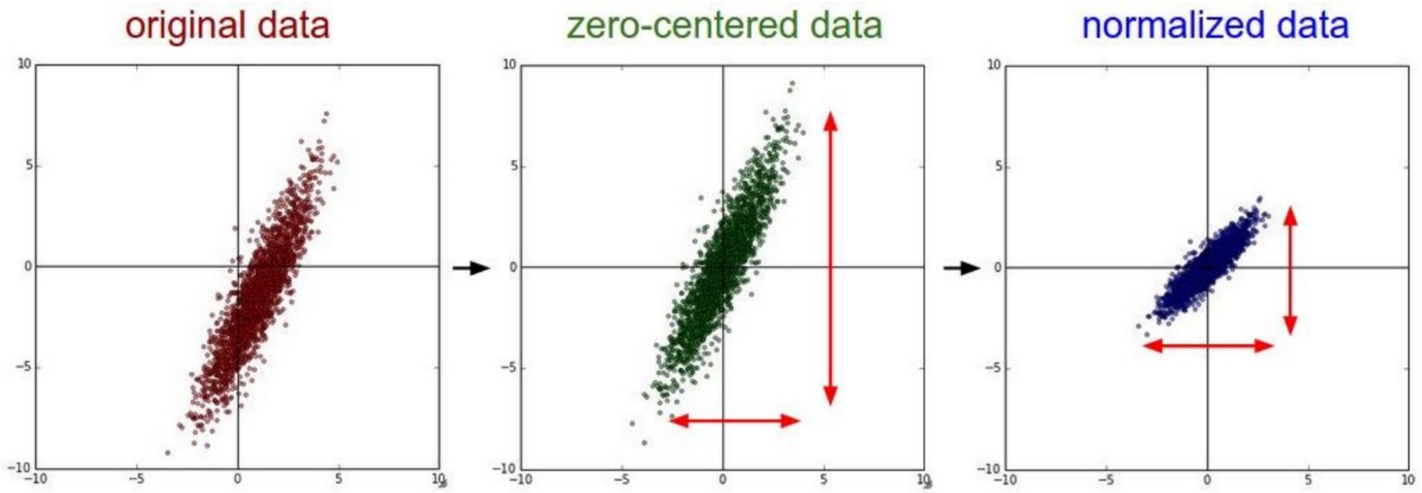
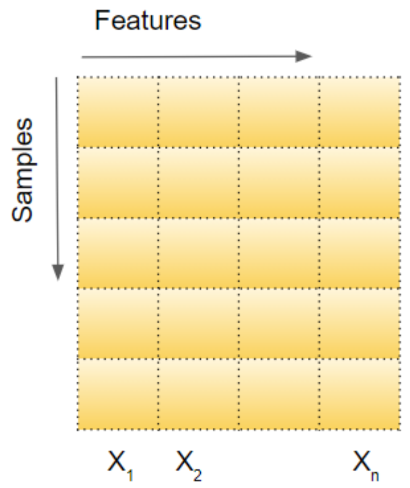
Normalization

Normalization

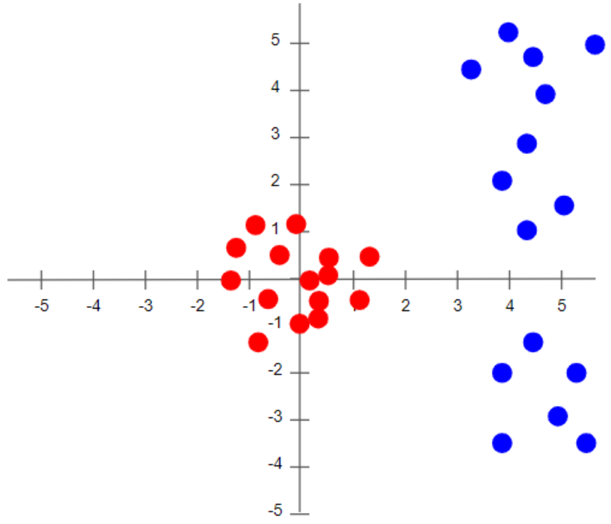
- ◈Data Normalization
  - – 데이터 정규화



- ◈How to normalize data?
  - – Normalization per feature!



$$X_i = \frac{X_i - Mean_i}{StdDev_i}$$



The original values (in blue) are now centered around zero and "std. dev." is one (in red).

[Source] https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Normalization

◈Why is it useful?

– Before normalization

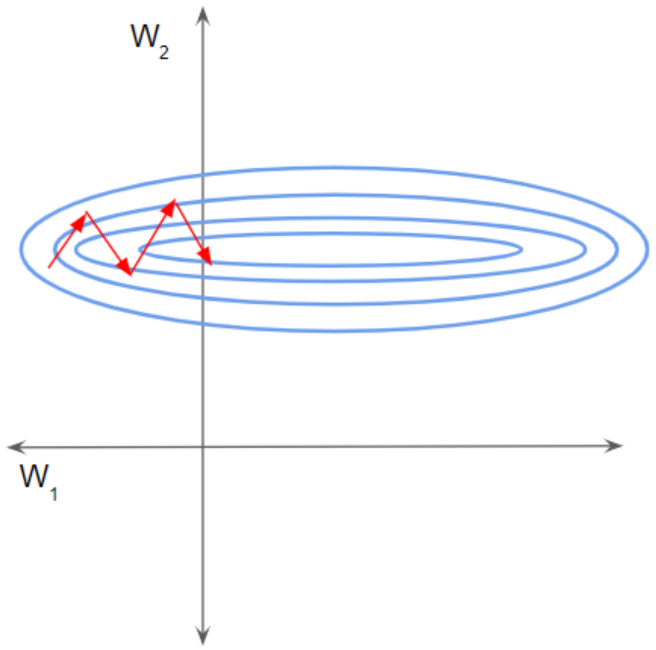
- A larger update to one weight due to its large gradient
  ➔ the gradient descent may bounce to the other side of the slope

