⚡ 지능 시스템 설계 Homework 5

> ***Dep: Electronics***
> ***ID: 2021220699***
> ***Name: Eunchan Lee***

What to do?

딥러닝 교재 [밑바닥부터 시작하는 딥러닝] Chapter 6. 학습 관련 기술들의 예제 코드들은 *MNIST dataset*에 대한 아래 내용들을 담고 있습니다.

- ***SGD, Momentum, AdaGrad, Adam을 비교***
- **가중치(W) 초기값 잘 정하는 방법**
- **배치 정규화**
- **오버피팅 탈출법 → 가중치 감소(weight decay), 드롭아웃(Dropout)**
- **적절한 하이퍼파라미터 [layer size, batch size, learning rate 등] 값 찾기**

> 💡 **[6장의 정리]**
> **이번 장에서 배운 것**
>
> **매개변수 갱신 방법에는 확률적 경사 하강법(SGD) 외에도 모멘텀, AdaGrad, Adam 등이 있음**
> **가중치 초기값을 정하는 방법은 올바른 학습을 하는데 매우 중요**
> **가중치의 초기값은 Xavier 초기값(Sigmoid, tanh)과 He 초기값(ReLU)이 효과적**
> **배치 정규화(normalization)를 이용하면 학습을 빠르게 진행, 초기값에 영향을 덜 받게 됨**
> **오버피팅을 억제하는 정규화(regularization) 기술로는 가중치 감소와 드롭아웃이 있음**
> **하이퍼파라미터 값 탐색은 최적 값이 존재할 법한 범위를 점차 좁히면서 하는 것이 효과적**

위의 예제 코드들을

- ***Fashion MNIST***
- ***Scikit-learn digits***

위 두가지 Dataset에 대해 적용하는 코드를 *ipynb* → *py*로 만들고

Code Review 등의 형식으로 Report화 하기

1. *for Fashion-MNIST* [code: *hw5-1.py*]

▶Fashion MNIST info @Eunchan Lee

To import by code

```
# tensorflow와 tf.keras를 임포트합니다
import tensorflow as tf
from tensorflow import keras
#Keras library makes it easy to use Fashion MNIST datasets.

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

Dataset info

```
train_images = (60000,28,28)  60000 images of 28x28
test_images  = (10000,28,28)  10000 images of 28x28
train_labels = (60000,)      60000 label values of 1D array
test_labels  = (10000,)      10000 label values of 1D array


label

0 ~ 9 exists

0 = T-shirt/top 1 = Trouser
2 = Pullover    3 = Dress
4 = Coat        5 = Sandal
6 = Shirt       7 = Sneaker
8 = Bag         9 = Ankle boot
```

▶FashionMnist에서 MNIST에 비해 바뀐 코드

데이터 전처리 부분을 수정해 주어야 합니다.

- MNIST 데이터 셋은 (,784)인 반면 f-MNIST는 (,28,28)의 형식으로 되어 있기 때문에 reshape 함수를 통해 데이터 사이즈를 Mnist와 같게 만들어 줍니다. 아래처럼 말이죠

```
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = train_images.reshape(60000,28*28)
test_images  = test_images.reshape(10000,28*28)
```

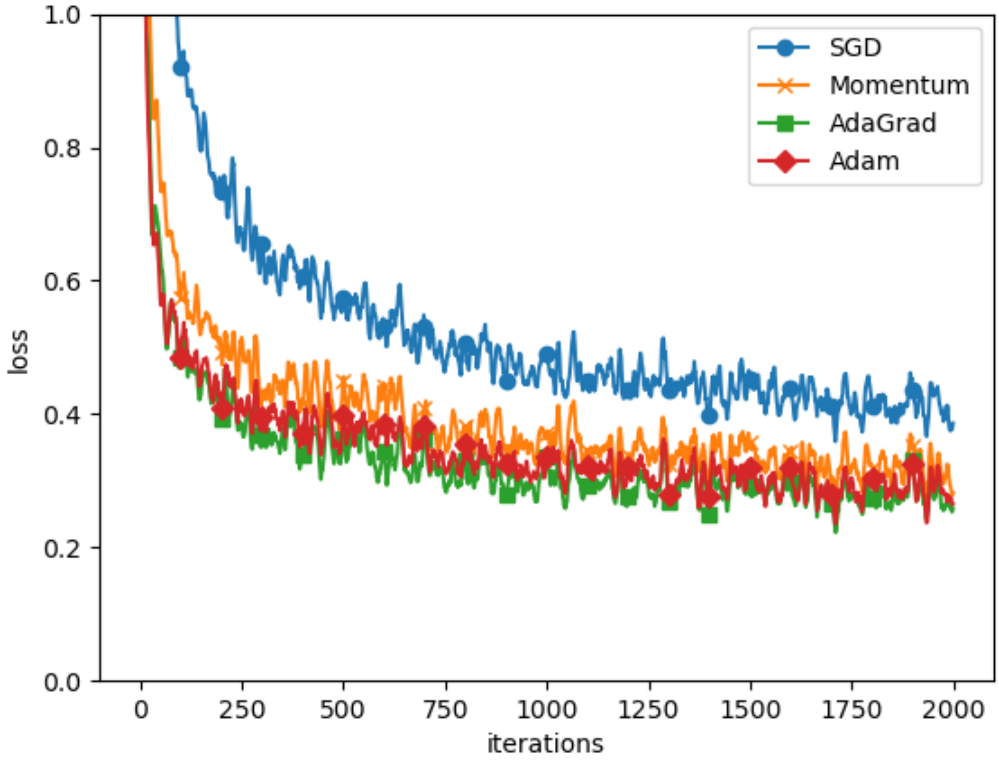
```
#Reduce dimensions
train_images = train_images/255
test_images  = test_images/255

