교육부 Ministry of Education

NRF 한국연구재단 National Research Foundation of Korea

인공지능시스템

batchnorm, Cifar100

Convergence and Open Sharing System COSS 인공지능

전남대학교 CHONNAM NATIONAL UNIVERSITY

성균관대학교 SUNG KYUN KWAN UNIVERSITY

KNU 경북대학교 KYUNGPOOK NATIONAL UNIVERSITY

전주대학교 JEONJU UNIVERSITY

서울시립대학교 UNIVERSITY OF SEOUL

서울과학기술대학교 SEOUL NATIONAL UNIVERSITY OF SCIENCE & TECHNOLOGY

영진전문대학교 YEUNGJIN UNIVERSITY

Contents

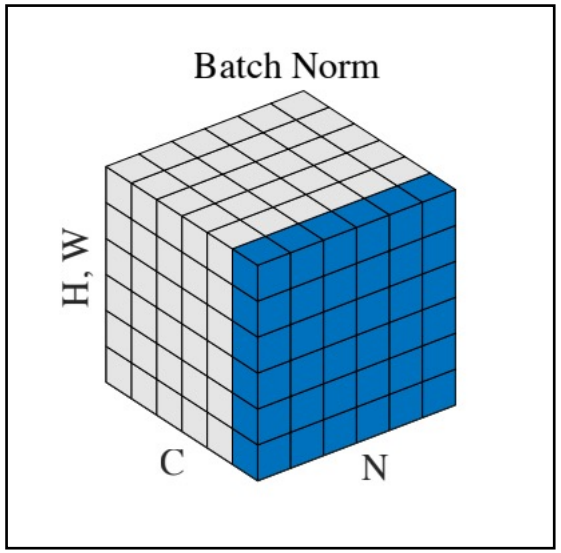
- **BatchNorm**

BatchNorm

Batch normalization (also known as **batch norm**) is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling.

- Wikipedia(https://en.wikipedia.org/wiki/Batch_normalization)

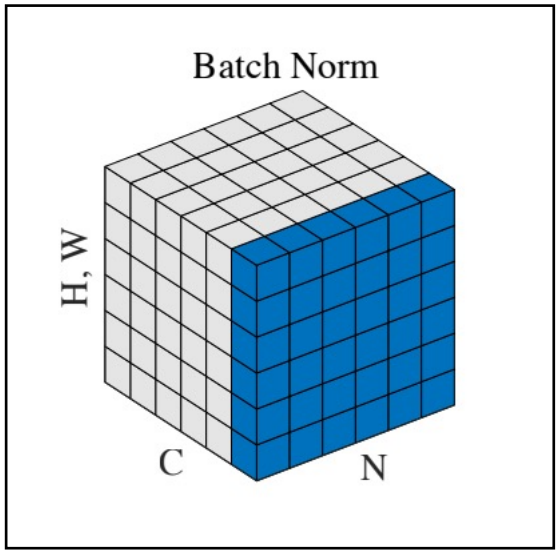
Feature map을 re-centering 및 re-scaling으로 normalization 하여 학습 속도를 안정화 및 가속하는 기법.



Batch Normalization 예시 그림

Group Normalization(Yuxin Wu, Kaiming He)

BatchNorm



Batch Normalization 예시 그림

Group Normalization(Yuxin Wu, Kaiming He)

해당 이미지가 많은 정보를 알려주긴 하지만, 처음 봤을 땐 이해하기 힘들며
생략된 정보가 많기에 Batch Normalization에 대해 자세히 알아보자

arXiv
https://arxiv.org › cs

Batch Normalization: Accelerating Deep Network Training ...

S Ioffe 저술 · 2015 · 50599회 인용 — Access **Paper**: Download a PDF of the **paper** titled **Batch Normalization**: Accelerating Deep Network Training by Reducing Internal Covariate ...

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

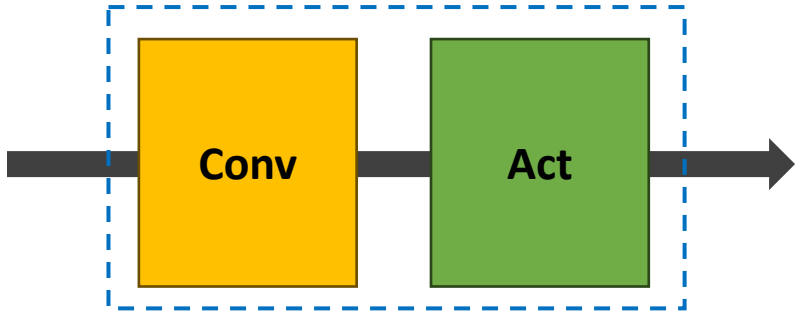
Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer in-

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

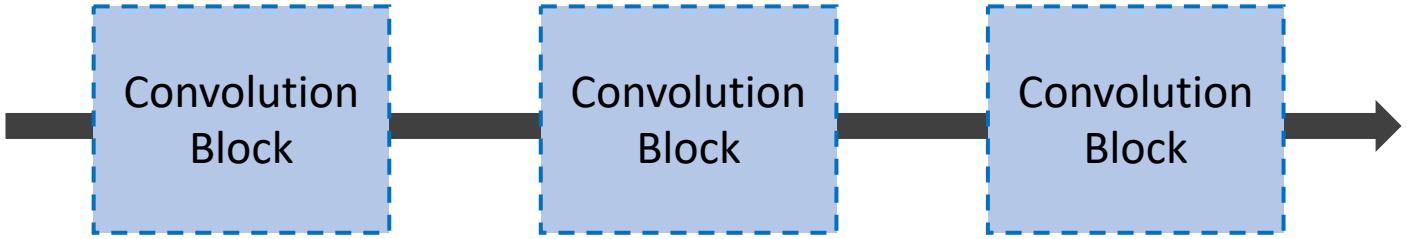
While stochastic gradient is simple and effective, it requires careful tuning of the model hyper-parameters

- **Why was Batch Normalization proposed? → Internal Covariate Shift**
  - Neural Network는 weight를 곱하고 bias를 더한 후, activation 하는 과정의 반복
  - layer를 통과한 output feature는 input feature 와 다른 새로운 분포를 갖게 된다



A convolution block

- **Why was Batch Normalization proposed? → Internal Covariate Shift**
  - Neural Network는 weight를 곱하고 bias를 더한 후, activation 하는 과정의 반복
  - layer를 통과한 output feature는 input feature 와 다른 새로운 분포를 갖게 된다
  - 하지만 이 과정이 반복될수록 feature의 분포 변화가 쌓여가며 앞쪽 layer에서 작은 변화가 뒷 layer에서 feature 분포에서 큰 변화를 만들게 됨.

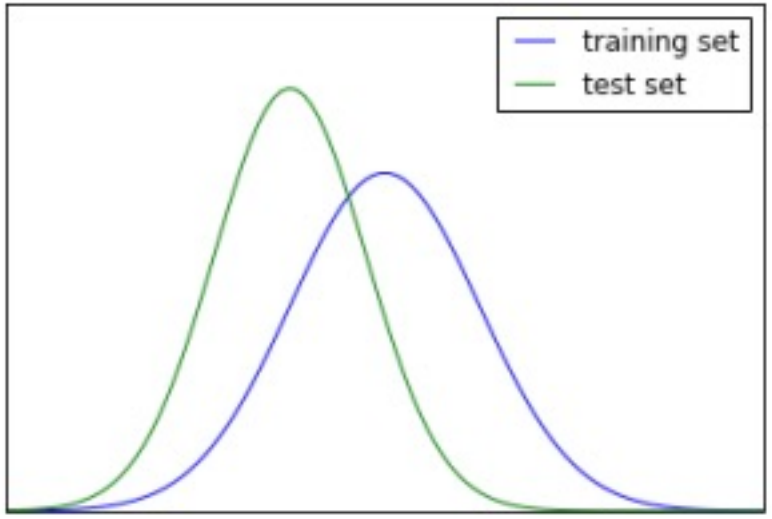


A convolution neural network

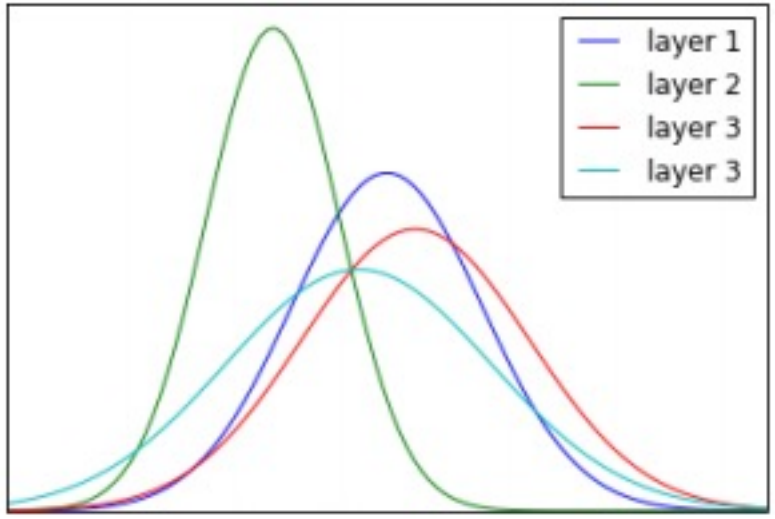
▪ Internal Covariate Shift

Machine learning에서 일반적으로..

- Covariate shift 는 train/test data 간의 분포 차이를 의미하고
- Internal covariate shift 는 각 layer를 통과한 hidden feature의 분포 차이를 의미한다.

