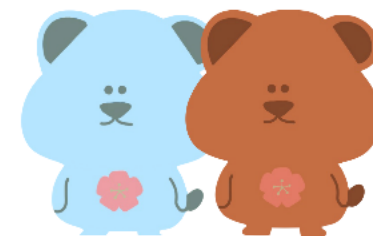
        plt.scatter(x_train.data.numpy(), y_train.data.numpy(), c='purple', alpha=0.5, label='train')
        plt.scatter(x_test.data.numpy(), y_test.data.numpy(), c='yellow', alpha=0.5, label='test')
        plt.plot(x_test.data.numpy(), test_pred.data.numpy(), 'b-', lw=3, label='normal')
        plt.plot(x_test.data.numpy(), test_pred_dropout.data.numpy(), 'g--', lw=3, label='dropout')

        plt.title('Epoch %d, Loss = %0.4f, Loss with dropout = %0.4f' % (epoch, test_loss, test_loss_dropout))
        plt.legend()
        model.train()
        model_dropout.train()
        plt.pause(0.05)
```



시간이 흐를 수록, 일반모델(보라색선)이 훈련데이터셋을 찾아가는, 과적합 문제가 발생한다.
드롭아웃을 적용한 모델(초록점선)은 과적합 현상이 발생하지 않는다

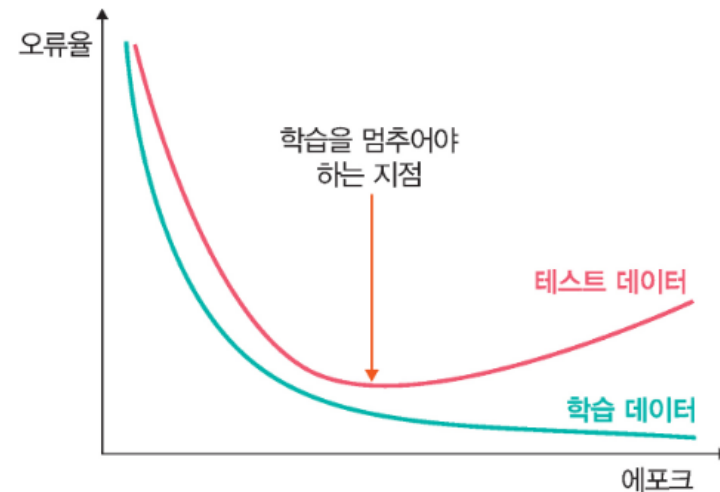
04. 조기 종료



#01 조기 종료와 학습률 감소

📌 조기 종료 (early stopping)

- 과적합을 회피하는 규제 기법
- 매 에포크마다 검증 데이터에 대한 오차 측정 → 모델의 종료 시점 제어
- 검증 데이터셋에 대한 오차가 증가하는 시점에서 학습 종료
- 최고의 성능을 보장하지는 X



📌 학습률 감소

- 학습률을 조금씩 낮추어 주는 성능 튜닝 기법
- 주어진 횟수만큼 검증 데이터셋에 대한 오차 감소가 없으면 학습률 감소

#02 코드 – 콜백 함수

📌 콜백 함수 (callback)

- 개발자는 함수 등록만 하고 **특정 이벤트 발생에 의해 함수를 호출하고 처리**하도록 하는 함수
(개발자가 명시적으로 함수 호출 X)
- 동기적 함수 : 코드가 위에서 아래로, 왼쪽에서 오른쪽으로 순차적으로 실행
- 비동기 함수 : 병렬 처리 -> 코드 실행 시 완료까지 오래 걸릴 경우 기다리지 않고 다른 코드 먼저 처리

#02 코드 – 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
        )
    def __call__(self, val_loss):
        self.lr_scheduler.step(val_loss)
```

학습률 스케줄러 :

patience 동안 검증 데이터셋에 대한 오차 감소가 없으면
학습률을 factor배로 감소

mode : 언제 학습률을 조정할지에 대한 기준

- 오차 → min
- 정확도 → max

patience : 학습률 업데이트 하기 전 몇 번의 에포크를
기다릴지

factor : 학습률을 얼마나 감소시킬지

min_lr : 학습률의 하한선

#02 코드 – 조기 종료

```
class EarlyStopping():
    def __init__(self, patience=5, verbose=False, delta=0, path='../chap08/data/checkpoint.pt'):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.path = path

    def __call__(self, val_loss, model):
        score = -val_loss
```

patience : 오차 개선이 없는 에포크를 얼마나 기다려줄지

delta : 오차가 개선되고 있다고 판단하기 위한 최소 변화량

①

```
if self.best_score is None:
    self.best_score = score
    self.save_checkpoint(val_loss, model)
```