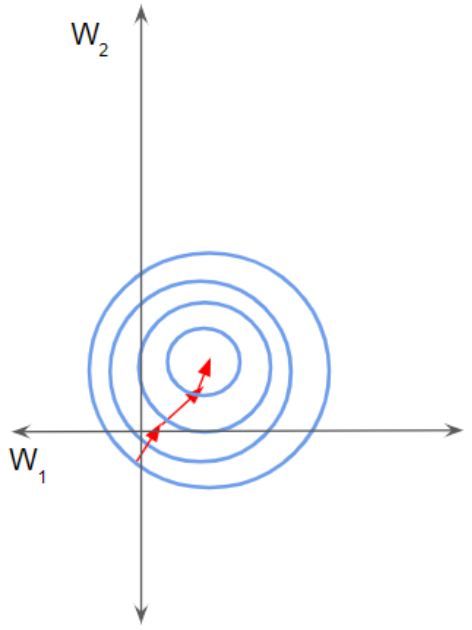
- A smaller update to one weight due to its small gradient
  ➔ making smaller weight updates over all weights

- This uneven trajectory takes longer for the network to converge

– After normalization

- Gradient descent can then proceed smoothly down to the minimum
- That is, it helps the faster convergence of weights and bias





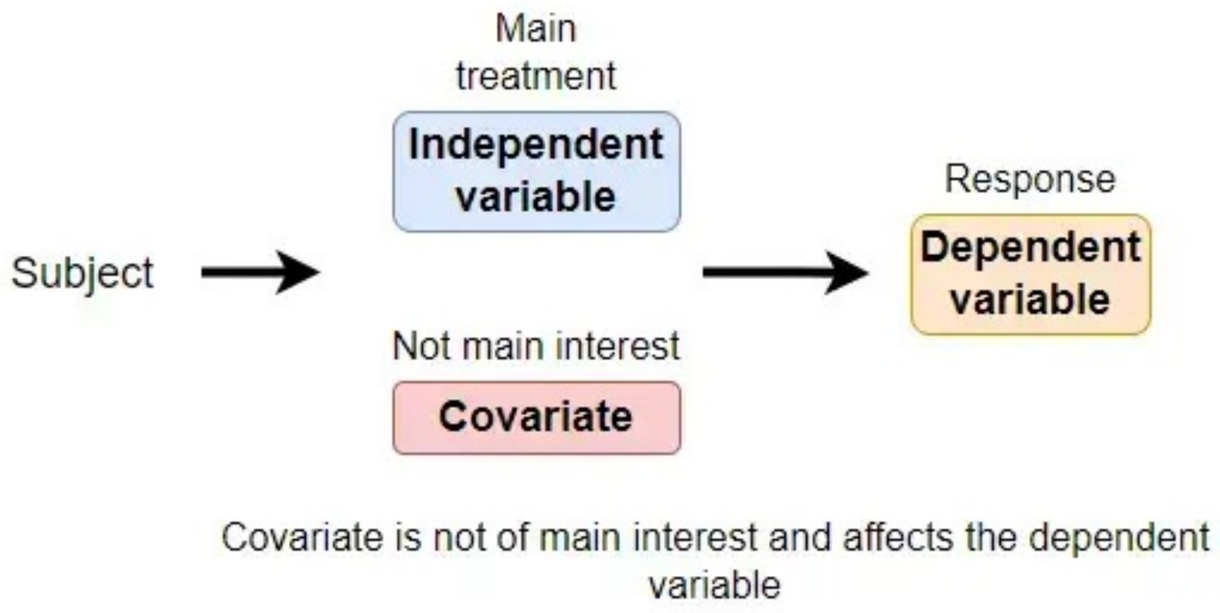
[Source] https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Internal Covariate Shift

◈Covariate (공변량)

– a variable that is related to both the independent and dependent variables

– It is usually not controlled during data collection



Covariate is not of main interest and affects the dependent variable

예를 들어, 한 식물 연구자는 식물의 수확량(Yield of the plant)이 식물의 유전자형(Genotype of the plant)에 따라 달라지는지 여부를 검정하려고 한다. 연구원은 다양한 식물 유전자형이 생산량에 어떠한 영향을 미치는 지를 연구하기 위하여 데이터를 수집한다.

그러나, 연구자는 식물의 키(Height of the plant)도 식물 수확량에 영향을 미친다는 것을 알고 있다. 식물의 키는 연구자의 주요 관심사는 아니지만, 정확한 결과를 얻으려면 통계 모델에서 이를 고려해야 한다.