x_train = train_images
t_train = train_labels
x_test = test_images
t_test = test_labels
```

▶Part 0. 네가지 갱신방법 비교

SGD, Momentum, AdaGrad, Adam을 비교

CODE OUTPUT

==========iteration:0==========
SGD:2.338572742435959
Momentum:2.3316449421782934
AdaGrad:2.881611633801733
Adam:2.055768785217421
==========iteration:100==========
SGD:0.9482711714689764
Momentum:0.5995801945127497
AdaGrad:0.5242184450962963
Adam:0.5048487965178724
==========iteration:200==========
SGD:0.7461409783302453
Momentum:0.45506252692848004
AdaGrad:0.349129780501647
Adam:0.36487599138942783
(중략)
==========iteration:1700==========
SGD:0.44319197113094777
Momentum:0.3020826523620053
AdaGrad:0.2677027688187607
Adam:0.3513885379316294
==========iteration:1800==========
SGD:0.33079525493022
Momentum:0.2758918136521546
AdaGrad:0.23363150963166643
Adam:0.2518663504734057
==========iteration:1900==========
SGD:0.3364835202096036
Momentum:0.3217723140700938
AdaGrad:0.26905973374100706
Adam:0.2882564819389687

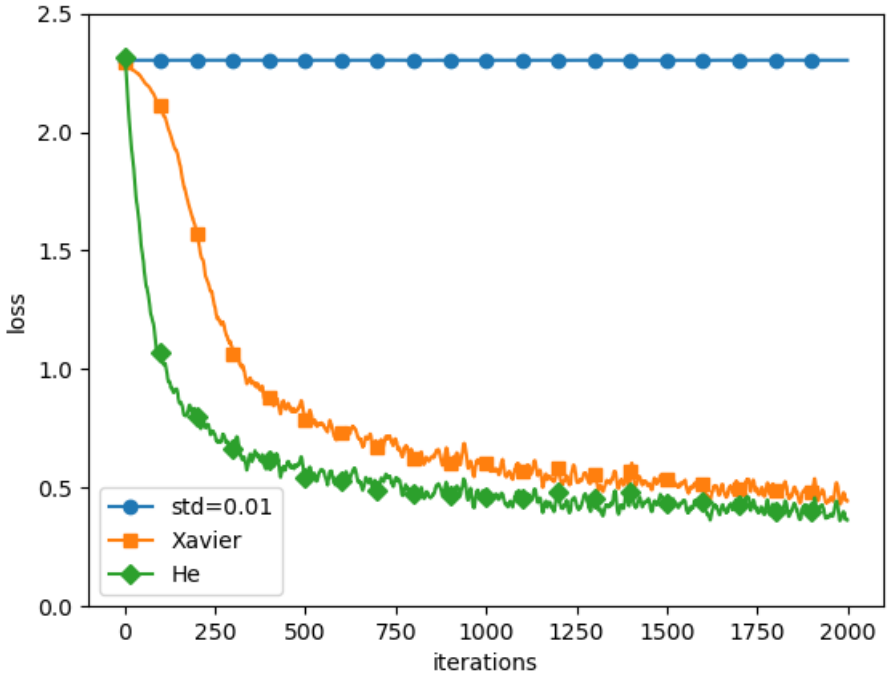


▶Part 1. Fashion-MNIST 데이터셋으로 본 가중치 초깃값

CODE OUTPUT

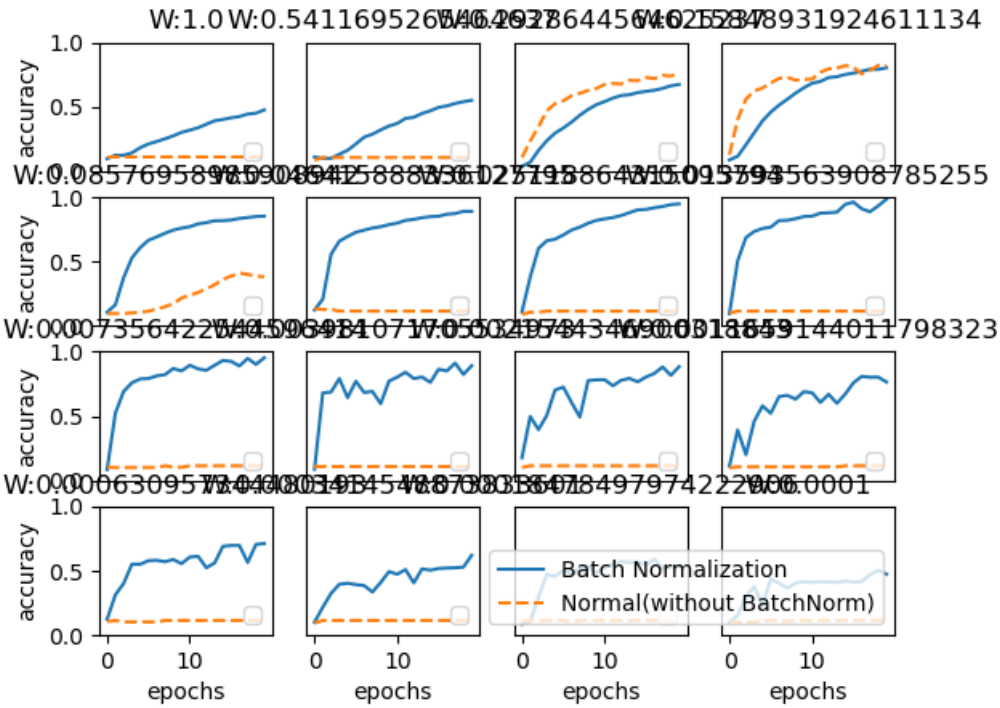
==========iteration:0==========
std=0.01:2.30247944118593
Xavier:2.290567794383307
He:2.3508048621074344
==========iteration:100==========
std=0.01:2.3023458873315774
Xavier:2.007403026114549
He:1.0289840993726291
==========iteration:200==========
std=0.01:2.302239350420974
Xavier:1.4418061896478063
He:0.7235701399284962
(중략)
==========iteration:1700==========
std=0.01:2.3032324527364483

Xavier:0.4217593098840762
He:0.3138726451045025
==========iteration:1800==========
std=0.01:2.3033039937454767
Xavier:0.5015770706215592
He:0.41694698942854447
==========iteration:1900==========
std=0.01:2.301549344852333
Xavier:0.4099202803436247
He:0.3158539571766924



결과: Fashion MNIST 데이터 적용에서도 초기 학습 속도는 He > Xavier 였고 마찬가지로 std=0.01에 대해서는 학습이 되지 않았음

▶Part 2. 배치 정규화



결과: Fashion MNIST 데이터 적용에서도 배치 정규화를 적용한 파란색 라인의 그래프가 Accuracy가 전체적으로 높았습니다.

▶Part 3. 오버피팅 탈출법 → 가중치 감소(weight decay), 드롭아웃(Dropout)

Part 3.1. 데이터 오버피팅 발생시키기

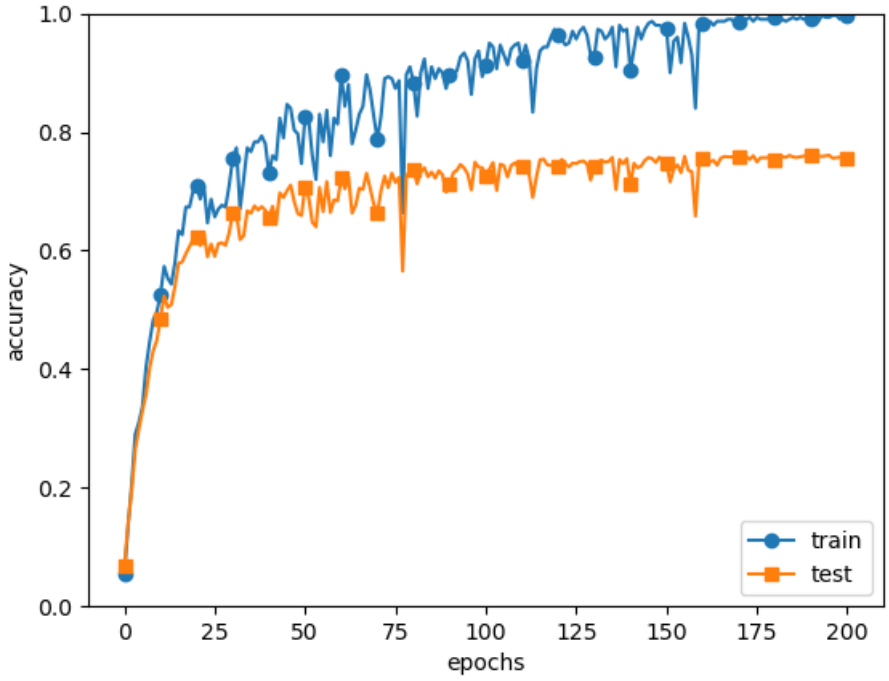
- 강제로 오버피팅을 만드는 코드에 ***Fashion MNIST***를 넣고 결과를 보기

CODE OUTPUT