②

```
elif score < self.best_score + self.delta:
    self.counter += 1
    print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
    if self.counter >= self.patience:
        self.early_stop = True
```

③

```
else:
    self.best_score = score
    self.save_checkpoint(val_loss, model)
    self.counter = 0
```

```
def save_checkpoint(self, val_loss, model):
    if self.verbose:
        print(f'Validation loss decreased ({self.val_loss_min:.6f} --> {val_loss:.6f}). Saving model ...')
    torch.save(model.state_dict(), self.path)
    self.val_loss_min = val_loss
```

①

best_score에 값이 존재하지 않으면 실행 :
best_score에 score 저장, save_checkpoint

②

best_score+delta가 score보다 크면 실행
즉, 오차가 delta만큼 개선되지 않았을 때 :
counter+1, counter가 patience만큼 채워지면 조기 종료

③

그 외 모든 경우 실행 :
best_score에 score 저장, save_checkpoint

#02 코드 – 조기 종료

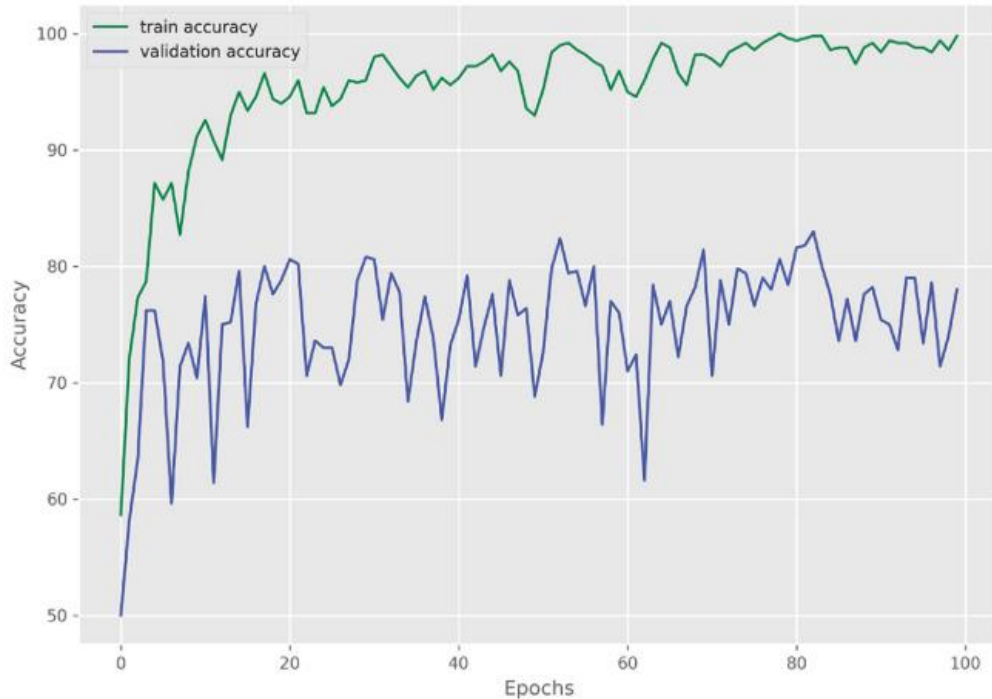
케라스의 콜백 이용하는 방법

monitor : Val_loss 값이 개선되었을 때 호출
save_best_only : 가장 최적의 값만 저장
auto : 시스템이 알아서 best 값을 찾으라는 것

```
from keras.callbacks import ModelCheckpoint, EarlyStopping
checkpoint = ModelCheckpoint('checkpoint-epoch.h5'.format(EPOCH, BATCH_SIZE),
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True,
                             mode='auto'
                             )
earlystopping = EarlyStopping(monitor='val_loss',
                              patience=10
                              )
```