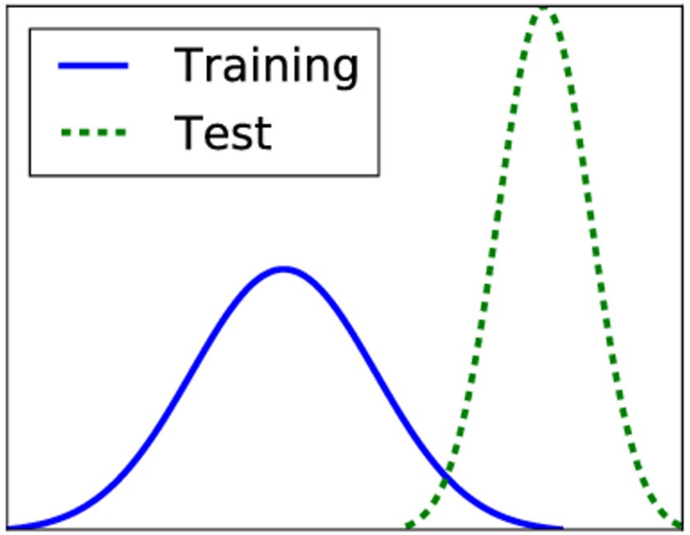
이 예에서 식물 유전자형은 독립 변수이고 식물 수확량은 종속 변수(반응 변수)이며 식물 키는 Covariate이다.

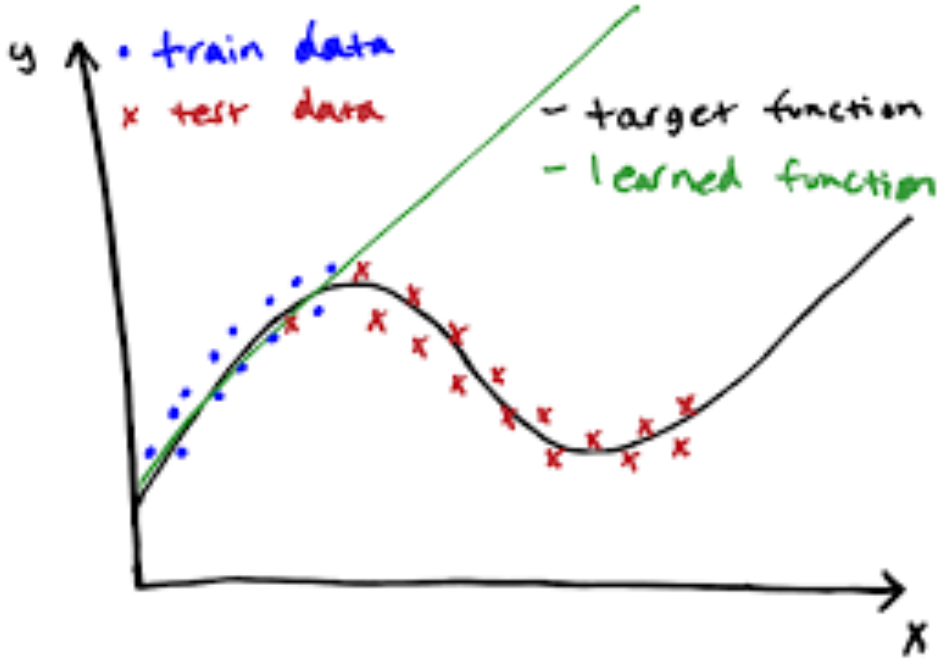
식물의 수확량: Yield of the plant → Dependent variable
식물의 유전자형: Genotype of the plant → Independent variable
식물의 키: Height of the plant → Covariate

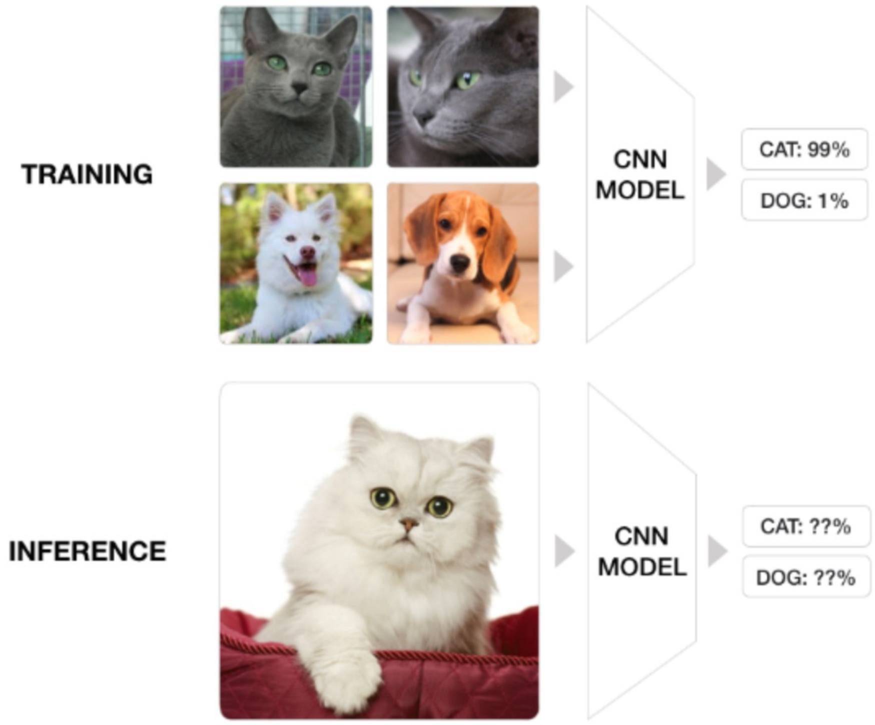
Internal Covariate Shift

◈Covariate Shift (공변량 변화) (or Dataset Distribution Shift)

- It is defined as a change in the distribution of dataset
- It usually occurs when the distribution of input data shifts between the training environment and test environment