epoch:0, train acc:0.05333333333333334, test acc:0.0673
epoch:1, train acc:0.13333333333333333, test acc:0.1375
epoch:2, train acc:0.20333333333333334, test acc:0.1938
epoch:3, train acc:0.29, test acc:0.2641

(중략)

epoch:198, train acc:0.99, test acc:0.7573
epoch:199, train acc:0.9966666666666667, test acc:0.7585
epoch:200, train acc:0.9966666666666667, test acc:0.7544



그래프에서 train-test의 정확도가 크게 벌어지는 것은 훈련 데이터에만 적응(fitting)한 결과임.

Fashion MNIST에 대해 오버피팅 잘 됨

명백한 증거는 train accuracy가 1을 찍었다는 것=100%의 정확도는 너무나 비현실적인 수치임(오버피팅)

Part 3.2. 가중치 감소를 통한 오버피팅 해소

- **가중치 감소를 통해 오버피팅을 해소하는 예제 코드를 Fashion MNIST에 적용해 보기**

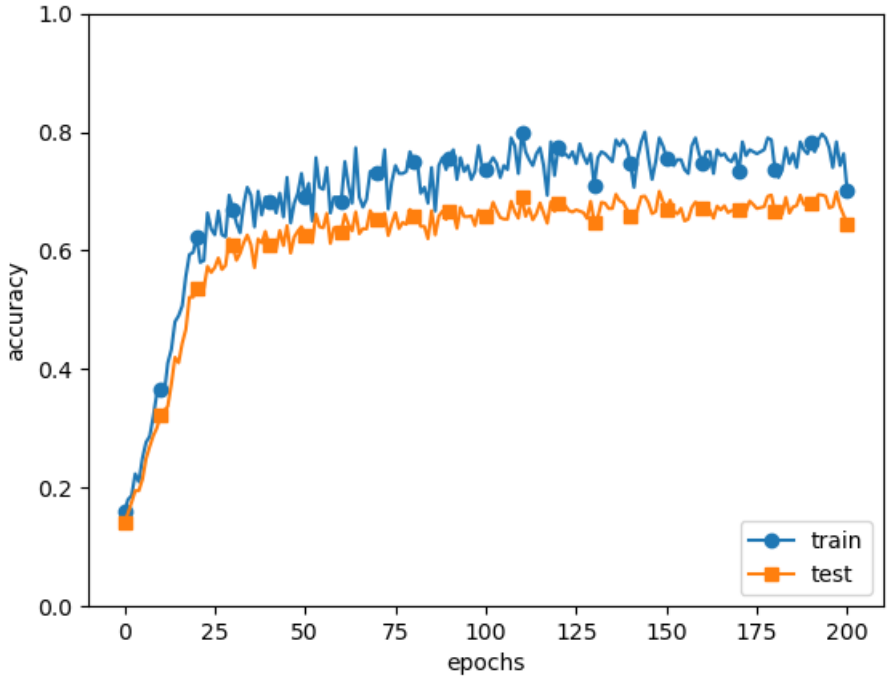
CODE OUTPUT

epoch:0, train acc:0.16, test acc:0.1414
epoch:1, train acc:0.18, test acc:0.1569

epoch:2, train acc:0.18666666666666668, test acc:0.1754
epoch:3, train acc:0.22333333333333333, test acc:0.1949

(중략)

epoch:197, train acc:0.7833333333333333, test acc:0.6987
epoch:198, train acc:0.7433333333333333, test acc:0.6745
epoch:199, train acc:0.7633333333333333, test acc:0.6598
epoch:200, train acc:0.7, test acc:0.6437



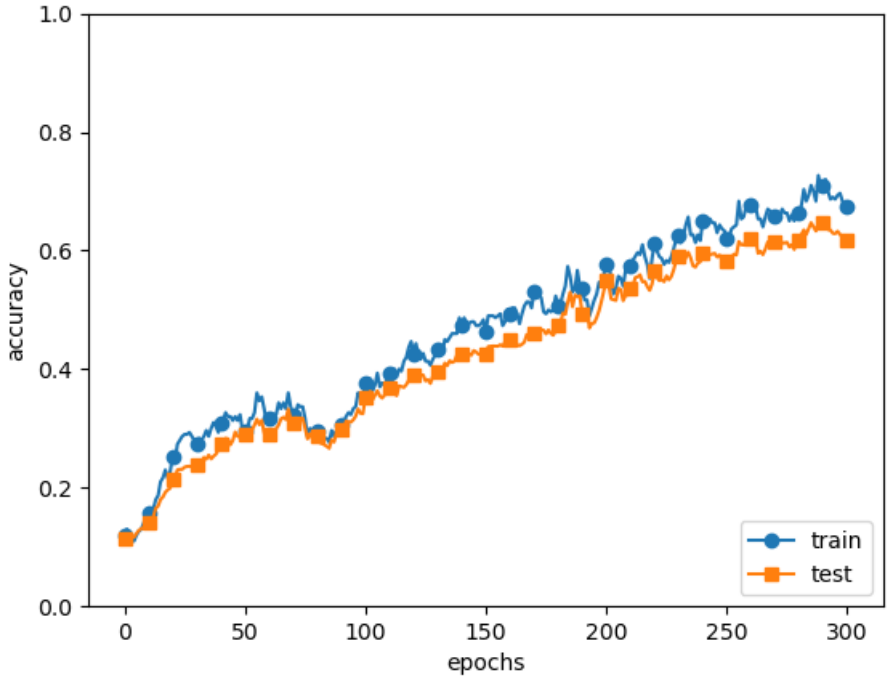
결과: F-MNIST 데이터도 mnist 예제와 마찬가지로 훈련 데이터에 대한 정확도와 시험 데이터에 대한 정확도는 더 줄었으나 train-test 사이의 정확도가 가까워졌다.(오버피팅이 억제됨)

Part 3.3. 드롭아웃을 통한 오버피팅 해소

- **드롭아웃을 통해 오버피팅을 잡는 예제 코드를 Fashion MNIST 데이터에도 적용해봄**

CODE OUTPUT

NO PRINT-LINE



결과:

***MNIST* 예제와 마찬가지로 F-Mnist 데이터 역시,**

드롭아웃을 적용하니 훈련 데이터와 시험 데이터에 대한 정확도 차이가 줄었고 드롭아웃을 이용하면 *표현력을 높이면서 오버피팅을 억제가능하다.*

▶Part 4. 적절한 하이퍼파라미터 값 찾기

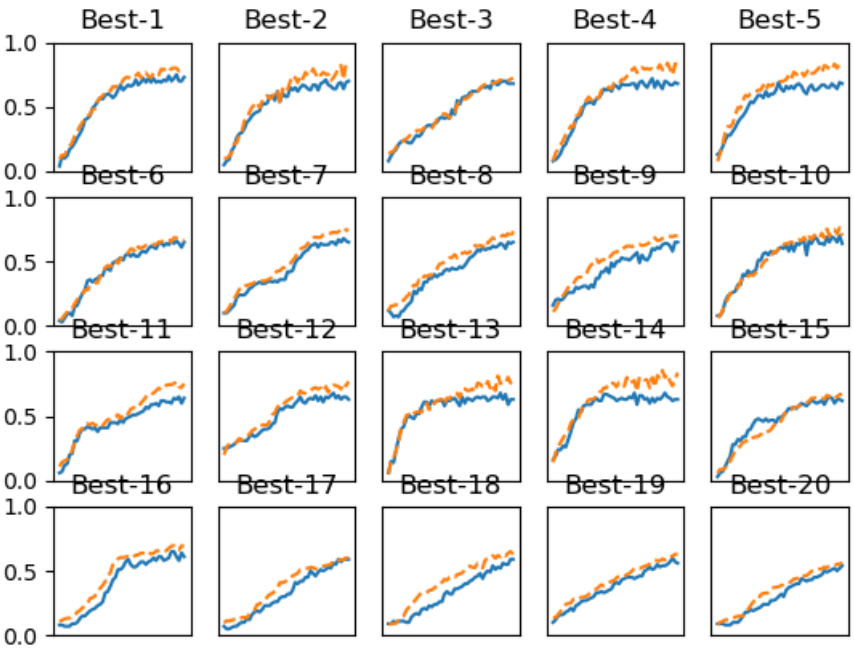
!?하이퍼파라미터

각 layer size, batch size, learning rate 등

CODE OUTPUT

val acc:0.65 | lr:0.004033105443126092, weight decay:3.7623876422381616e-06
val acc:0.22 | lr:0.00015213408248236778, weight decay:1.2981599245557e-08
val acc:0.11 | lr:0.00073560770004926, weight decay:2.2196004570280973e-08
val acc:0.36 | lr:0.0005539552911149701, weight decay:4.201017497921518e-06
val acc:0.18 | lr:0.00025294906268407444, weight decay:7.715698907585391e-07

val acc:0.2 | lr:0.00018067198643234475, weight decay:1.0629046021075823e-07
val acc:0.07 | lr:1.059763922380416e-06, weight decay:9.296592967915885e-06
(중략)