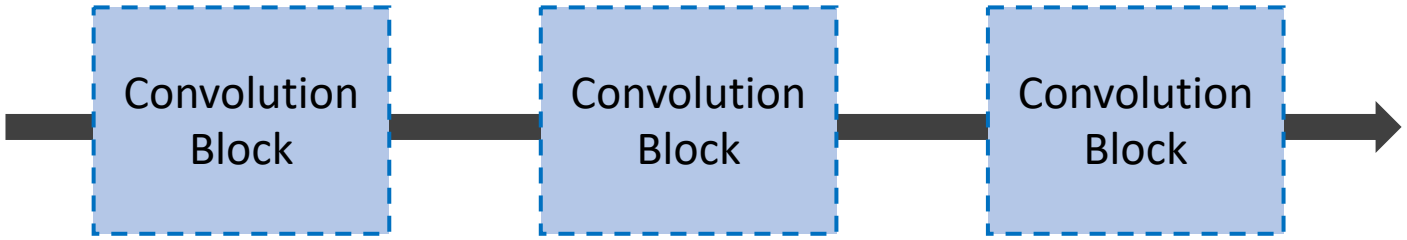


Covariate shift



Internal covariate shift

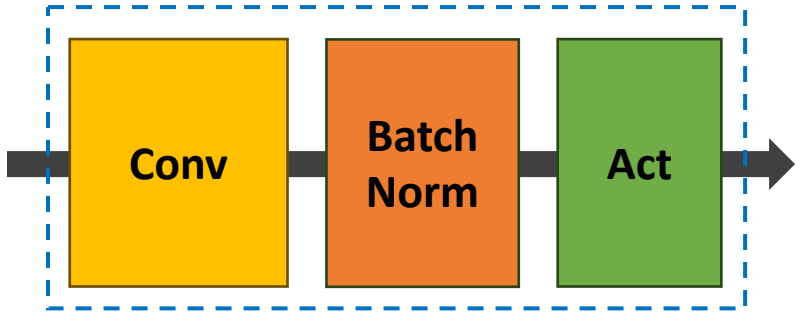
- **Straight forward remedies for Internal Covariate Shift**
  - 정교한 파라미터 초기화 → **Difficult**
  - 작은 Learning rate 적용 → **Slow learning**



A convolution neural network

▪ Idea of Batch Normalization (BatchNorm)

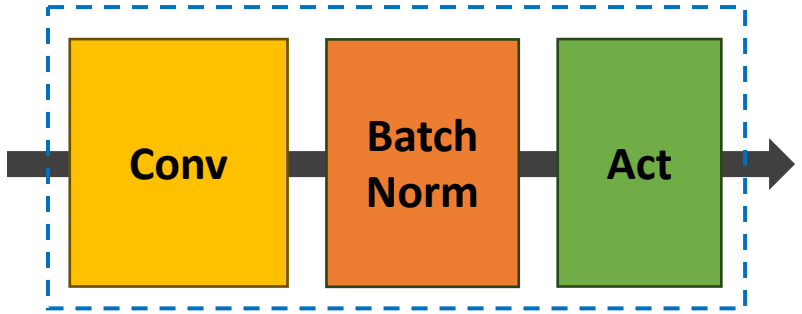
- Activation 전에 Feature map을 normalize 하여 이상적인 분포로 만들어 보자
- Network의 layer가 deep하더라도 **분포가 일정하게 유지되면서 학습의 난이도를 낮춰 줄 것을 기대**



A convolution block with batch normalization

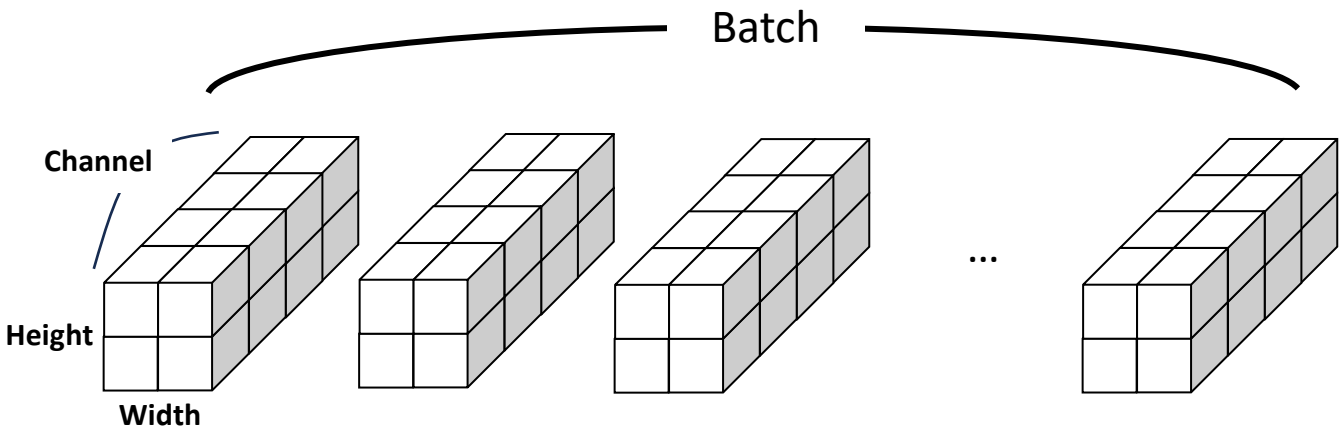
▪ Procedure of BatchNorm

1. Feature map의 통계량 (평균, 분산) 계산
2. 1 에서 계산한 통계량으로 Feature map을 Normalization
3. Learnable parameter(γ, β)로 다시 scaling 및 shifting



A convolution block with batch normalization

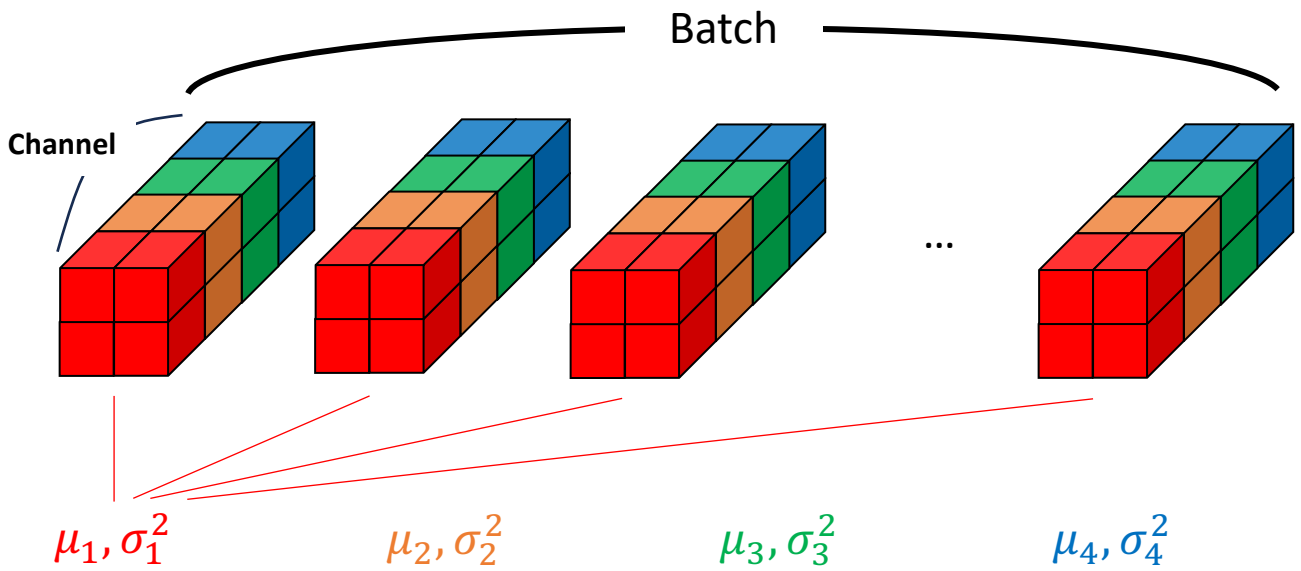
1. Feature map의 통계량 (평균, 분산) 계산



Convolution을 거친 (Channel * Height * Width)의 Feature map이 Batch 개 만큼 들어옴.

-> 위 예시에서는 (Batch, 4, 2, 2) 크기의 Feature map

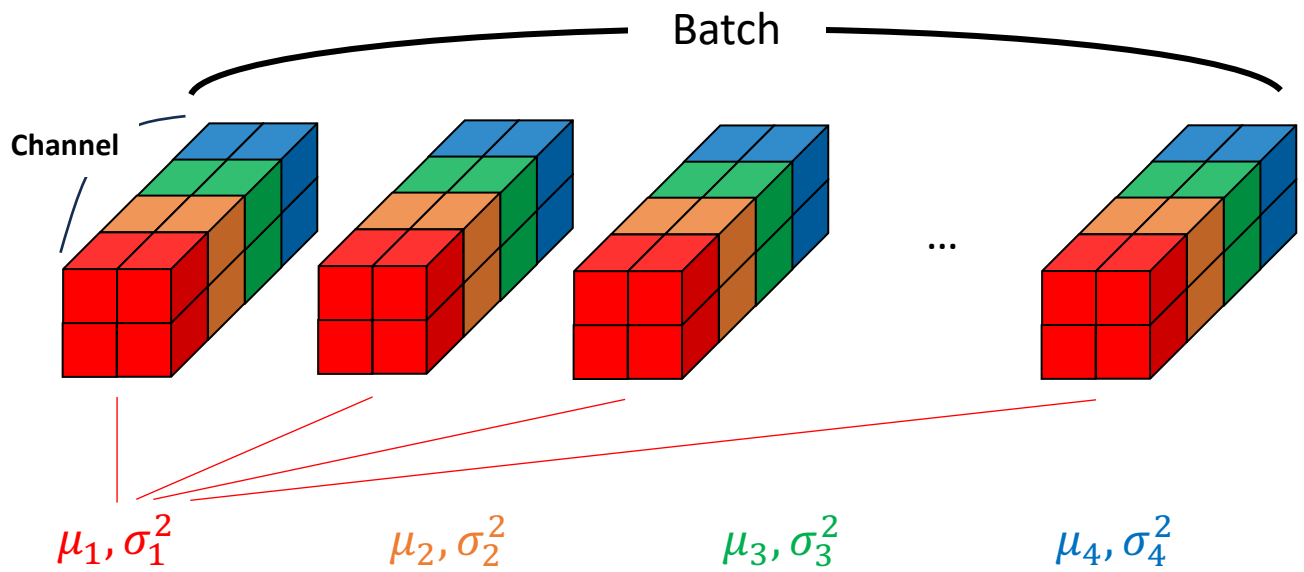
1. Feature map의 통계량 (평균, 분산) 계산



Channel-Wise로 Feature의 통계량(평균, 분산)을 계산함.

-> 위 예시에서는 Channel이 4개였기에 4개의 평균과 분산을 구함.