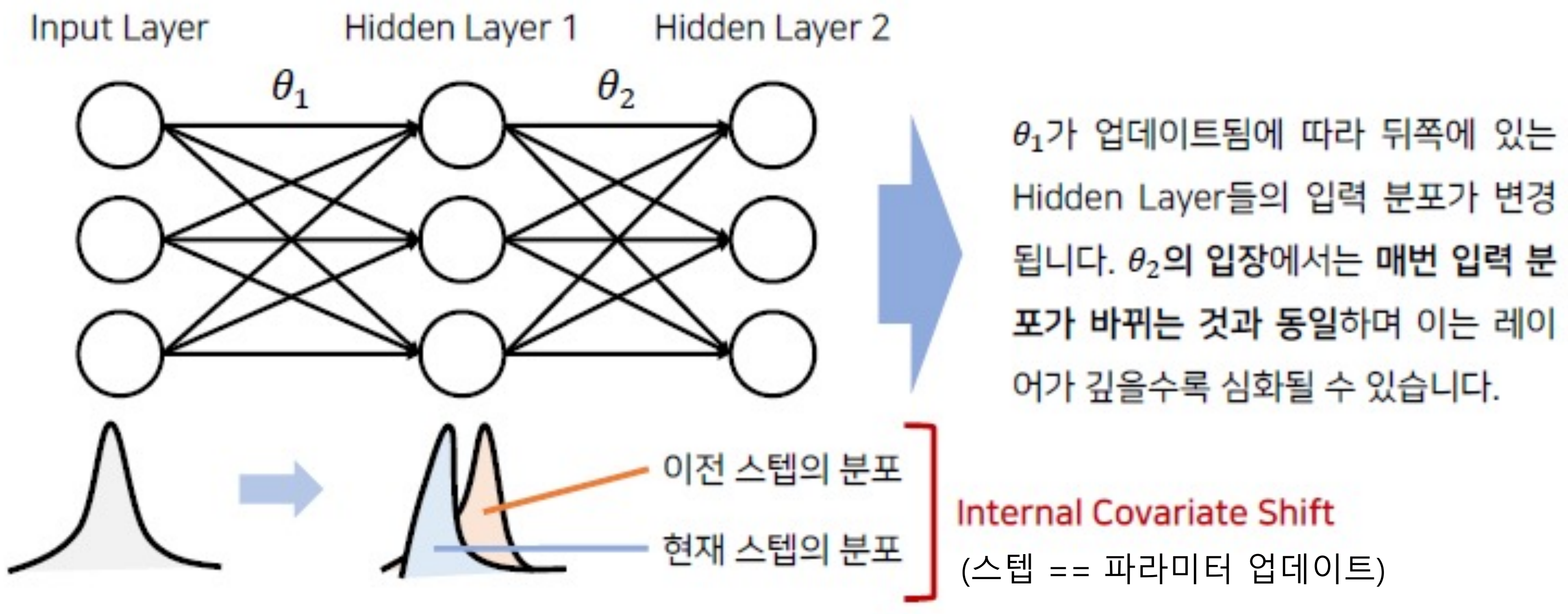


Internal Covariate Shift

◈Internal Covariate Shift

– It refers to the phenomenon in deep neural networks where the distribution of each layer's inputs can change during training, slowing down the training process

– This can happen because the parameters of earlier layers are constantly changing as the network learns, causing the distribution of inputs to subsequent layers to shift

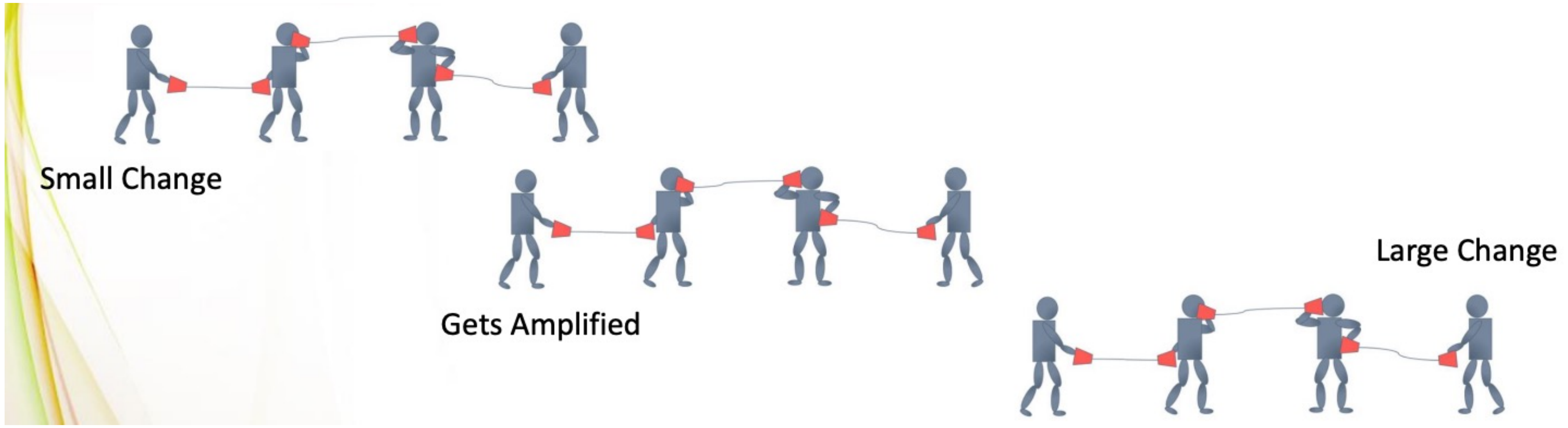


θ_1가 업데이트됨에 따라 뒤쪽에 있는 Hidden Layer들의 입력 분포가 변경됩니다. θ_2의 입장에서는 **매번 입력 분포가 바뀌는 것과 동일하며** 이는 레이어가 깊을수록 심화될 수 있습니다.