val acc:0.63 | lr:0.008715251414602509, weight decay:1.9403525570564994e-08
val acc:0.13 | lr:0.0001136334112603958, weight decay:2.595468087200897e-08
val acc:0.09 | lr:3.534844116200619e-06, weight decay:1.992151722206056e-07
val acc:0.05 | lr:1.5006309484927133e-06, weight decay:2.036886239711004e-08
val acc:0.05 | lr:5.010014660684901e-06, weight decay:1.6610184412802237e-05
val acc:0.15 | lr:0.00017475514203204695, weight decay:3.152902785463229e-05
val acc:0.59 | lr:0.0021431851389777847, weight decay:2.7849442336588078e-08
val acc:0.65 | lr:0.003931707546097702, weight decay:8.24538573644075e-08
val acc:0.16 | lr:1.5864578961347132e-05, weight decay:6.585769735374058e-05
val acc:0.05 | lr:1.7922730778293529e-06, weight decay:3.88229975353511e-06
val acc:0.15 | lr:8.162428939766814e-06, weight decay:2.0672605653508538e-05
========== Hyper-Parameter Optimization Result ==========
Best-1(val acc:0.73) | lr:0.008725857704941933, weight decay:5.3321135380693484e-08
Best-2(val acc:0.7) | lr:0.008865961367491513, weight decay:1.6070104629490852e-05
Best-3(val acc:0.68) | lr:0.0038690734796556895, weight decay:1.4971860610157802e-06
Best-4(val acc:0.68) | lr:0.006275329521413578, weight decay:8.089340785899472e-07
Best-5(val acc:0.68) | lr:0.006837535553010721, weight decay:2.6765997131868352e-05
Best-6(val acc:0.65) | lr:0.004033105443126092, weight decay:3.7623876422381616e-06
Best-7(val acc:0.65) | lr:0.004710210891857069, weight decay:4.740701658993109e-06
Best-8(val acc:0.65) | lr:0.004120317896197646, weight decay:5.4741876265802166e-06
Best-9(val acc:0.65) | lr:0.003931707546097702, weight decay:8.24538573644075e-08
Best-10(val acc:0.64) | lr:0.00648819247949308, weight decay:1.2559239468347054e-08
Best-11(val acc:0.64) | lr:0.003697168606858239, weight decay:1.7484403771013956e-08
Best-12(val acc:0.63) | lr:0.00619765701237026, weight decay:1.218382555386167e-08
Best-13(val acc:0.63) | lr:0.0077209198288607115, weight decay:2.1229103941722626e-07
Best-14(val acc:0.63) | lr:0.008715251414602509, weight decay:1.9403525570564994e-08
Best-15(val acc:0.62) | lr:0.003857943707425303, weight decay:1.1683755855400557e-06