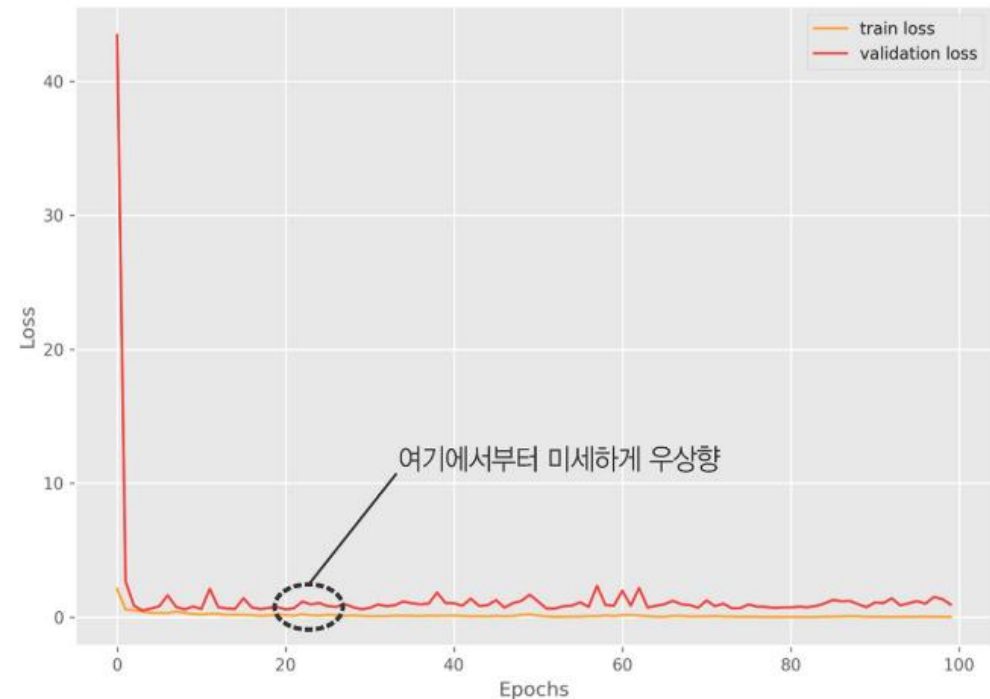
monitor : 학습률 업데이트 기준 = val_loss
patience : 개선이 없는 에포크를 얼마나 기다려줄지

#03 결과 그래프 비교 - 기본



정확도

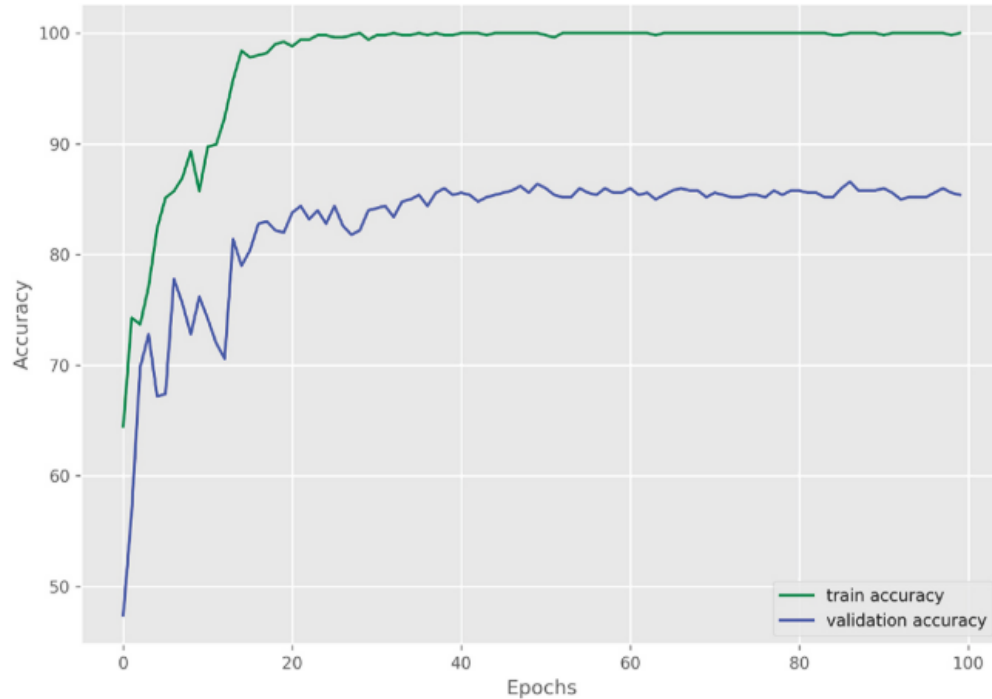
- 위아래로 많은 변동
- 10% 이상 차이 나는 일부 에포크 사이 기복 심함