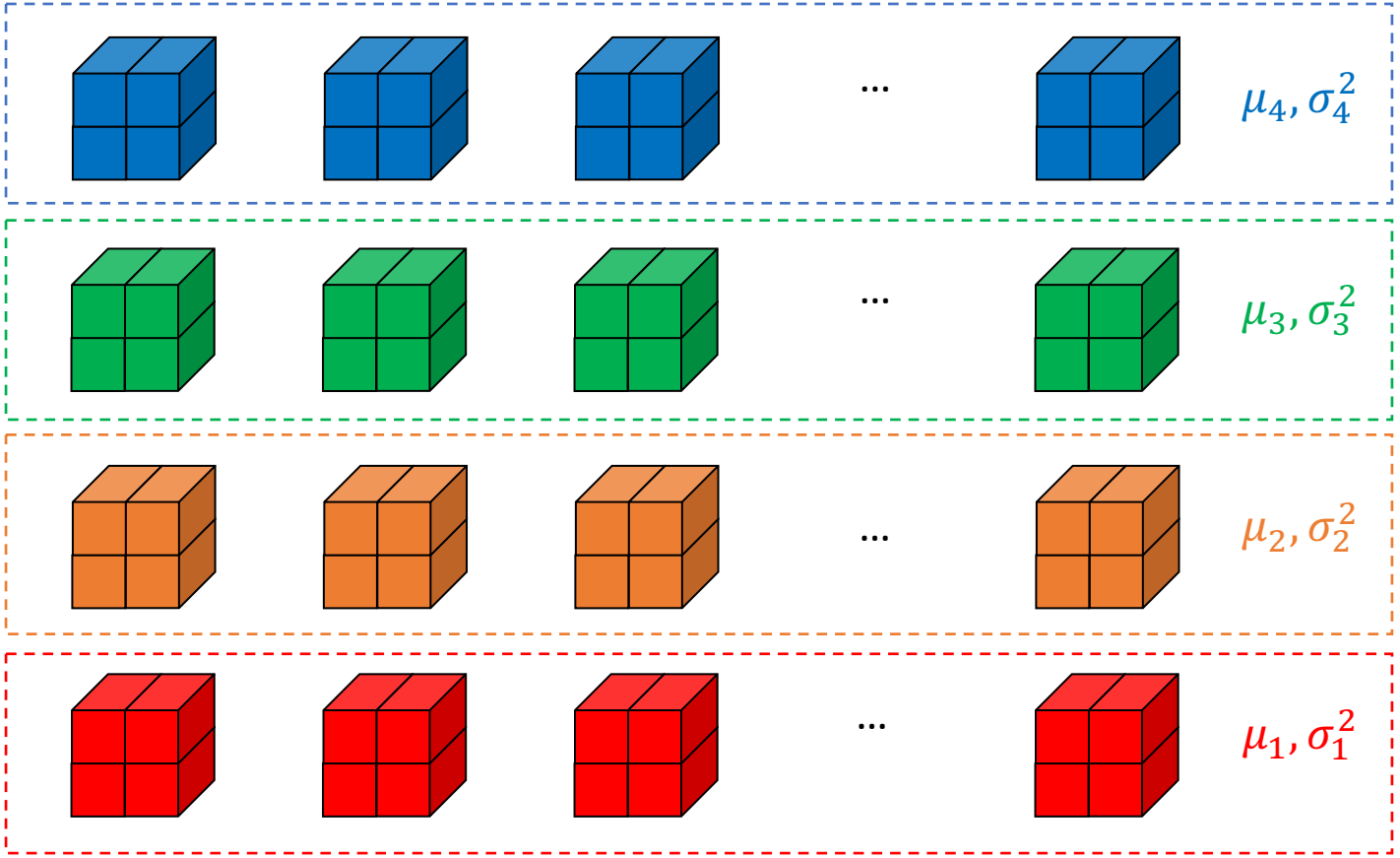
2. 통계량으로 Feature map을 Normalization



Channel-Wise로 Feature의 통계량(평균, 분산)을 계산함.

-> 위 예시에서는 Channel이 4개였기에 4개의 평균과 분산을 구함.

2. 통계량으로 Feature map을 Normalization

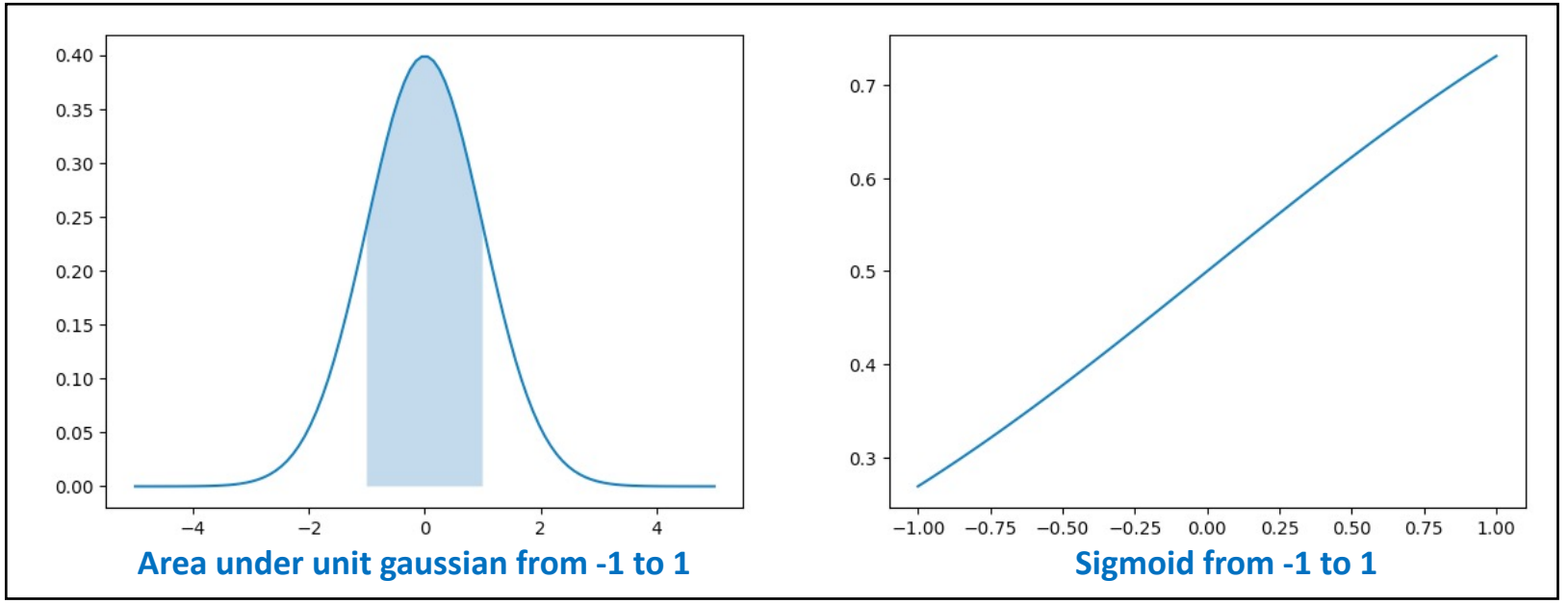


Channel-Wise로 Normalize.
0으로 나누는 것을 방지하기 위해 variance에 작은 숫자(ϵ)를 더함.

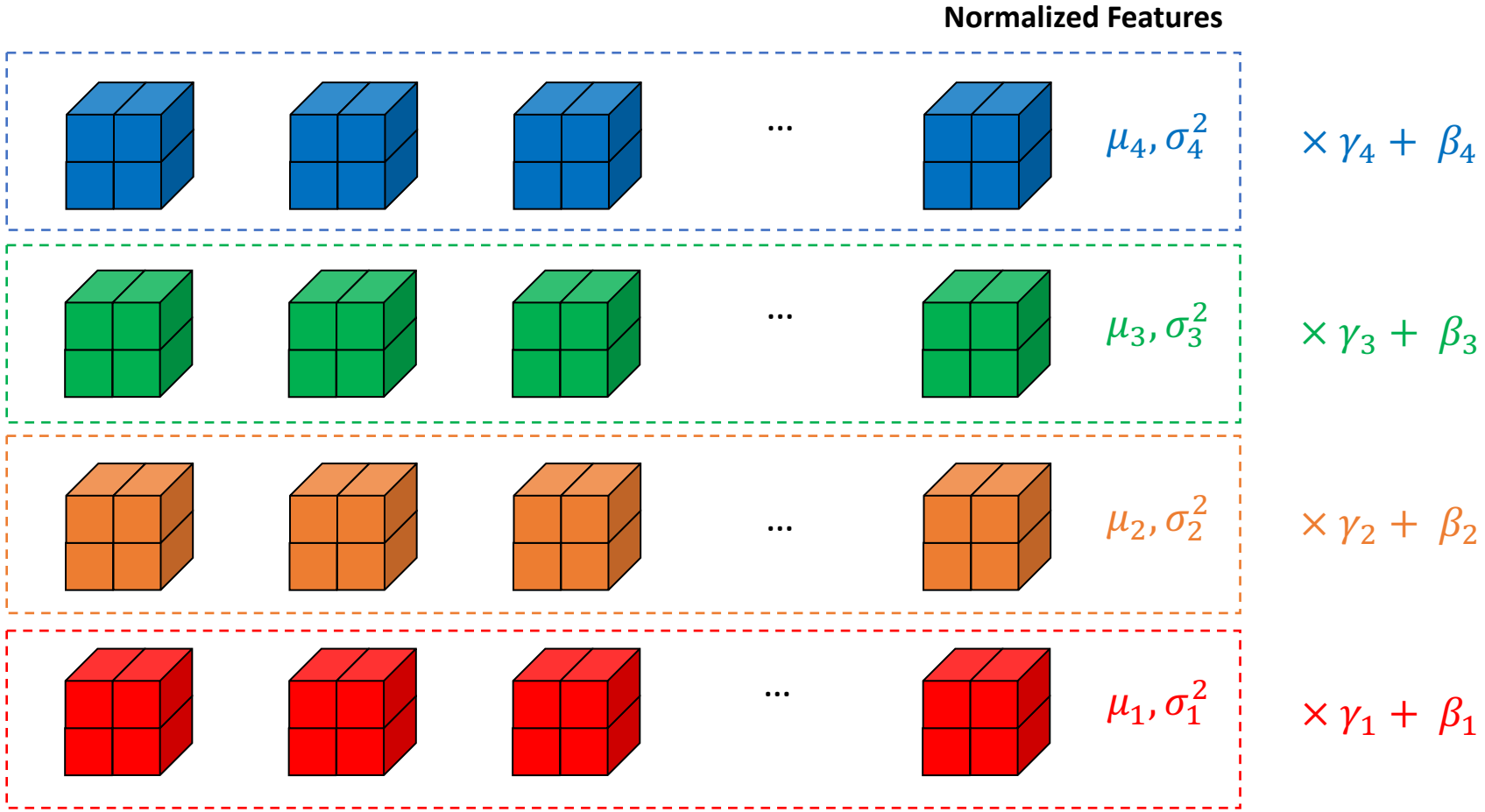
$$\frac{x_c - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

2. 통계량으로 Feature map을 Normalization

- Normalize 된 Feature는 Normal distribution의 형태가 될 것
- Feature의 70%정도가 분포하는 -1 ~ 1 구간은 해당 논문이 발표될 때 많이 사용했던 Sigmoid activation에서 거의 선형함수와 유사함
- **normalized feature를 activation function의 비선형 구간에 mapping하는 것이 추가로 필요함**