Best-16(val acc:0.61) | lr:0.005082261262731077, weight decay:4.2537173070618346e-05
Best-17(val acc:0.59) | lr:0.0027233364547094995, weight decay:1.5735388744243827e-07
Best-18(val acc:0.59) | lr:0.0021431851389777847, weight decay:2.7849442336588078e-08
Best-19(val acc:0.56) | lr:0.002349709759228044, weight decay:6.2074457518607715e-06
Best-20(val acc:0.54) | lr:0.0025761140721383678, weight decay:4.1616726590893945e-06



결과: 주황선은 Train_data_acc, 파랑선은 Vali_data_acc이며, BEST-숫자의 의미는 훈련에서 가장 잘 된 순으로 정렬 된 것입니다. 상위 5개 항목에 대해 분석해보면 적절한 하이퍼파라미터의 범위를 고를 수 있습니다. 그 값은 어떻게 될까요?

▶ ***적절한 하이퍼 파라미터 값(대략적인 범위):***

- **학습률(lr): 0.0038 ~ 0.0088**
- **가중치 감소계수(weight decay):** $5.3 * 10^{-8} \sim 2.67 * 10^{-5}$

Best-1(val acc:0.73) | lr:0.008725857704941933, weight decay:5.3321135380693484e-08
Best-2(val acc:0.7) | lr:0.008865961367491513, weight decay:1.6070104629490852e-05
Best-3(val acc:0.68) | lr:0.0038690734796556895, weight decay:1.4971860610157802e-06
Best-4(val acc:0.68) | lr:0.006275329521413578, weight decay:8.089340785899472e-07
Best-5(val acc:0.68) | lr:0.006837535553010721, weight decay:2.6765997131868352e-05

2. *for sklearn-digits* [code: *hw5-2.py*]

▶Part 0. 네가지 갱신방법 비교

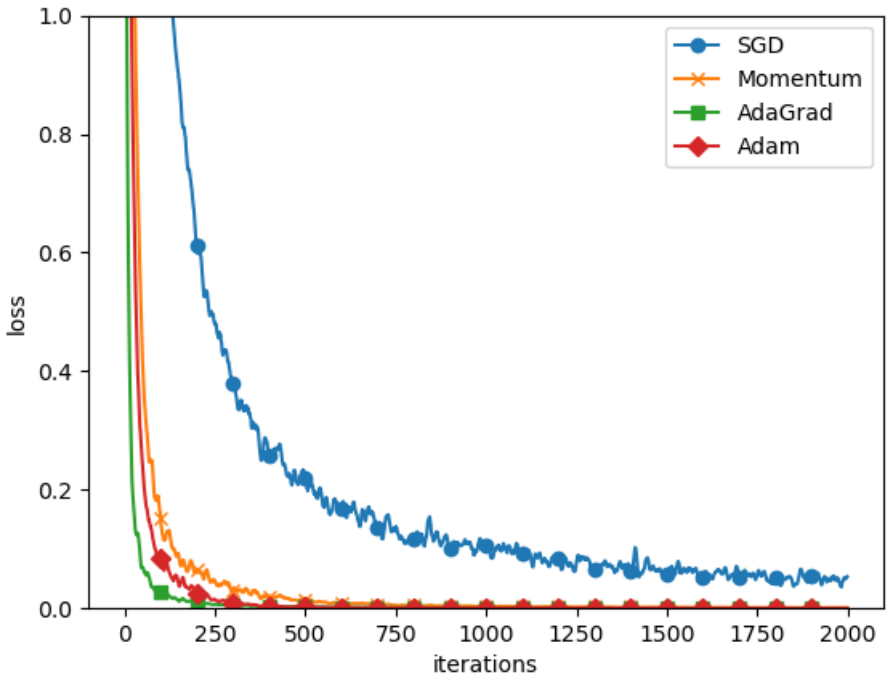
SGD, Momentum, AdaGrad, Adam을 비교

CODE OUTPUT

==========iteration:0==========
SGD:2.2994419906666512
Momentum:2.3977519998976256
AdaGrad:2.1029783113922327
Adam:2.3489094522858416
==========iteration:100==========
SGD:1.1948926232730746
Momentum:0.12116370627068467
AdaGrad:0.018021243733671623
Adam:0.05516304837635172
==========iteration:200==========
SGD:0.5671649975034307
Momentum:0.05605184620458264
AdaGrad:0.011148037963119934
Adam:0.03320270395050573

(중략)

==========iteration:1700==========
SGD:0.05365048664887991
Momentum:0.0011896145420766994
AdaGrad:0.0003869430363490381
Adam:7.85938257278033e-05
==========iteration:1800==========
SGD:0.05324706385635931
Momentum:0.0016138777969438604
AdaGrad:0.00046188396831097075
Adam:0.00011202757169856569
==========iteration:1900==========
SGD:0.07052380847764084
Momentum:0.0012828344611217782
AdaGrad:0.00027117099242014107
Adam:6.478180149242527e-05