오차

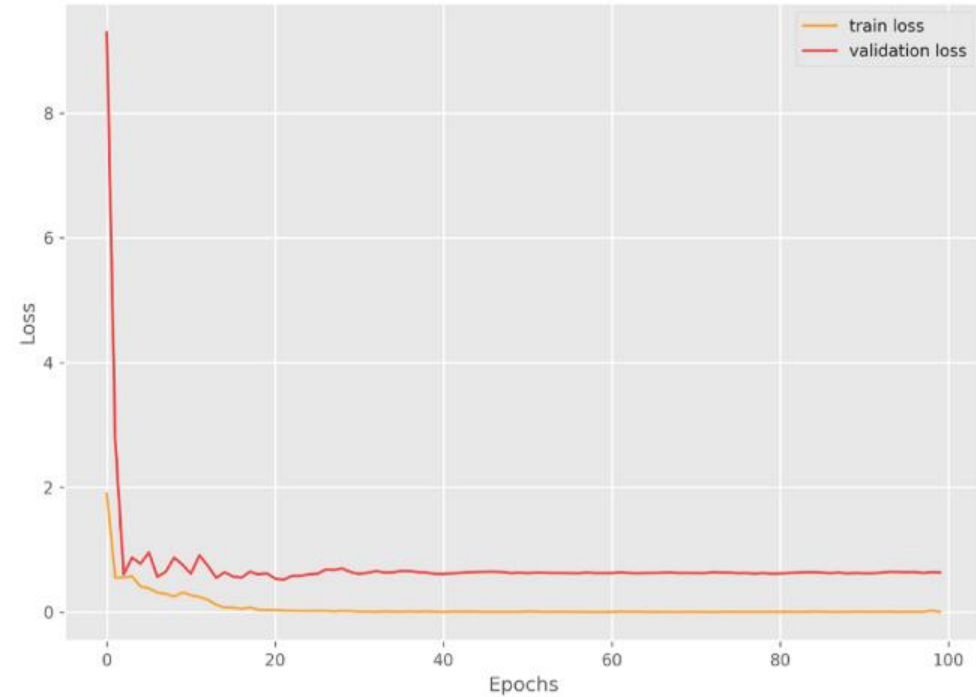
- 에포크 10 이후 검증 데이터셋에 대한 오차가 미세한 차이로 이상향 → 과적합 시작