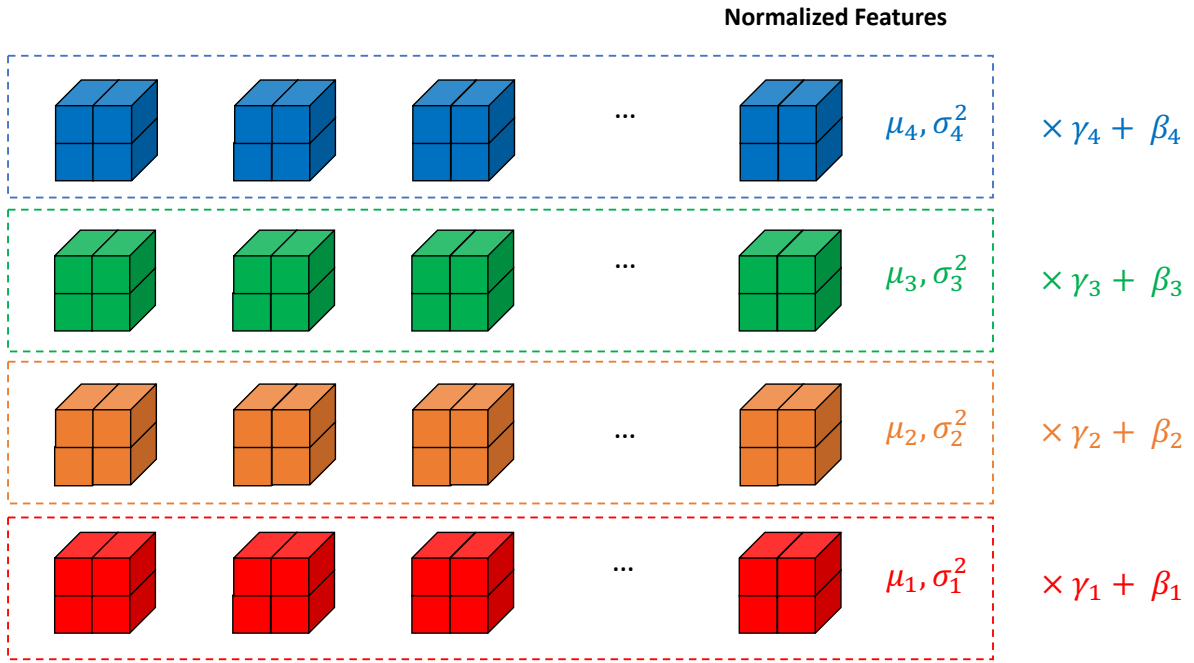


3. Learnable parameter(γ, β)로 다시 scaling 및 shifting



Normalization의 역 연산

3. Learnable parameter(γ, β)로 다시 scaling 및 shifting



γ 는 1, β 는 0으로 initialize하여 처음에는 기여도가 없다가
학습과정을 통해서 update가 이루어 지도록 한다.

$$\left(\frac{x_c - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}\right) \times \boldsymbol{\gamma_c} + \boldsymbol{\beta_c}$$

(x_c는 H*W 길이의 vector)

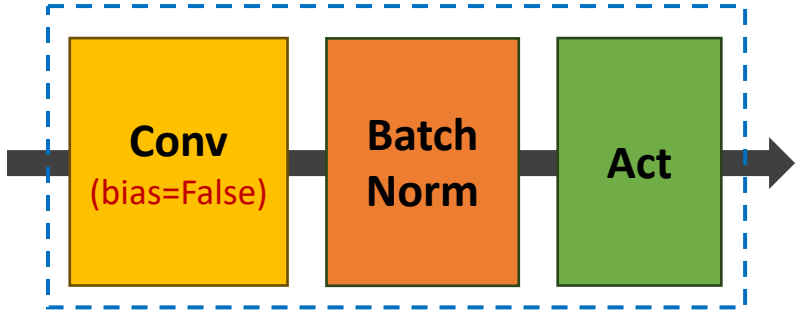
3. Learnable parameter(γ, β)로 다시 scaling 및 shifting

- 일반적으로 Conv -> BatchNorm -> Act 순서로 사용함
  (Convolution 연산을 $x \circledast K + b$ 라고 표현하면)

$$\left(\frac{x_c - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}\right) \times \boldsymbol{\delta_c} + \boldsymbol{\beta_c} \rightarrow \left(\frac{(x \circledast K + b)_c - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}\right) \times \boldsymbol{\delta_c} + \boldsymbol{\beta_c}$$

- Bias (b)를 더하는 것은 계산된 통계량의 평균(μ_c)을 빼는 것에 포함되고 이후 다시 학습된 $\boldsymbol{\beta_c}$ 로 보정하는 연산이 있기 때문에

- BatchNorm 이전 Convolution의 bias를 제거함

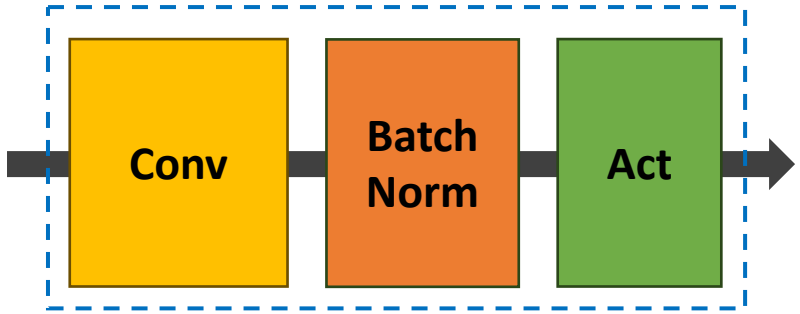


A convolution block with batch normalization

BatchNorm (Review)

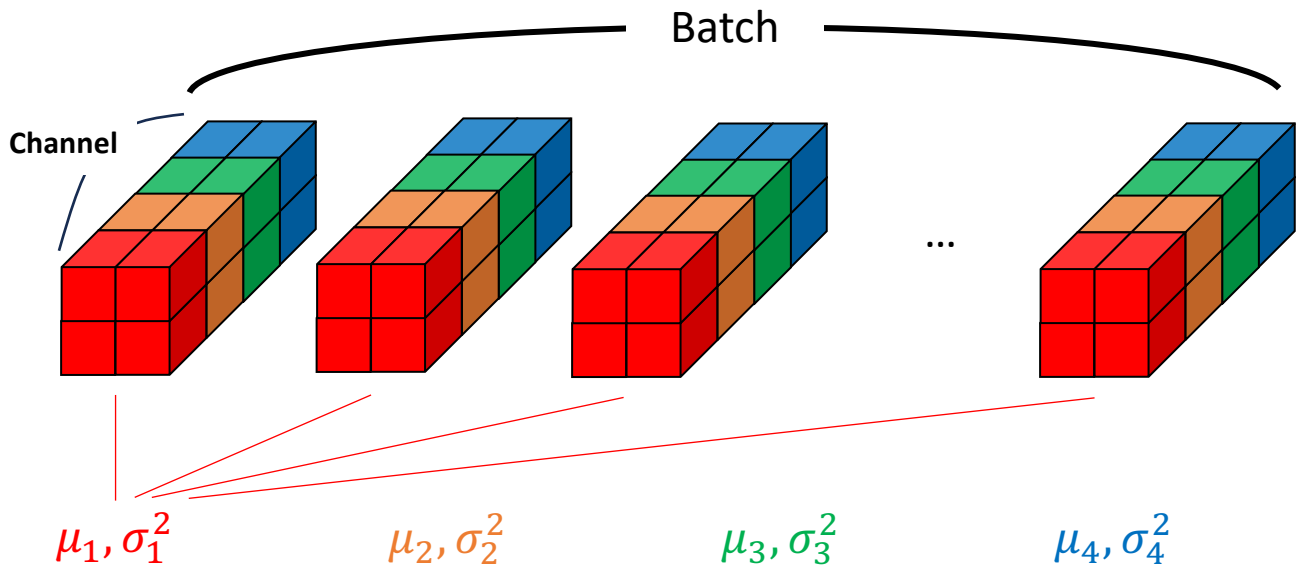
- **Procedure of BatchNorm training**

1. Feature map의 통계량 (평균, 분산) 계산
2. 1 에서 계산한 통계량으로 Feature map을 Normalization
3. Learnable parameter(γ, β)로 다시 scaling 및 shifting



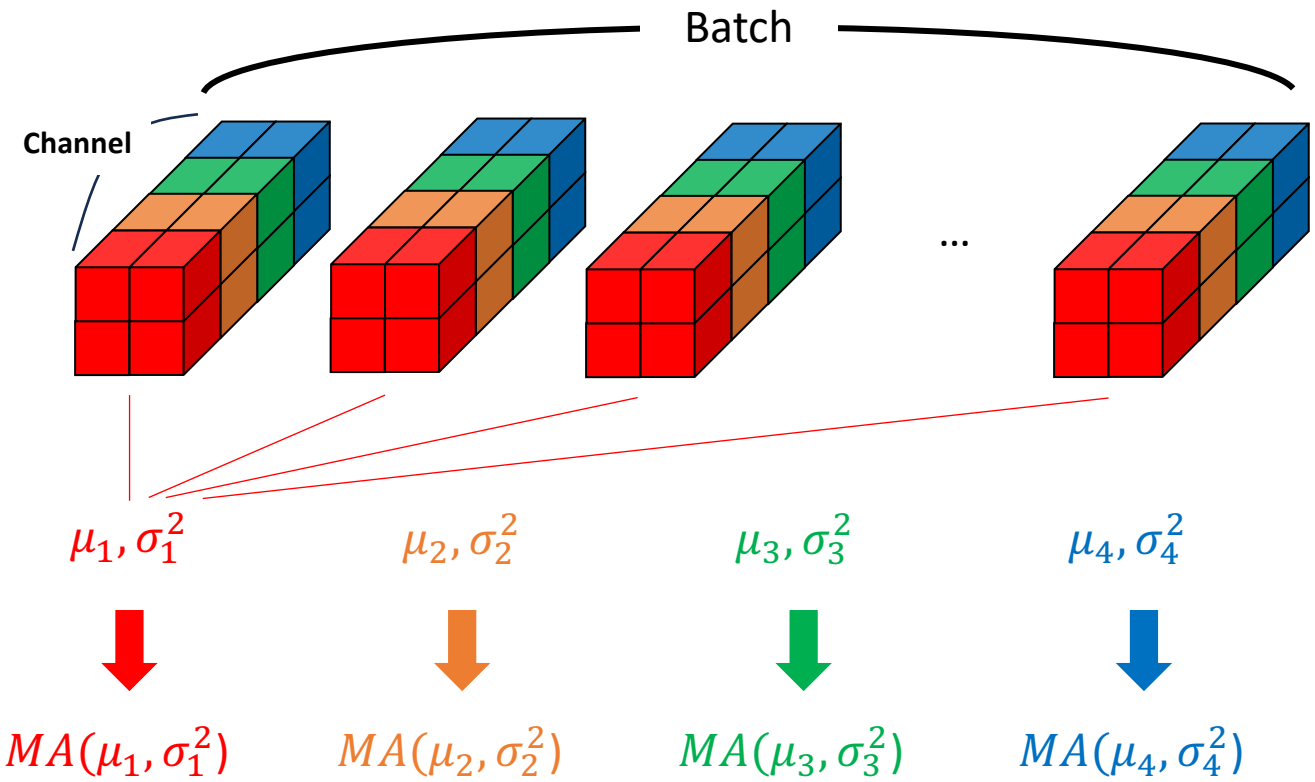
A convolution block with batch normalization

- **Procedure of BatchNorm at Inference time**



- 학습 때는 Batch size 가 고정되어 있으므로 학습하기에 통계량(평균, 분산)이 안정적
- 하지만 학습 후 Inference time (test) 에는 Batch size가 달라지기 때문에 **통계량의 차이가 발생함**

BatchNorm



- 학습할 때 계산한 통계량들의 이동 평균을 저장하여 Inference time (test) 에 사용

$$v_t = m \times v_{t-1} + (1 - m) \times x_t$$

v_t : t 시점의 이동평균 , m : 이동평균 파라미터

Implementation - BatchNorm

```python
class BatchNorm2d(nn.Module):
    def __init__(self, dim, eps=1e-5, momentum=0.1, affine=True):
        super().__init__()
        self.eps = eps
        self.momentum = momentum
        self.affine = affine
        if self.affine:
            self.gamma = nn.Parameter(torch.ones(1, dim, 1, 1))
            self.beta = nn.Parameter(torch.zeros(1, dim, 1, 1))
        self.register_buffer('running_mean', torch.zeros(1, dim, 1, 1))
        self.register_buffer('running_var', torch.ones(1, dim, 1, 1))
    def forward(self, x):
        # 1. Calculate Statistics(mu, var) channel-wise.
        if self.training:
            var, mu = torch.var_mean(x, dim=(0, 2, 3), unbiased=False, keepdim=True)
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * mu
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * var
        else: # if inference, use running statistics calculated in training.
            mu = self.running_mean
            var = self.running_var
        # 2. Normalize
        x = (x - mu) / torch.sqrt(var + self.eps)
        # 3. Scale and Shift
        if self.affine:
            x = x * self.gamma + self.beta
        return x
```

학습중이면 Feature의 통계량을 계산 후 이동평균을 Update

아니면 이동평균을 불러옴.