▶Part 1. Digits 데이터셋으로 본 가중치 초깃값

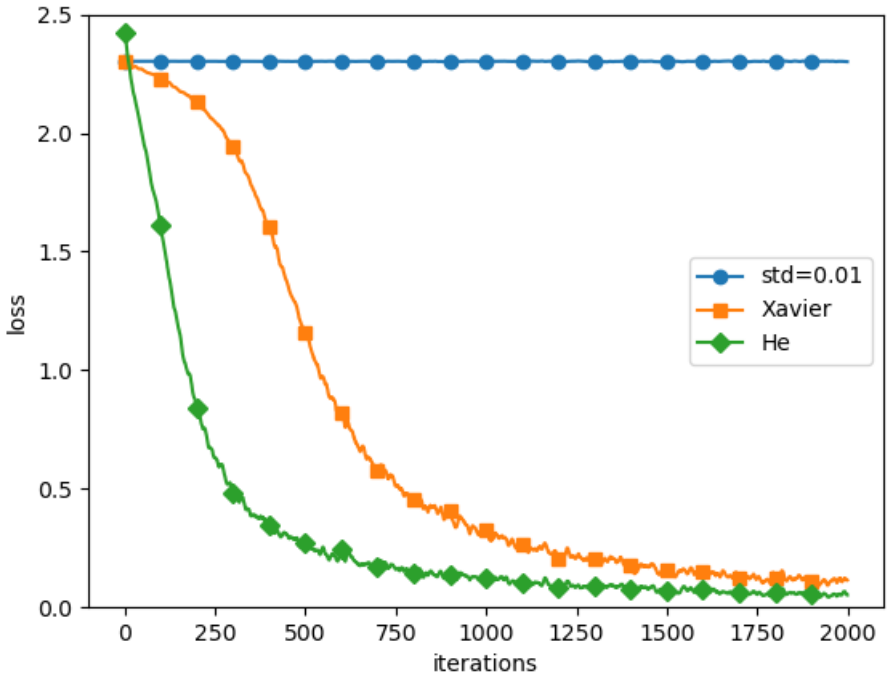
CODE OUTPUT

==========iteration:0==========
std=0.01:2.302515458895547
Xavier:2.3107350107755495
He:2.4460560955427013
==========iteration:100==========
std=0.01:2.302483043051067
Xavier:2.2285779749203734
He:1.59874566697205
==========iteration:200==========
std=0.01:2.3025126500905
Xavier:2.1279954610365035
He:0.8138986344706441

(중략)

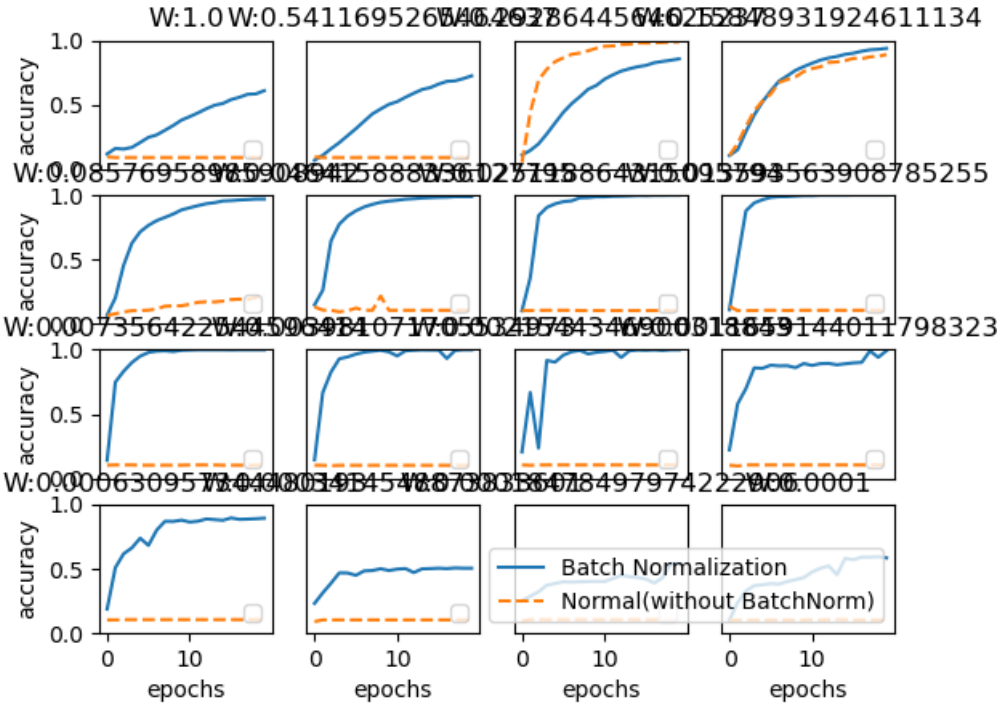
==========iteration:1700==========
std=0.01:2.3067335953016155
Xavier:0.10733457038669975
He:0.04379060359117231
==========iteration:1800==========
std=0.01:2.308953409741611

Xavier:0.07807213063933505
He:0.034102793078299255
==========iteration:1900==========
std=0.01:2.3046152496782666
Xavier:0.060769342635440135
He:0.03217016329290266



결과: digits 데이터 적용에서도 초기 학습 속도는 He > Xavier 였고 마찬가지로 std=0.01 에 대해서는 학습이 되지 않았음

▶Part 2. 배치 정규화



결과: digits 데이터 적용에서도 배치 정규화를 적용한 파란색 라인의 그래프가 Accuracy가 전체적으로 높았습니다.

▶Part 3. 오버피팅 탈출법 → 가중치 감소(weight decay), 드롭아웃(Dropout)

Part 3.1. 데이터 오버피팅 발생시키기

- **강제로 오버피팅을 만드는 코드에 digits를 넣고 결과를 보기**

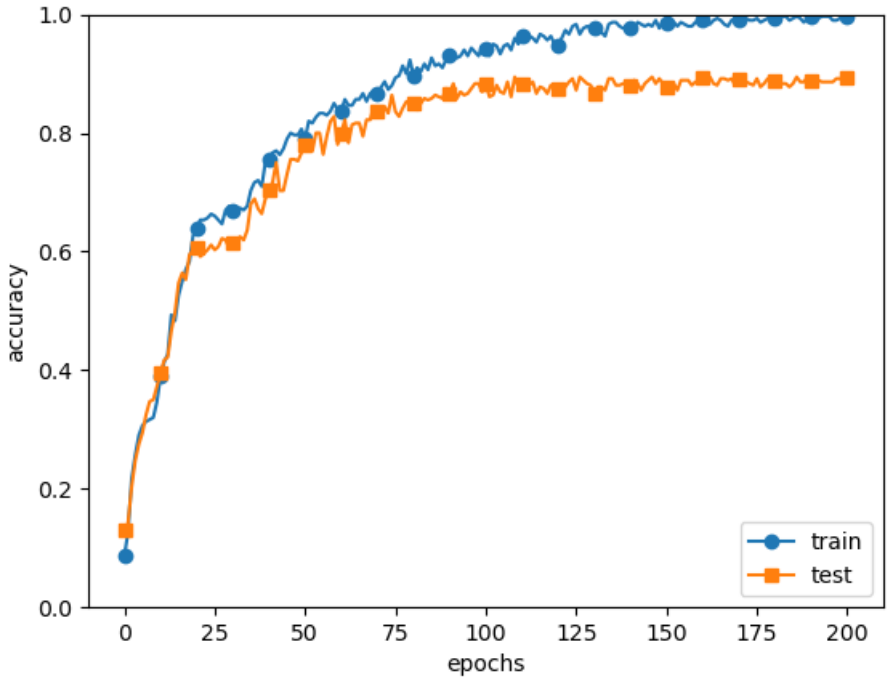
CODE OUTPUT

epoch:0, train acc:0.08666666666666667, test acc:0.13055555555555556
epoch:1, train acc:0.12666666666666668, test acc:0.14166666666666666

epoch:2, train acc:0.21666666666666667, test acc:0.20277777777777778
epoch:3, train acc:0.25666666666666665, test acc:0.24722222222222223

(중략)

epoch:198, train acc:0.9933333333333333, test acc:0.8916666666666667
epoch:199, train acc:0.9966666666666667, test acc:0.8861111111111111
epoch:200, train acc:0.9966666666666667, test acc:0.8944444444444445



그래프에서 train-test의 정확도가 크게 벌어지는 것은 훈련 데이터에만 적응(fitting)한 결과임.

***digits*에 대해 오버피팅 잘 됨**

명백한 증거는 train accuracy가 1을 찍었다는 것=100%의 정확도는 너무나 비현실적인 수치임(오버피팅)

Part 3.2. 가중치 감소를 통한 오버피팅 해소

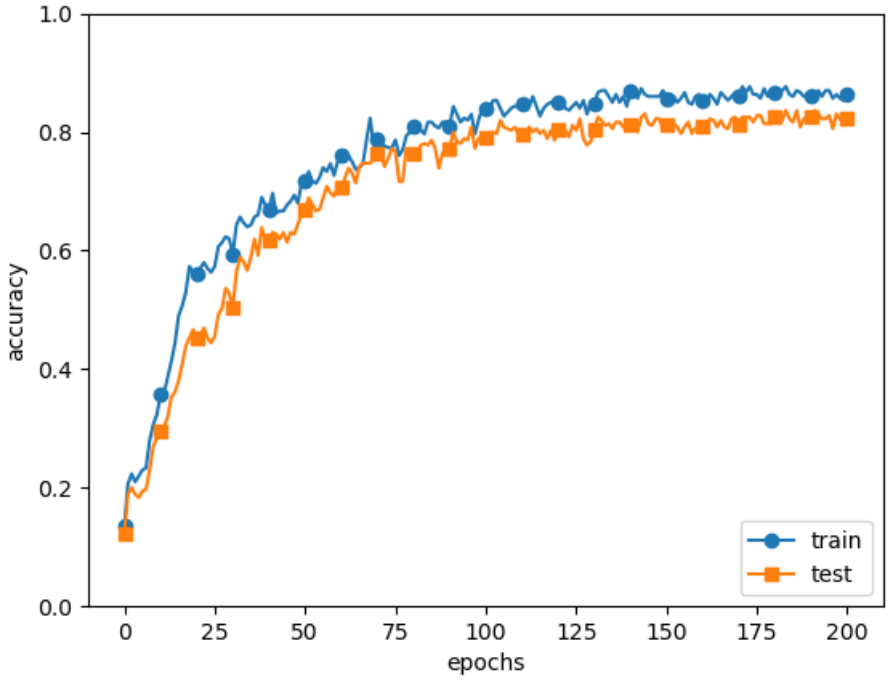
- **가중치 감소를 통해 오버피팅을 해소하는 예제 코드를 sklearn Digits에 적용해 보기**

CODE OUTPUT

epoch:0, train acc:0.13666666666666666, test acc:0.12222222222222222
epoch:1, train acc:0.2066666666666667, test acc:0.18888888888888888
epoch:2, train acc:0.22333333333333333, test acc:0.2
epoch:3, train acc:0.21, test acc:0.18888888888888888