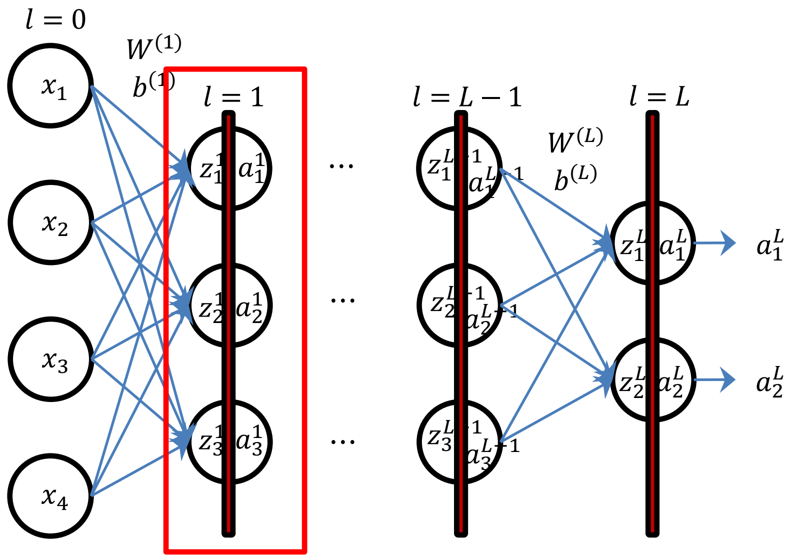
[Source] https://github.com/ndb796/Deep-Learning-Paper-Review-and-Practice

Internal Covariate Shift

- Problem at Training with "minibatch"
  - Training assumes the training data are all similarly distributed
    - Minibatches have similar distribution
  - In practice, each minibatch may have a different distribution
    - Internal covariate shift
  - Covariate shifts can be large!
    - All internal covariate shifts can affect training badly
    - This forces the higher layers to adapt to that drift, which slows down learning



Batch Normalization



◈Batch Normalization Training (Batch Data Shape: M x F)

- $z_{i,j}^{[b](l)}$: the jth feature of ith sample of a batch b at l layer
- M: batch size

3 Features ($j = 1, 2, 3$)

M Samples at a batch b

$z_{1,1}^{[b](1)}$	$z_{1,2}^{[b](1)}$	$z_{1,3}^{[b](1)}$
$z_{2,1}^{[b](1)}$	$z_{2,2}^{[b](1)}$	$z_{2,3}^{[b](1)}$
…	…	…
$z_{M,1}^{[b](1)}$	$z_{M,2}^{[b](1)}$	$z_{M,3}^{[b](1)}$

Done at only training

Mean and Std. Dev.

$$\mu_j^{[b](1)} = \frac{1}{M}\sum_{i}^{M} z_{i,j}^{[b](1)}$$

$$v_j^{[b](1)} = \frac{1}{M}\sum_{i}^{M} \left(z_{i,j}^{[b](1)} - \mu_j^{[b](1)}\right)^2$$

$$\hat{\mu}_j^{(1)} = (1-\tau)\hat{\mu}_j^{(1)} + \tau\mu_j^{[b](1)}$$

$$\hat{v}_j^{(1)} = (1-\tau)v_j^{(1)} + \tau v_j^{[b](1)}$$

- Running average for Mean and Std. Dev.
- It is averaged over each minibatch

Normalize

$$u_{i,j}^{[b](1)} = \frac{z_{i,j}^{[b](1)} - \mu_j^{[b](1)}}{\sqrt{v_j^{[b](1)} + \epsilon}}$$

Scale and Shift

$$\hat{z}_{i,j}^{[b](1)} = \gamma_j^{(1)} u_{i,j}^{[b](1)} + \beta_j^{(1)}$$

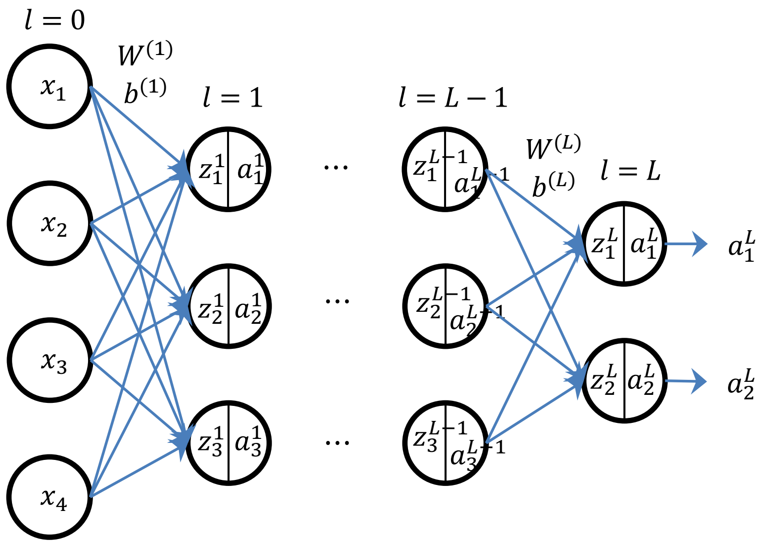
Activate

$$a_{i,j}^{[b](1)} = \sigma\left(\hat{z}_{i,j}^{[b](1)}\right)$$

Batch Normalization

◈Batch Normalization Training (Batch Data Shape: M x F)