Implementation - BatchNorm

```python
class BatchNorm2d(nn.Module):
    def __init__(self, dim, eps=1e-5, momentum=0.1, affine=True):
        super().__init__()
        self.eps = eps
        self.momentum = momentum
        self.affine = affine
        if self.affine:
            self.gamma = nn.Parameter(torch.ones(1, dim, 1, 1))
            self.beta = nn.Parameter(torch.zeros(1, dim, 1, 1))
        self.register_buffer('running_mean', torch.zeros(1, dim, 1, 1))
        self.register_buffer('running_var', torch.ones(1, dim, 1, 1))
    def forward(self, x):
        # 1. Calculate Statistics(mu, var) channel-wise.
        if self.training:
            var, mu = torch.var_mean(x, dim=(0, 2, 3), unbiased=False, keepdim=True)
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * mu
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * var
        else: # if inference, use running statistics calculated in training.
            mu = self.running_mean
            var = self.running_var
        # 2. Normalize
        x = (x - mu) / torch.sqrt(var + self.eps)
        # 3. Scale and Shift
        if self.affine:
            x = x * self.gamma + self.beta
        return x
```

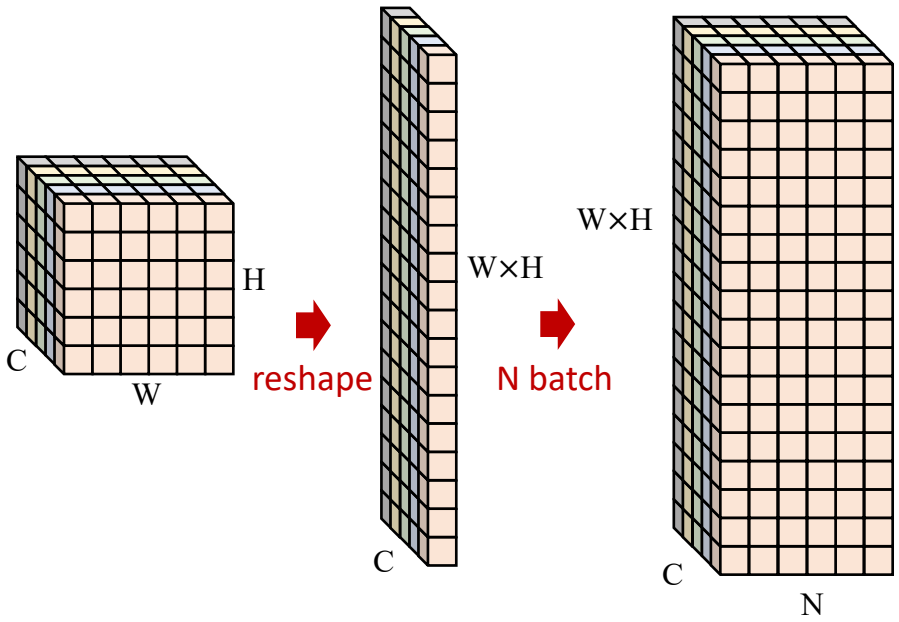
평균/ 분산으로 Feature map을 normalize

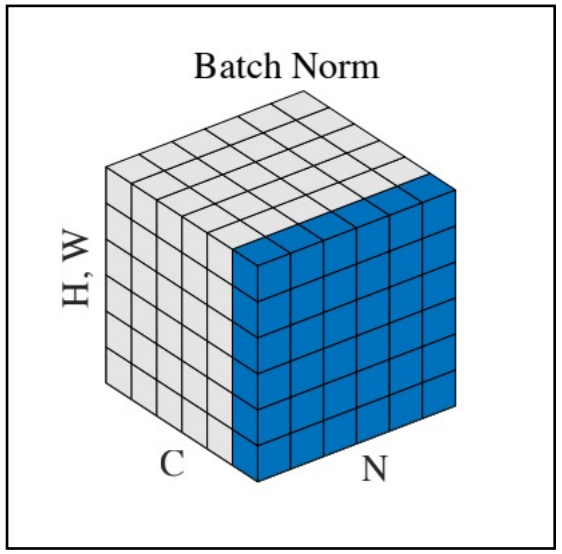
Implementation - BatchNorm

```python
class BatchNorm2d(nn.Module):
    def __init__(self, dim, eps=1e-5, momentum=0.1, affine=True):
        super().__init__()
        self.eps = eps
        self.momentum = momentum
        self.affine = affine
        if self.affine:
            self.gamma = nn.Parameter(torch.ones(1, dim, 1, 1))
            self.beta = nn.Parameter(torch.zeros(1, dim, 1, 1))
        self.register_buffer('running_mean', torch.zeros(1, dim, 1, 1))
        self.register_buffer('running_var', torch.ones(1, dim, 1, 1))
    def forward(self, x):
        # 1. Calculate Statistics(mu, var) channel-wise.
        if self.training:
            var, mu = torch.var_mean(x, dim=(0, 2, 3), unbiased=False, keepdim=True)
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * mu
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * var
        else: # if inference, use running statistics calculated in training.
            mu = self.running_mean
            var = self.running_var
        # 2. Normalize
        x = (x - mu) / torch.sqrt(var + self.eps)
        # 3. Scale and Shift
        if self.affine:
            x = x * self.gamma + self.beta
        return x
```

Learnable parameter로 다시 scaling 및 shifting

BatchNorm

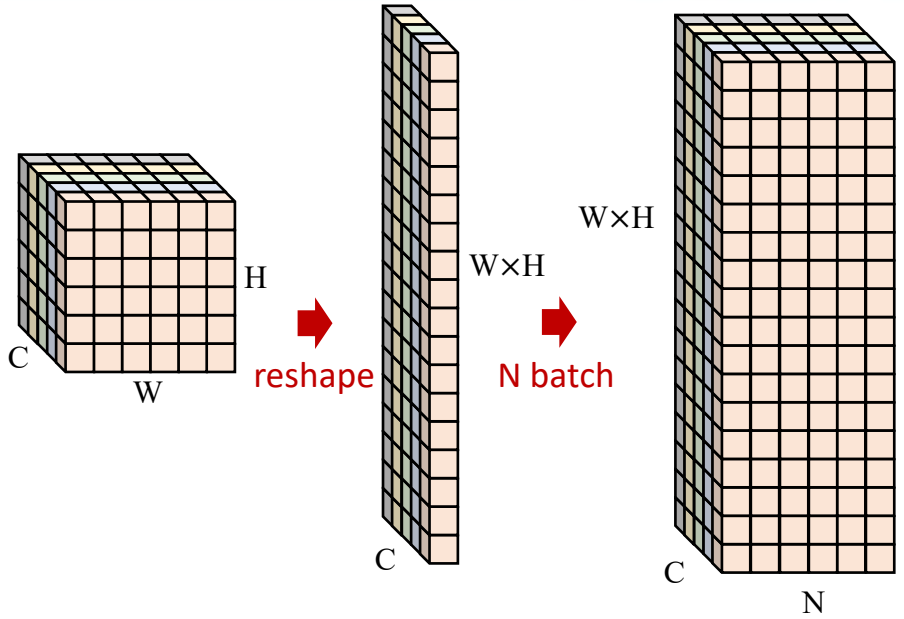


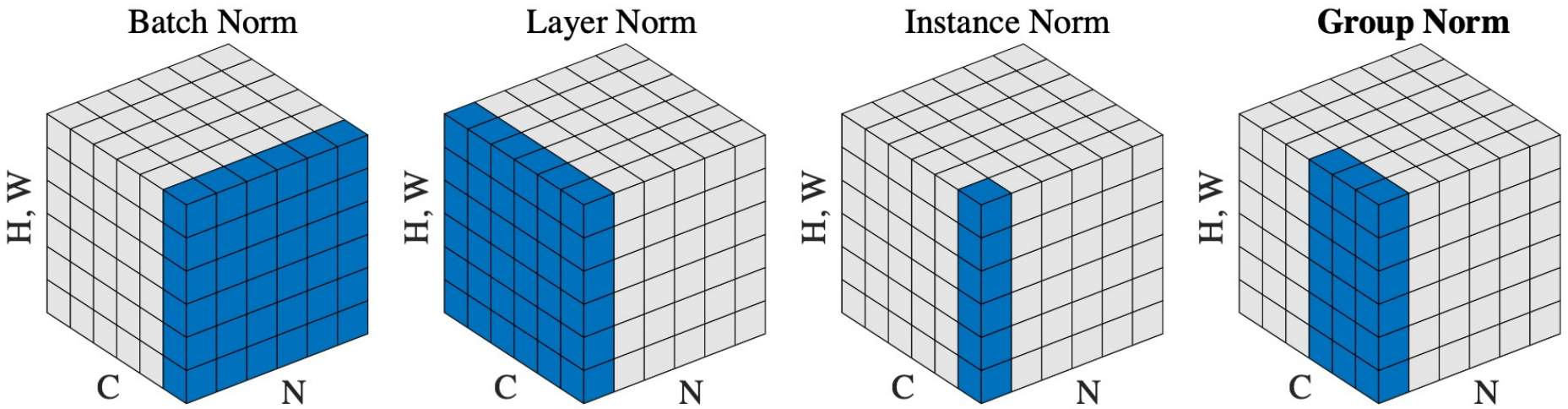


Batch Normalization 예시 그림

Group Normalization(Yuxin Wu, Kaiming He)

BatchNorm


H
C
W
reshape
W×H
C
W×H
N batch
C
N


Batch Norm
H, W
C
N
Layer Norm
H, W
C
N
Instance Norm
H, W
C
N
Group Norm
H, W
C
N

1. GPU사용하기

1-1. GPU란?

1-2. GPU 사용하기

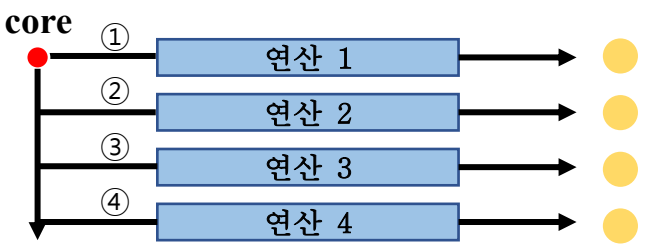
GPU란?

- GPU(Graphics Processing Unit):

그래프 처리 장치로 병렬 수치 연산을 고속으로 처리할 수 있음. 대부분의 딥러닝 프레임워크 대부분이 GPU를 활용해 대량의 연산을 고속으로 처리한다.