(중략)

epoch:198, train acc:0.8566666666666667, test acc:0.8222222222222222
epoch:199, train acc:0.8533333333333334, test acc:0.8111111111111111
epoch:200, train acc:0.8633333333333333, test acc:0.8222222222222222



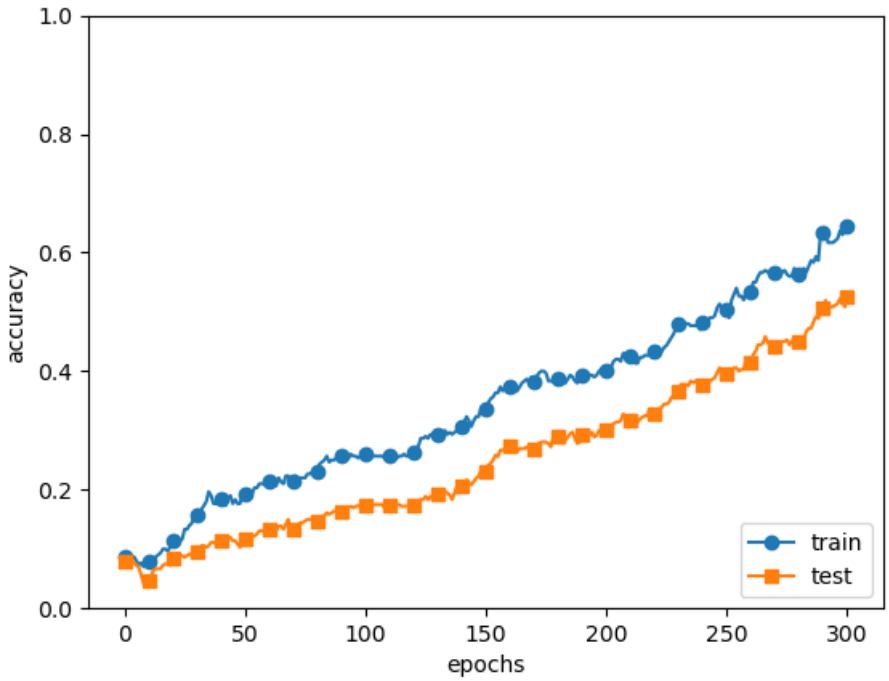
결과: digits 데이터도 mnist/Fashion-MNIST 예제와 마찬가지로 훈련 데이터에 대한 정확도와 시험 데이터에 대한 정확도는 더 줄었으나 train-test 사이의 정확도가 가까워졌다.(오버피팅이 억제됨)

Part 3.3. 드롭아웃을 통한 오버피팅 해소

- **드롭아웃을 통해 오버피팅을 잡는 예제 코드를 Digits 데이터에도 적용해봄**

CODE OUTPUT

NO PRINT-LINE



결과:

***MNIST* 예제와 마찬가지로 digits 데이터 역시,**

드롭아웃을 적용하니 훈련 데이터와 시험 데이터에 대한 정확도 차이가 줄었고 드롭아웃을 이용하면 *표현력을 높이면서 오버피팅을 억제가능하다.*

▶Part 4. 적절한 하이퍼파라미터 값 찾기

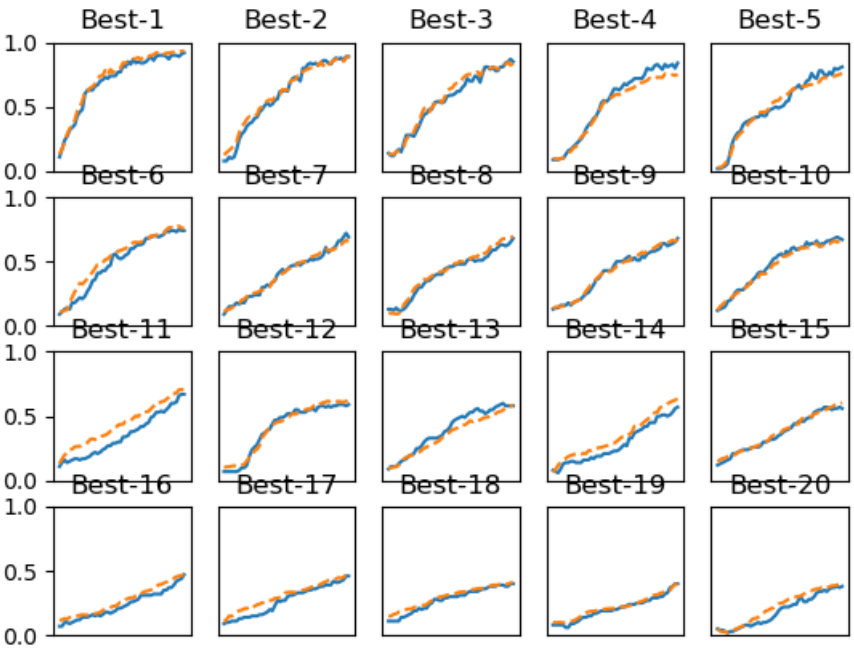
!?하이퍼파라미터

각 layer size, batch size, learning rate 등

CODE OUTPUT

val acc:0.12 | lr:0.0001192109668319687, weight decay:3.2086517469117953e-06
val acc:0.08 | lr:2.418748721424833e-06, weight decay:7.396559226175216e-08
val acc:0.12 | lr:4.836102571212388e-05, weight decay:1.8794764983027266e-07
val acc:0.05 | lr:1.0465364285202407e-06, weight decay:1.6100119399914457e-05
val acc:0.05 | lr:6.682571683835225e-06, weight decay:1.2706245533578214e-05
(중략)
val acc:0.89 | lr:0.00802161536272497, weight decay:4.401467050245399e-06
val acc:0.01 | lr:5.486097038440444e-05, weight decay:5.0835679312960357e-08
val acc:0.58 | lr:0.0027172804773943405, weight decay:2.7437585459725016e-06

val acc:0.09 | lr:3.124776743681842e-06, weight decay:9.593133520324551e-07
val acc:0.12 | lr:9.379489823073218e-06, weight decay:7.780639032278238e-05
val acc:0.05 | lr:0.0003207619617403599, weight decay:2.2227007375380647e-08
========== Hyper-Parameter Optimization Result ==========
Best-1(val acc:0.92) | lr:0.009743329553508515, weight decay:2.575490423654973e-05
Best-2(val acc:0.89) | lr:0.00802161536272497, weight decay:4.401467050245399e-06
Best-3(val acc:0.85) | lr:0.006434075591694893, weight decay:5.430558760397632e-06
Best-4(val acc:0.84) | lr:0.004026596110068881, weight decay:9.464155899122225e-05
Best-5(val acc:0.81) | lr:0.005502661705761284, weight decay:3.8838173503005467e-07
Best-6(val acc:0.74) | lr:0.00485269530718975, weight decay:1.3676319104508411e-08