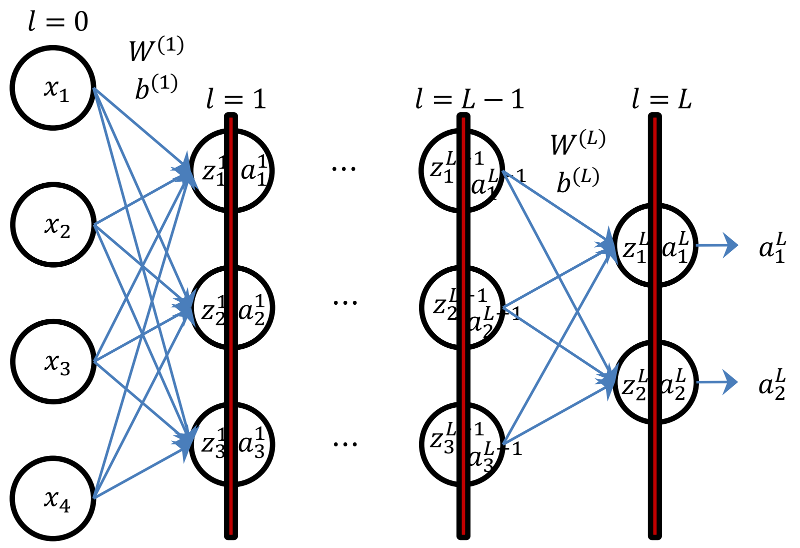
– $\boldsymbol{Z}^{[b](l)}$: the samples of a batch b at l layer



```
for each epoch do:
  for each batch $b$ do:
      $\boldsymbol{Z}^{[b](0)} = \boldsymbol{A}^{[b](0)} = \boldsymbol{X}^{[b]}$
      for $l = 0, 1, 2, \dots, L-1$ do:
          $\boldsymbol{Z}^{[b](l+1)} = \boldsymbol{W}^{(l+1)}\boldsymbol{A}^{[b](l)} + \boldsymbol{B}^{(l+1)}$
          $\boldsymbol{A}^{[b](l+1)} = \sigma\left(\boldsymbol{Z}^{[b](l+1)}\right)$
```

Batch Normalization

- **Batch Normalization Training (Batch Data Shape: M x F)**
  - $z_{i,j}^{[b](l)}$: the jth feature of ith sample of a batch b at l layer $\left(z_{i,j}^{[b](l)} \in Z^{[b](l+1)}\right)$
  - M: batch size



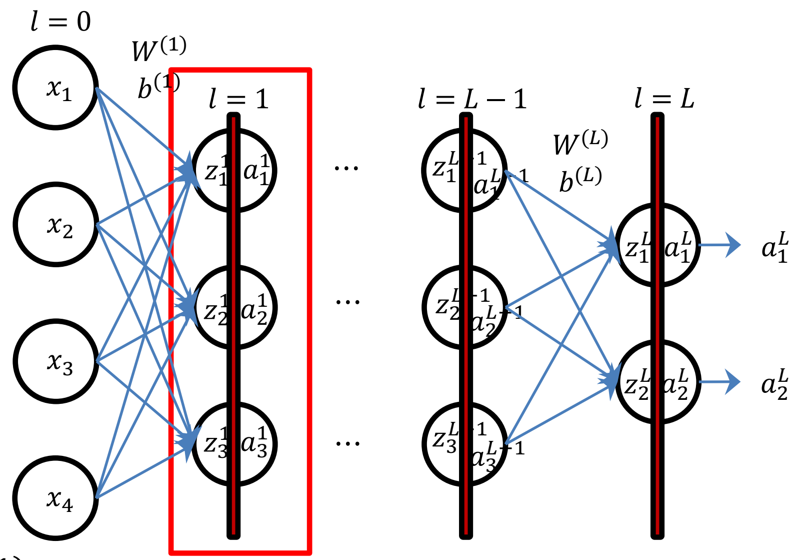
```
for each epoch do:
  for each batch $b$ do:
    $Z^{[b](0)} = A^{[b](0)} = X^{[b]}$
    for $l = 0, 1, 2, \ldots, L-1$ do:
      $Z^{[b](l+1)} = W^{(l+1)} A^{[b](l)} + B^{(l+1)}$
      $\boxed{\hat{Z}^{[b](l+1)} = BN\left(Z^{[b](l+1)}\right)}$
      $\mu_j^{[b](l+1)} = \frac{1}{M}\sum_i^M z_{i,j}^{[b](l+1)}$
      $v_j^{[b](l+1)} = \frac{1}{M}\sum_i^M \left(z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}\right)^2$
      $u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}}{\sqrt{v_j^{[b](l+1)} + \epsilon}}$
      $\hat{z}_{i,j}^{[b](l+1)} = \gamma_j^{(l+1)} u_{i,j}^{[b](l+1)} + \beta_j^{(l+1)}$
      $\hat{\mu}_j^{(l+1)} = (1-\tau)\hat{\mu}_j^{(l+1)} + \tau\mu_j^{[b](l+1)}$
      $\hat{v}_j^{(l+1)} = (1-\tau)v_j^{(l+1)} + \tau v_j^{[b](l+1)}$
      $A^{[b](l+1)} = \sigma\left(\hat{Z}^{[b](l+1)}\right)$
```