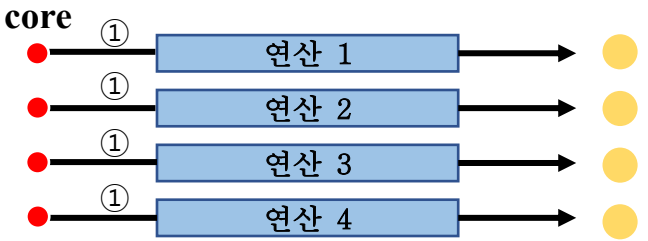
- CPU vs GPU
  - CPU: Core가 순차적으로 하나씩 연산을 함
  - GPU: 여러 개의 Core가 한번에 병렬로 연산을 함

<CPU>



→ 4의 시간이 필요

<GPU>



→ 1의 시간에 해결

GPU 활용

- GPU확인: torch.cuda.is_available( )
  - GPU 사용 가능 → True
  - GPU 사용 불가 → False

- 사용 가능한 GPU 개수 확인: torch.cuda.device_count( )

- 사용할 디바이스 종류 지정해주기:

device = torch.device('cuda:0') if torch.cuda.is_available( ) else torch.device('cpu')

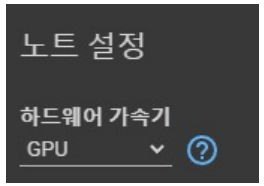
  - GPU 사용 가능 → device에 GPU를 지정
  - GPU 사용 불가 → deviec에 CPU를 지정

GPU 활용

- Model과 Tensor에 GPU 할당하기

  model = Net( ).cuda( ) 혹은 model = Net( ).to(device)

- 구글 Colab사용시 상단 베너의 『수정』-『노트설정』에서 하드웨어 가속기를 GPU로 설정하기



GPU 활용시 주의할 점

- GPU에 할당된 tensor와 CPU에 할당된 tensor와의 연산

```
import torch

tensor1 = torch.tensor([1,2,3,4,5]).cuda()
tensor2 = torch.tensor([1,2,3,4,5])
print(tensor1 + tensor2)
```

```
Traceback (most recent call last):
  File "/home/osh/test.py", line 5, in <module>
    print(tensor1 + tensor2)
RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
```

GPU 활용시 주의할 점

- GPU에 할당된 tensor를 matplot.pyplot을 이용하여 그래프 그리기

```
tensor1 = torch.tensor([1,2,3,4,5]).cuda()
plt.plot(tensor1)   # 현재 GPU에 할당되어 있는 상태
plt.show()
```

```
TypeError: can't convert cuda:0 device type tensor to numpy.
Use Tensor.cpu() to copy the tensor to host memory first.
```

data.cpu( )를 사용하여 tensor를 cpu에 할당한 후 그래프 그리기

```
tensor1 = torch.tensor([1,2,3,4,5]).cuda()
plt.plot(tensor1.cpu())
plt.show()
```

Cifar100 Classification

1. Cifar100 Dataset
2. torch module 함수 이용하기
3. train함수

Cifar100 Dataset

- 100개의 Class로 구성된 데이터셋 (20개의 super class)
- Dataset Size: train data: 50,000
  test data : 10,000
- Image Size: 32x32x3

Input transforms

- torchvision.transforms.Compose( ) : 여러 가지의 이미지 변환 기능(transform)들을 Compose로 구성할 수 있도록 해줌 (Cifar10 코드에서 input_preprocess 역할)

Cifar10 Code

```
def input_preprocess(x):
    x = x.float()
    x = x/255.0     # input data를 0~1사이의 float로 바꿔주기
    half = int(x.shape[0]/2)
    x[0:half, :] = torch.flip(x[0:half, :], dims=[2])
    return x
```

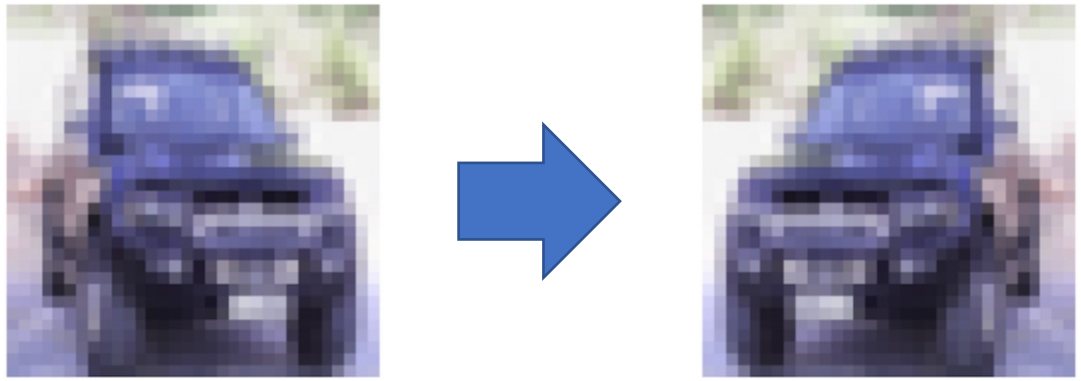
torchvision.transforms.Compose( )

Code

```
resize = (32, 32)
mean = (0.5, 0.5, 0.5)
std = (0.5, 0.5, 0.5)
transform_list_train = transforms.Compose([
    transforms.Resize(resize, interpolation=InterpolationMode.BICUBIC),  # Image의 크기 바꾸기
    transforms.RandomHorizontalFlip(p=0.5),  # Augmentation
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
transform_list_test = transforms.Compose([
    transforms.Resize(resize, interpolation=InterpolationMode.BICUBIC),  # Image의 크기 바꾸기
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

Input transforms

- torchvision.transforms.Compose( ) : 여러 가지의 이미지 변환 기능(transform)들을 Compose 로 구성할 수 있도록 해줌 (Cifar10 코드에서 input_preprocess 역할)
  - transforms.resize(new_size, interpolation=InterpolationMode.Bicubic)
    - ➢ 주어진 image의 크기를 new_size로 변경함
    - ➢ interpolation: 이미지 크기 변경 시 사용되는 알고리즘을 지정
  - transforms.RandomHorizontalFlip( )
    - ➢ 랜덤하게 이미지들을 HorizontalFlip을 실행함



    - ➢ Model을 test 할 때는 사용하지 않음

Input transforms

- transforms.ToTensor( )
  - ➢ 주어진 image를 pytorch가 받아 드릴 수 있는 tensor로 바꿔줌
  - ➢ 모든 이미지의 픽셀값이 0~1의 범위를 가지도록 변환함

- transforms.Normalize((mean1,mean2,mean3), (std1, std2, std3))
  - ➢ 주어진 값들을 이용하여 image를 normalize 해줌
    (데이터의 통계를 이용해서 mean, std 값은 구할 수도 있음)

Input transforms

Code

```
resize = (32, 32)
mean = (0.5, 0.5, 0.5)
std = (0.5, 0.5, 0.5)
transform_list_train = transforms.Compose([
    transforms.Resize(resize, interpolation=InterpolationMode.BICUBIC),  # Image의 크기 바꾸기
    transforms.RandomHorizontalFlip(p=0.5),  # Augmentation
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
transform_list_test = transforms.Compose([
    transforms.Resize(resize, interpolation=InterpolationMode.BICUBIC),  # Image의 크기 바꾸기
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

- 이미지 크기를 (32,32)로 resize해줌
- transforms.Normalize를 이용하여 픽셀값의 범위를 -1~1로 변환해줌
- Train데이터에 대해서 랜덤하게 HorizontalFlip을 수행함

Cifar100 load

- torchvision.datasets.CIFAR100( ) : cifar100 데이터 다운로드 및 가져오기

Cifar10 Code

```
def load_dataset(dataset, root_path):
    train_dataset = None
    test_dataset = None

    if dataset == 'CIFAR10':
        train_dataset = datasets.CIFAR10(root_path,           # train data
                                         download=True,
                                         train=True,
                                         )
        train_dataset.data = torch.tensor(train_dataset.data)
        train_dataset.targets = torch.tensor(train_dataset.targets)
        test_dataset = datasets.CIFAR10(root_path,          # test data
                                        download=False,
                                        train=False,)
        test_dataset.data = torch.tensor(test_dataset.data)
        test_dataset.targets = torch.tensor(test_dataset.targets)
    else:
        print("Incorrect dataset!")
    return train_dataset, test_dataset
```

⬇ torchvision.datasets.CIFAR100( )

Code

```
cifar100_training = torchvision.datasets.CIFAR100(root='./cifar100_data/'
                                , train=True, download=True, transform=transform_train_list)
cifar100_testing = torchvision.datasets.CIFAR100(root='./cifar100_data/'
                                , train=False, download=True, transform=transform_test_list)
```

Cifar100 load

- torchvision.datasets.CIFAR100( ) : cifar100 데이터 다운로드 및 가져오기

Code

```
cifar100_training = torchvision.datasets.CIFAR100(root='./cifar100_data/'
                                , train=True, download=True, transform=transform_train_list)
cifar100_testing = torchvision.datasets.CIFAR100(root='./cifar100_data/'
                                , train=False, download=True, transform=transform_test_list)
```

- Cifar100 데이터를 root에 다운로드를 받음
- 앞에서 설정한 Compose대로 이미지를 변환함

Cifar100 load

- torch.utils.data.DataLoader(): 데이터를 랜덤하게 섞어주고 학습 시 mini batch를 가져옴

Cifar10 Code

```
class RandomSampler:
    def __init__(self, full_size, batch_size):
        self.full_size = full_size   # 데이터셋 전체 사이즈 50000
        self.batch_size = batch_size # mini-batch 사이즈 64
        self.indices = np.arange(0, self.full_size)  # create index
        self.shuffling()             # 클래스 안에 shuffling() 함수(메소드)를 call
        self.start = 0               # 현재 mini-batch의 시작 index
        self.end = self.batch_size   # 현재 mini-batch의 끝 index

    def shuffling(self):             # 1~50,000까지의 index를 섞어주는 함수
        np.random.shuffle(self.indices)

    def get_random_idx(self):        # 랜덤한 index를 return 해주는 함수
        if self.end > len(self.indices): # 현재 mini-batch의 끝 index가 전체 train_data의 size보다 클 경우
            print("1 epoch trained")
            self.shuffling()         # 데이터 다시 섞어주기
            self.start = 0           # 현재 mini-batch의 index 초기화
            self.end = self.batch_size

        idx = self.indices[self.start:self.end] # index 뽑아주기
        self.start += self.batch_size # 미리 다음 mini-batch로 index를 옮김
        self.end += self.batch_size
        return idx
```

⬇ torch.utils.data.DataLoader()

Code

```
dataloaders = {}
dataloaders['train'] = torch.utils.data.DataLoader(train_dataset, shuffle=True,
                                                   drop_last=False, batch_size=batch_size['train'])
dataloaders['test'] = torch.utils.data.DataLoader(test_dataset, shuffle=False,
                                                  drop_last=False, batch_size=batch_size['test'])
dataset_sizes = {'train': len(train_dataset), 'test': len(test_dataset)}
```