Best-7(val acc:0.69) | lr:0.004349678056007648, weight decay:5.511895455119525e-05
Best-8(val acc:0.68) | lr:0.003895716224832331, weight decay:3.200200435093347e-05
Best-9(val acc:0.68) | lr:0.0020783063681908565, weight decay:8.365502318818075e-06
Best-10(val acc:0.67) | lr:0.004008241127526191, weight decay:3.58697393128988e-05
Best-11(val acc:0.67) | lr:0.0037288875526938266, weight decay:7.151681686274684e-06
Best-12(val acc:0.59) | lr:0.003979247389699313, weight decay:5.310871610092756e-08
Best-13(val acc:0.58) | lr:0.0027172804773943405, weight decay:2.7437585459725016e-06
Best-14(val acc:0.57) | lr:0.0019132349211672685, weight decay:8.797171312176448e-07
Best-15(val acc:0.56) | lr:0.0012086095793351736, weight decay:2.359372860659198e-08
Best-16(val acc:0.47) | lr:0.0013116639224658201, weight decay:8.718183022136018e-06
Best-17(val acc:0.46) | lr:0.0022448358457952417, weight decay:1.5403590543624657e-05
Best-18(val acc:0.4) | lr:0.0011098936543028788, weight decay:7.678214057227183e-06
Best-19(val acc:0.4) | lr:0.0018931376116616866, weight decay:2.4187513574668546e-07
Best-20(val acc:0.38) | lr:0.0013708366504698334, weight decay:1.1769332419627744e-07



결과: 주황선은 Train_data_acc, 파랑선은 Vali_data_acc이며, BEST-숫자의 의미는 훈련에서 가장 잘 된 순으로 정렬 된 것입니다. 상위 5개 항목에 대해 분석해보면 적절한 하이퍼파

라미터의 범위를 고를 수 있습니다. 그 값은 어떻게 될까요?

▶ 적절한 하이퍼 파라미터 값*(대략적인 범위)*:

- **학습률(lr): 0.0040 ~ 0.0097**
- **가중치 감소계수(weight decay):** $3.88 * 10^{-7}$~ $9.46 * 10^{-5}$

Best-1(val acc:0.92) | lr:0.009743329553508515, weight decay:2.575490423654973e-05
Best-2(val acc:0.89) | lr:0.00802161536272497, weight decay:4.401467050245399e-06
Best-3(val acc:0.85) | lr:0.006434075591694893, weight decay:5.430558760397632e-06
Best-4(val acc:0.84) | lr:0.004026596110068881, weight decay:9.464155899122225e-05
Best-5(val acc:0.81) | lr:0.005502661705761284, weight decay:3.8838173503005467e-07

🍀결론

데이터 상관없이 데이터를 잘 분석해서 데이터에 맞게 Neural Network의 모델링을 잘하면 잘 된다...

이 과제를 통해서 단순한 딥러닝 오픈소스 모델 이용에 그치치 않고

- **모델의 하이퍼파라미터를 잘 짜는 법**
- **모델의 오버피팅을 해소하는 몇가지 기법**
- **모델의 옵티마이저를 고르는 기준**

등을 실제로 프로그래밍 시에 깐깐히 생각하면서 더 효율을 높이면서 프로그래밍해 볼 수 있을 것 같다.

(와 딥러닝 너무 재밌다....😂😂)