#03 결과 그래프 비교 - 학습률 감소



정확도

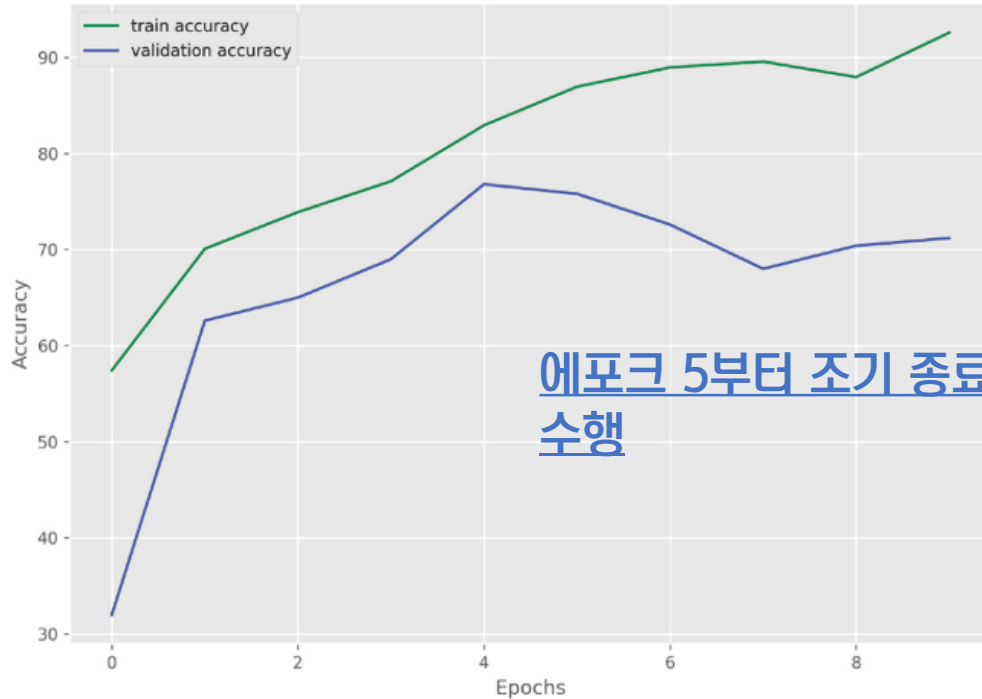
- 완만한 곡선 형태
- 훈련 종료 시점의 검증 데이터셋 정확도 높음



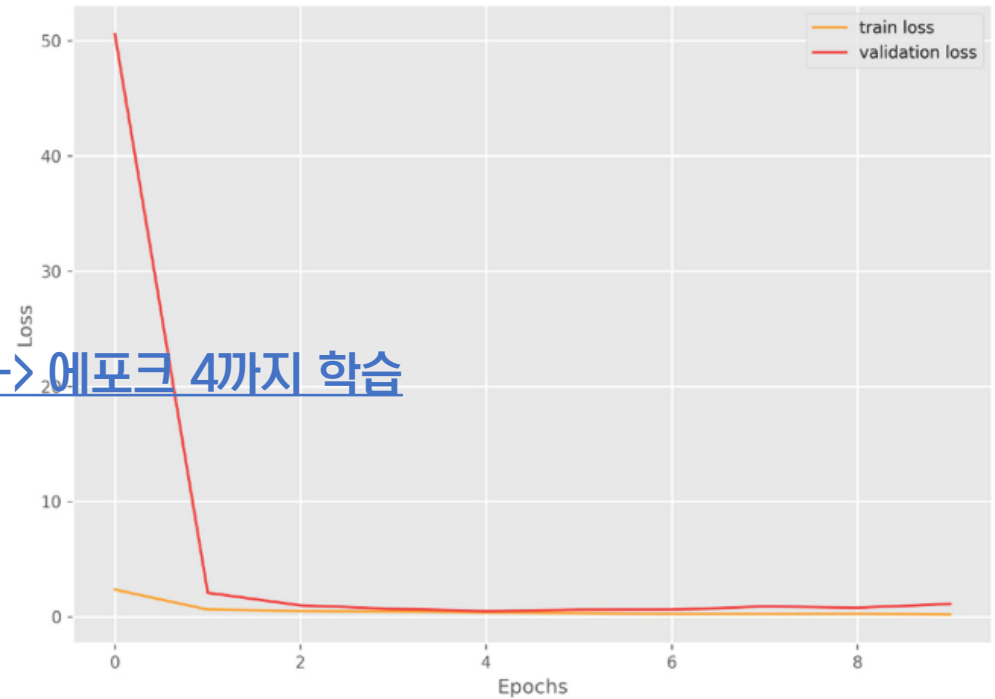
오차

- 이상향 현상 없어짐
- 에포크 20 정도에서 오차 정체 시작 → 에포크 30만 수행해도 결과 좋을 것

#03 결과 그래프 비교 – 조기 종료



에포크 5부터 조기 종료 수행 -> 에포크 4까지 학습 수행



정확도

- 검증 데이터셋에 대한 정확도 불안정
- But, 학습을 이어 간다고 해서 더 좋은 결과를 얻을 수 있다는 보장도 X

오차

- 정확도는 좋지 않지만 오차는 많이 낮아짐 -> 정확도만 보고 조기 종료 효과가 없다고 할 수 X

#03 결과 그래프 비교

학습률 조정 → 모델 성능 향상

조기 종료 → 자원(CPU/메모리) 효율화

도움이 되는 것은 분명함..

조기 종료가 항상 성능에 좋은 영향을 미치는 것은 X

조기 종료로 인해 모델이 제대로 학습하지 못할 수 O

⇒ 일괄적으로 적용 X, **출력된 그래프 결과를 잘 이해하고 해석하여 적절한 성능 향상 방안 적용 필요!!**

THANK YOU