Cifar100 load

- torch.utils.data.DataLoader(): 데이터를 랜덤하게 섞어주고 학습 시 mini batch를 가져옴

Code

```
dataloaders = {}
dataloaders['train'] = torch.utils.data.DataLoader(train_dataset, shuffle=True,
                                                   drop_last=False, batch_size=batch_size['train'])
dataloaders['test'] = torch.utils.data.DataLoader(test_dataset, shuffle=False,
                                                  drop_last=False, batch_size=batch_size['test'])
dataset_sizes = {'train': len(train_dataset), 'test': len(test_dataset)}
```

- shuffle: 데이터를 랜덤하게 섞을지 선택
- drop_last: 마지막 batch를 사용할지 사용 안 할지 선택
- batch_size: DataLoader가 반환해주는 데이터 배치의 크기

Pytorch CrossEntropyLoss

- torch.nn.CrossEntropyLoss(): 데이터의 label을 따로 onehot encoding할 필요없이 CrossEntropy Loss를 계산해줌
- Softmax와 Loss 계산이 한번에 이뤄짐

Cifar10 Code

```
def softmax(x):
    x_exp = torch.exp(x)
    x_exp_sum = torch.sum(x_exp, dim=1)     # softmax 분모
    out = x_exp/x_exp_sum.unsqueeze(dim=1)
    return out
```

+

```
def cross_entropy(x, y):
    loss = -torch.log(x)*y
    loss = loss.sum()/x.shape[0]
    return loss
```

↓ torch.nn.CrossEntropyLoss()

Code

```
criterion = nn.CrossEntropyLoss()
```

```
loss = criterion(model_output, label)   # Train 함수 내부
```

Train 함수

- 한 epoch 학습이 끝날 때마다 test를 수행하여 Model의 test Accuracy를 계산

- Train/Test phase를 나누어

  Train phase에서는 train data에 대해서 Model이 업데이트가 되고,

  Test phase에서는 test data에 대해서 Loss와 Accuracy를 구함

Train 함수

Code(전체)

```python
def train(model, dataloaders, train_epoch, loss_fn, optimizer, scheduler, dataset_sizes):
    program_start = time.time()

    # 학습 결과 확인을 위한 리스트 선언
    loss_list = {'train': [], 'test': []}
    acc_list = {'train': [], 'test': []}

    for epoch in range(train_epoch):
        print(f"\n\t{epoch} - epoch train")
        for phase in ['train', 'test']:
            if phase == 'train':
                model.train(True)  # Set model to training mode
            else:
                model.train(False)  # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            for data in dataloaders[phase]:
                inputs, label = data
                inputs = inputs.to(device)  # device 에 할당 시키기
                label = label.to(device)
                model_output = model(inputs)  # forward propagation
                _, preds = torch.max(model_output.data, 1)
                loss = loss_fn(model_output, label)

                if phase == 'train':    # train data에 대해서는 back-propagation
                    optimizer.zero_grad()
                    loss.backward()  # back propagation
                    optimizer.step()

                # statistics
                running_loss += loss.data   # loss
                running_corrects += torch.sum(preds == label.data)  # acc

            running_corrects = running_corrects.float()
            epoch_loss = running_loss / len(dataloaders[phase])
            epoch_acc = running_corrects / dataset_sizes[phase]

            print(f'{phase} Loss: {epoch_loss:.6f} Acc: {100 * epoch_acc:.4f} %')
            loss_list[phase].append(epoch_loss)     # 현재까지의 loss 저장
            acc_list[phase].append(100*epoch_acc)   # 현재까지의 acc 저장

        scheduler.step()

    time_elapsed = time.time() - program_start
    print(f'{(time.time() - program_start) / 60:.5f}mins')
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))

    return loss_list, acc_list
```

Train 함수

Code(1/2)

```
def train(model, dataloaders, train_epoch, loss_fn, optimizer, scheduler, dataset_sizes):
    program_start = time.time()

    # 학습 결과 확인을 위한 리스트 선언
    loss_list = {'train': [], 'test': []}
    acc_list = {'train': [], 'test': []}

    for epoch in range(train_epoch):
        print(f"\n\t{epoch} - epoch train")
        for phase in ['train', 'test']:
            if phase == 'train':
                model.train(True)  # Set model to training mode
            else:
                model.train(False)  # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            for data in dataloaders[phase]:
                inputs, label = data
                inputs = inputs.to(device)  # device 에 할당 시키기
                label = label.to(device)
                model_output = model(inputs)  # forward propagation
                _, preds = torch.max(model_output.data, 1)
                loss = loss_fn(model_output, label)

                if phase == 'train':    # train data에 대해서는 back-propagation
                    optimizer.zero_grad()
                    loss.backward()  # back propagation
                    optimizer.step()

                # statistics
                running_loss += loss.data   # loss
                running_corrects += torch.sum(preds == label.data)  # acc
```

Train 함수

Code(2/2)

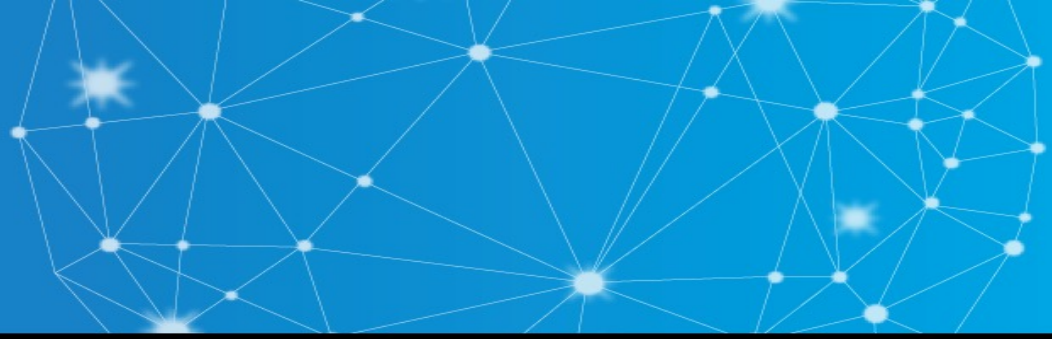
```
            running_corrects = running_corrects.float()
            epoch_loss = running_loss / len(dataloaders[phase])
            epoch_acc = running_corrects / dataset_sizes[phase]

            print(f'{phase} Loss: {epoch_loss:.6f} Acc: {100 * epoch_acc:.4f} %')
            loss_list[phase].append(epoch_loss)     # 현재까지의 loss 저장
            acc_list[phase].append(100*epoch_acc)   # 현재까지의 acc 저장

        scheduler.step()

    time_elapsed = time.time() - program_start
    print(f'{(time.time() - program_start) / 60:.5f}mins')
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))

    return loss_list, acc_list
```



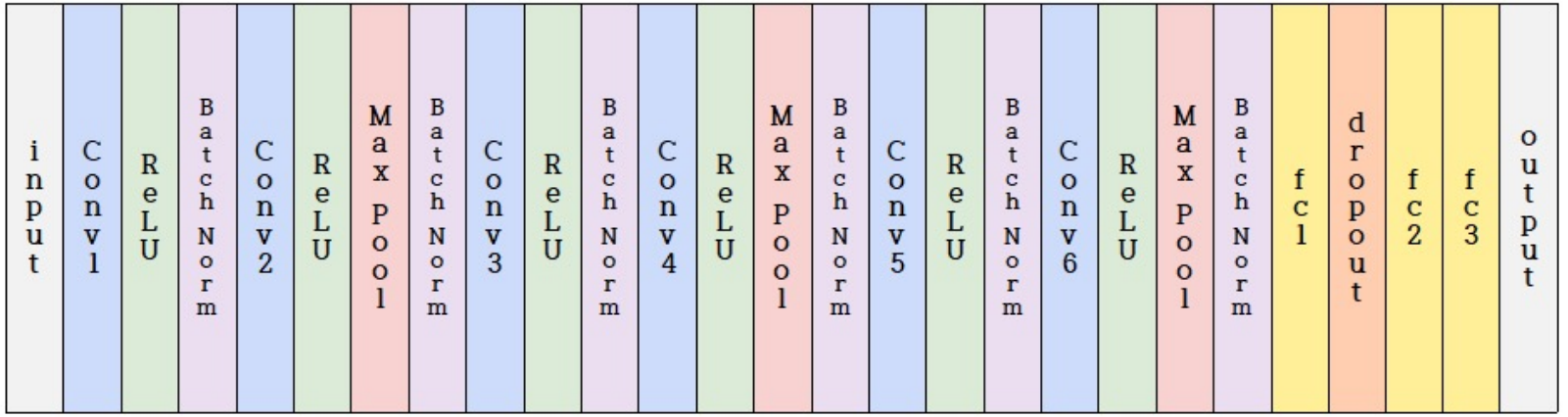
3. CNN Model

3-1. layer 6구조의 CNN

3-2. layer 10구조의 CNN

Conv Layer 6개 구조 CNN

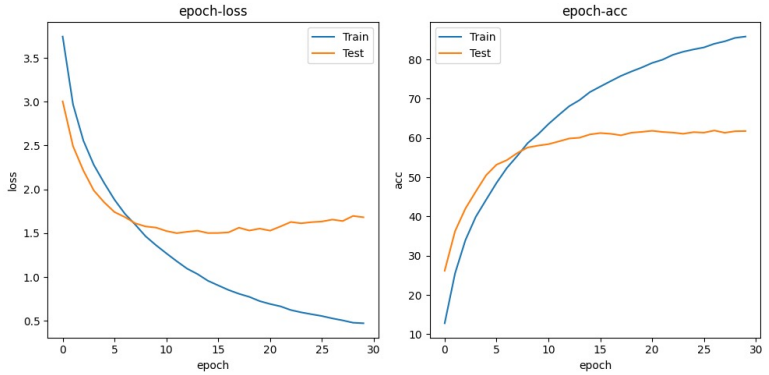
- Model Architecture



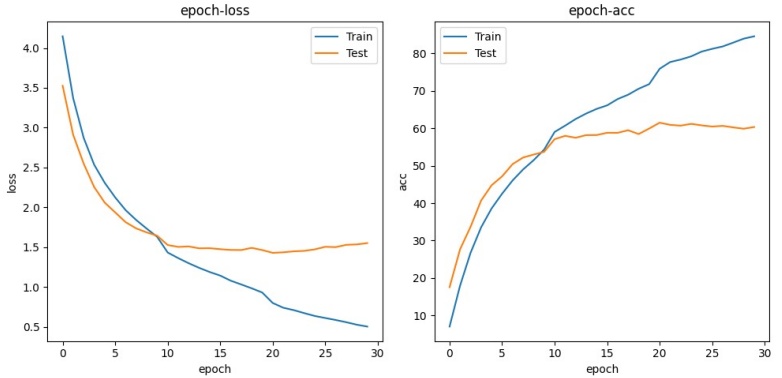
- Train Hyper-Parameters
  - Train-Batch Size: 200
  - Train epoch: 30
- Optimizer & Schdeuler
  - Option1 : Adam(lr= 0.001)
  - Option2 : SGD(lr= 0.005, momentum=0.9, weight_decay=0.0005) + StepLR
  - Option3 : SGD(lr= 0.005, momentum=0.9, weight_decay=0.0005) + CosineAnnealingLR