Learnable Parameters

Batch Normalization

◈ Batch Normalization Training (Batch Data Shape: M x F)

- $\mu_j^{[b](l)}$: the mean of jth feature of bth batch at l layer
- $v_j^{[b](l)}$: the variance of jth feature of bth batch at l layer



$\boldsymbol{Z}^{(1)}$ $\boldsymbol{U}^{(1)}$ $\hat{\boldsymbol{Z}}^{(1)}$ $\boldsymbol{A}^{(1)}$

$$\hat{\mu}_j^{(1)} = 0, \qquad \hat{v}_j^{(1)} = 0$$

Batch 1

$$\begin{tikzcd}
 & \mu_j^{[1](1)} \arrow[dr] & & \gamma_j^{(1)} \arrow[dr] & & \\
z_{i,j}^{(1)} \arrow[ur] \arrow[dr] & & u_{i,j}^{[1](1)} \arrow[ur] \arrow[dr] & & \hat{z}_{i,j}^{[1](1)} \arrow[r, "\text{activation}"] & a_{i,j}^{[1](1)} \arrow[lllll, bend left, "\text{backpropagation}"] \\
 & v_j^{[1](1)} \arrow[ur] & & \beta_j^{(1)}\ \text{Trained} \arrow[ur] & &
\end{tikzcd}$$

$$\hat{\mu}_j^{(1)} = (1-\tau)\hat{\mu}_j^{(1)} + \tau\mu_j^{[1](1)}$$
$$\hat{v}_j^{(1)} = (1-\tau)\hat{v}_j^{(1)} + \tau v_j^{[1](1)}$$

Batch 2

$$\begin{tikzcd}
 & \mu_j^{[2](1)} \arrow[dr] & & \gamma_j^{(1)} \arrow[dr] & & \\
z_{i,j}^{(1)} \arrow[ur] \arrow[dr] & & u_{i,j}^{[2](1)} \arrow[ur] \arrow[dr] & & \hat{z}_{i,j}^{[2](1)} \arrow[r, "\text{activation}"] & a_{i,j}^{[2](1)} \arrow[lllll, bend left, "\text{backpropagation}"] \\
 & v_j^{[2](1)} \arrow[ur] & & \beta_j^{(1)}\ \text{Trained} \arrow[ur] & &
\end{tikzcd}$$

$$\hat{\mu}_j^{(1)} = (1-\tau)\hat{\mu}_j^{(1)} + \tau\mu_j^{[2](1)}$$
$$\hat{v}_j^{(1)} = (1-\tau)\hat{v}_j^{(1)} + \tau v_j^{[2](1)}$$

⋮

Batch b

$$\begin{tikzcd}
 & \mu_j^{[b](1)} \arrow[dr] & & \gamma_j^{(1)} \arrow[dr] & & \\
z_{i,j}^{(1)} \arrow[ur] \arrow[dr] & & u_{i,j}^{[b](1)} \arrow[ur] \arrow[dr] & & \hat{z}_{i,j}^{[b](1)} \arrow[r, "\text{activation}"] & a_{i,j}^{[b](1)} \arrow[lllll, bend left, "\text{backpropagation}"] \\
 & v_j^{[b](1)} \arrow[ur] & & \beta_j^{(1)}\ \text{Trained} \arrow[ur] & &
\end{tikzcd}$$

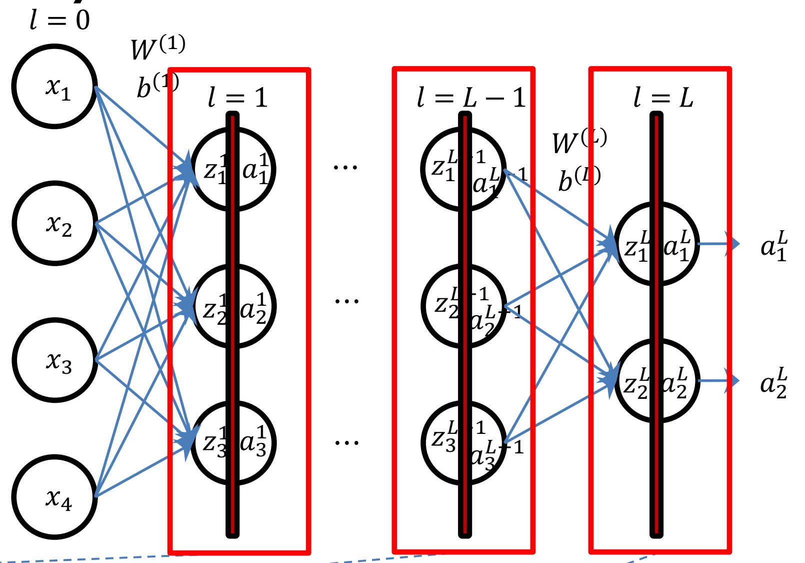
$$\hat{\mu}_j^{(1)} = (1-\tau)\hat{\mu}_j^{(1)} + \tau\mu_j^{[b](1)}$$
$$\hat{v}_j^{(1)} = (1-\tau)\hat{v}_j^{(1)} + \tau v_j^{[b](1)}$$

⋮

Batch Normalization

◈Batch Normalization Training (Batch Data Shape: M x F)