Layer 6개 구조 CNN

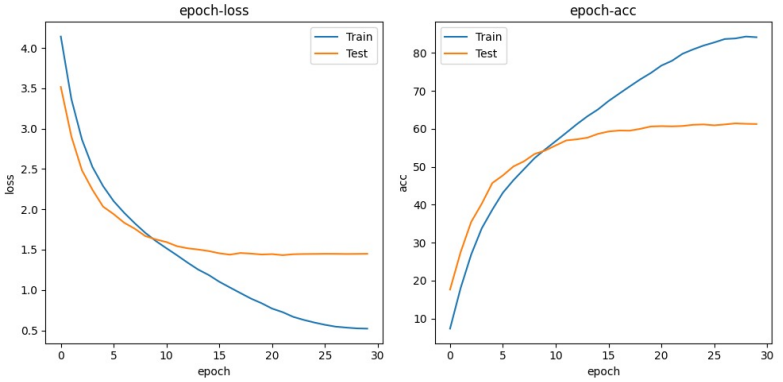
- #1 Adam
- Max Accuracy: 61.89% (27epoch)



- #2 SGD + StepLR (step_size=10, gamma= 0.5)
- Max Accuracy: 61.73% (21epoch)

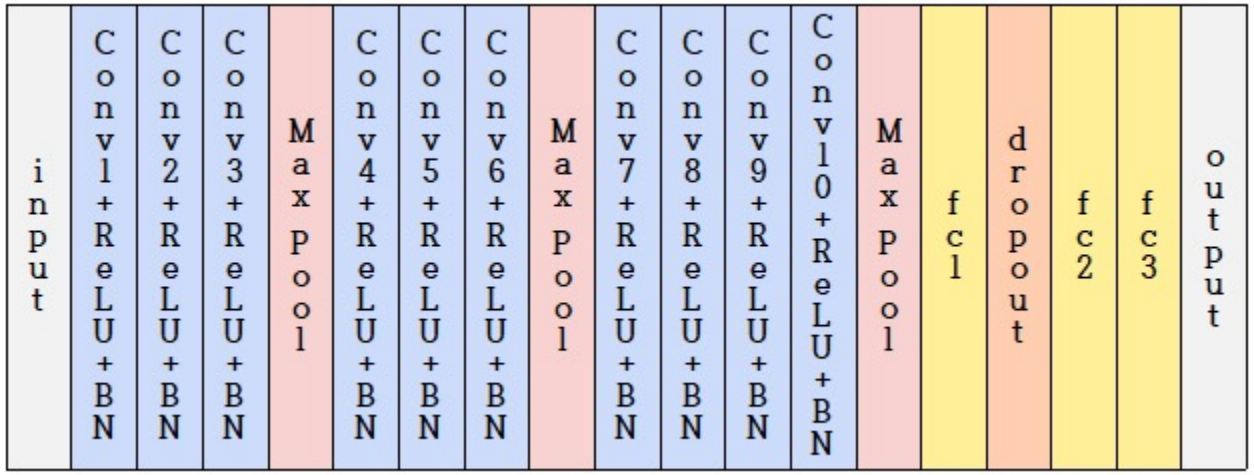


- #3 SGD + CosineAnnealingLR (T_max= epoch)
- Max Accuracy: 61.67% (28epoch)



Conv Layer 10개 구조 CNN

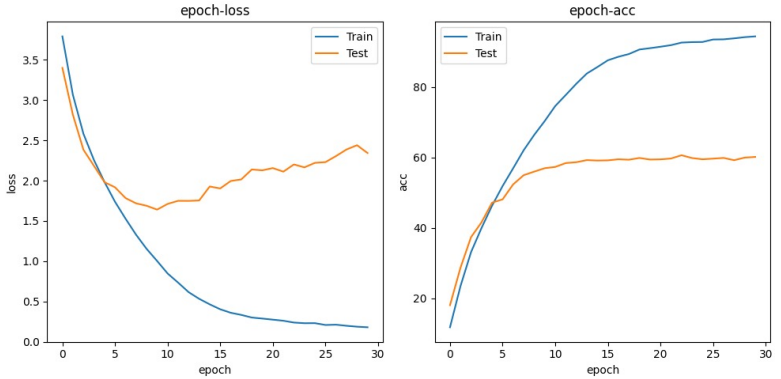
- Model Architecture



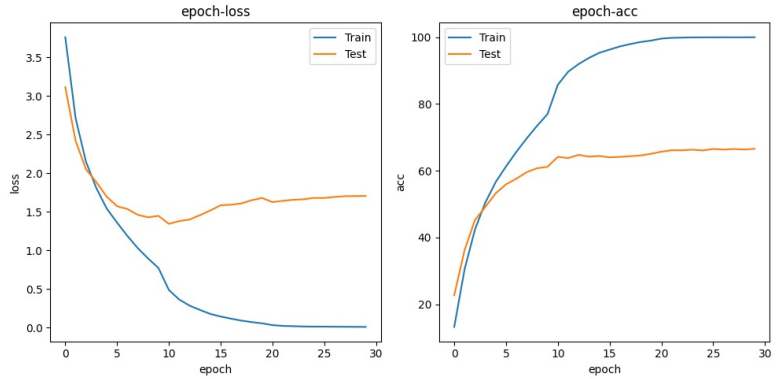
- Train Hyper-Parameters
  - Train-Batch Size: 200
  - Train epoch: 30
- Optimizer & Schdeuler
  - Option1 : Adam(lr= 0.001)
  - Option2 : SGD(lr= 0.005, momentum=0.9, weight_decay=0.0005) + StepLR
  - Option3 : SGD(lr= 0.005, momentum=0.9, weight_decay=0.0005) + CosineAnnealingLR

Layer 10개 구조 CNN

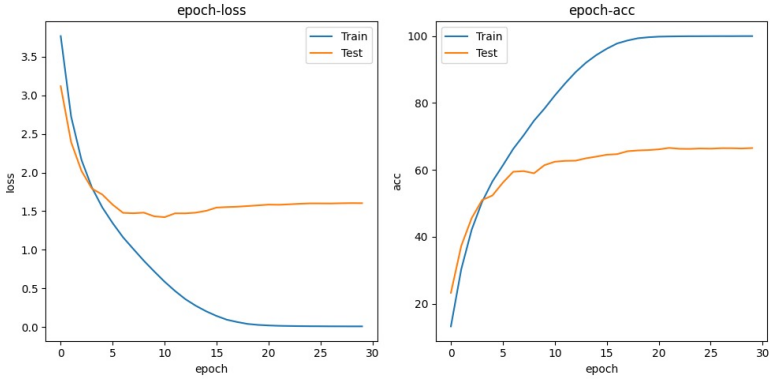
- #1 Adam
- Max Accuracy: 60.62% (23epoch)



- #2 SGD + StepLR (step_size=10 , gamma= 0.5)
- Max Accuracy: 66.58% (30epoch)

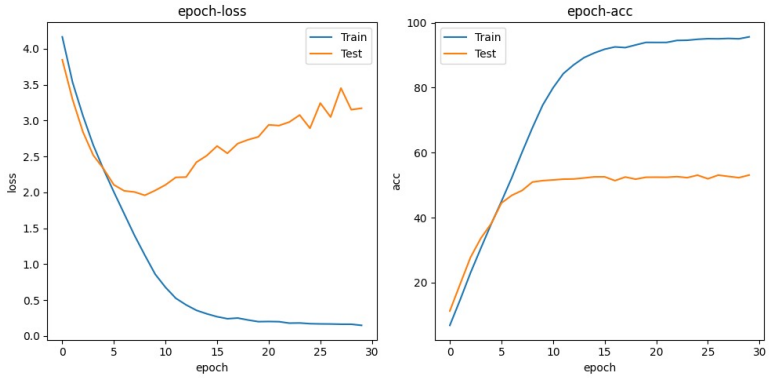


- #3 SGD + CosineAnnealingLR (T_max= epoch)
- Max Accuracy: 66.56% (22epoch)

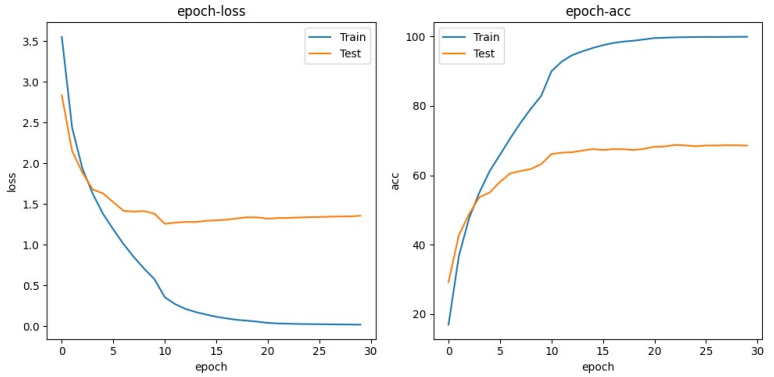


Layer 10개 구조 CNN – size 64

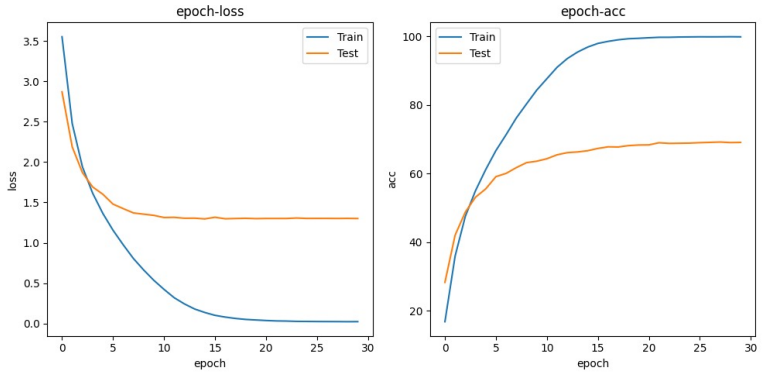
- #1 Adam
- Max Accuracy: 53.10% (30epoch)



- #2 SGD + StepLR (step_size=10 , gamma= 0.5)
- Max Accuracy: 68.74% (23epoch)



- #3 SGD + CosineAnnealingLR (T_max= epoch)
- Max Accuracy: 69.23% (28epoch)



교육부 Ministry of Education

NRF 한국연구재단 National Research Foundation of Korea

thank you

본 과제(결과물)는 교육부와 한국연구재단의 재원으로 지원을 받아 수행된
디지털신기술인재양성 혁신공유대학사업의 연구결과입니다.

Convergence and Open Sharing System COSS 인공지능 전남대학교 CHONNAM NATIONAL UNIVERSITY 성균관대학교 SUNGKYUNKWAN UNIVERSITY knu 경북대학교 KYUNGPOOK NATIONAL UNIVERSITY 전주대학교 JEONJU UNIVERSITY 서울시립대학교 UNIVERSITY OF SEOUL 서울과학기술대학교 SEOUL NATIONAL UNIVERSITY OF SCIENCE & TECHNOLOGY 영진전문대학교 YEUNGJIN UNIVERSITY