- $\hat{\mu}_j^{(l)}$ & $\hat{v}_j^{(l)}$: Running mean and variance of jth feature at l layer
  - Calculated by forward pass
  - Running (exponentially weighted) average over each batch
- $\gamma_j^{(l)}$ & $\beta_j^{(l)}$: Scale and Shift parameters of jth feature at l layer
  - Trained by backward pass



$$\hat{\mu}_j^{(1)} = (1-\tau)\hat{\mu}_j^{(1)} + \tau\mu_j^{[b](1)}$$
$$\hat{v}_j^{(1)} = (1-\tau)\hat{v}_j^{(1)} + \tau v_j^{[b](1)}$$
$$\gamma_j^{(1)}$$
$$\beta_j^{(1)}$$

…

$$\hat{\mu}_j^{(L-1)} = (1-\tau)\hat{\mu}_j^{(L-1)} + \tau\mu_j^{[b](L-1)}$$
$$\hat{v}_j^{(L-1)} = (1-\tau)\hat{v}_j^{(L-1)} + \tau v_j^{[b](L-1)}$$
$$\gamma_j^{(L-1)}$$
$$\beta_j^{(L-1)}$$

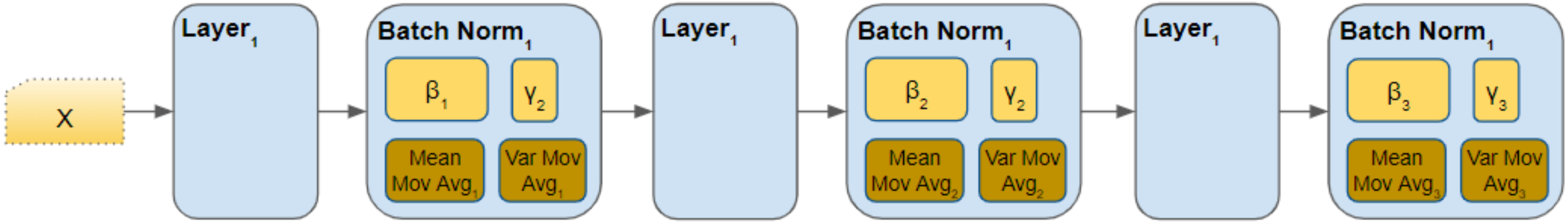
$$\hat{\mu}_j^{(L)} = (1-\tau)\hat{\mu}_j^{(L)} + \tau\mu_j^{[b](L)}$$
$$\hat{v}_j^{(L)} = (1-\tau)\hat{v}_j^{(L)} + \tau v_j^{[b](L)}$$
$$\gamma_j^{(L)}$$
$$\beta_j^{(L)}$$

Batch Normalization

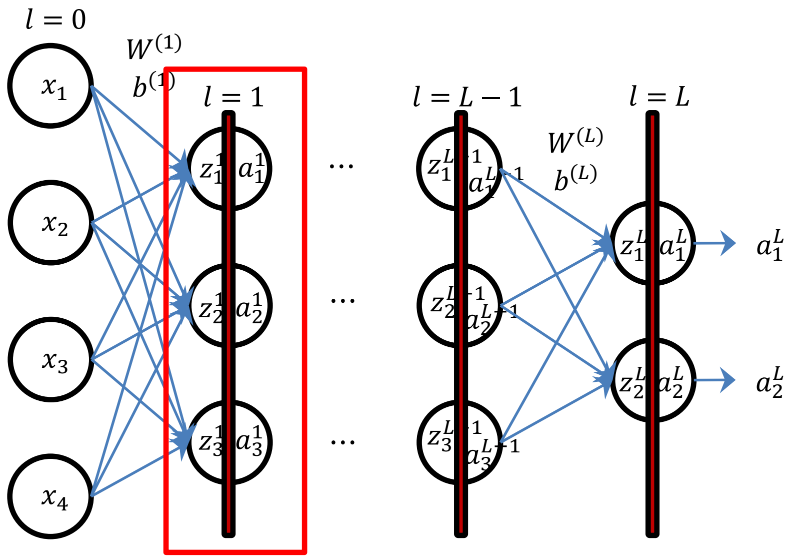
◈ Batch Normalization Training (Batch Data Shape: M x F)

– Four values per layer are determined by batched values seen during training

- $\hat{\mu}_j^{(l)}$ & $\hat{v}_j^{(l)}$: Running mean and variance of jth feature at l layer
- $\gamma_j^{(l)}$ & $\beta_j^{(l)}$: Scale and Shift parameters of jth feature at l layer



Batch Normalization



◈Validation or Test with Batch Normalization (Batch Data Shape: M x F or 1 x F)

3 Features ($j = 1, 2, 3$)

$$\begin{array}{c|ccc} & & & \\ \hline \text{M Samples} & z_{1,1}^{(1)} & z_{1,2}^{(1)} & z_{1,3}^{(1)} \\ & z_{2,1}^{(1)} & z_{2,2}^{(1)} & z_{2,3}^{(1)} \\ & \cdots & \cdots & \cdots \\ & z_{M,1}^{(1)} & z_{M,2}^{(1)} & z_{M,3}^{(1)} \end{array}$$

3 Features ($j = 1, 2, 3$)

$$\begin{array}{c|ccc} \text{1 Samples} & z_{1,1}^{(1)} & z_{1,2}^{(1)} & z_{1,3}^{(1)} \end{array}$$

Normalize

$$u_{i,j}^{(1)} = \frac{z_{i,j}^{(1)} - \hat{\mu}_j^{(1)}}{\sqrt{\hat{v}_j^{(1)} + \epsilon}}$$

Scale and Shift

$$\hat{z}_{i,j}^{(1)} = \gamma_j^{(1)} u_{i,j}^{(1)} + \beta_j^{(1)}$$

Activate

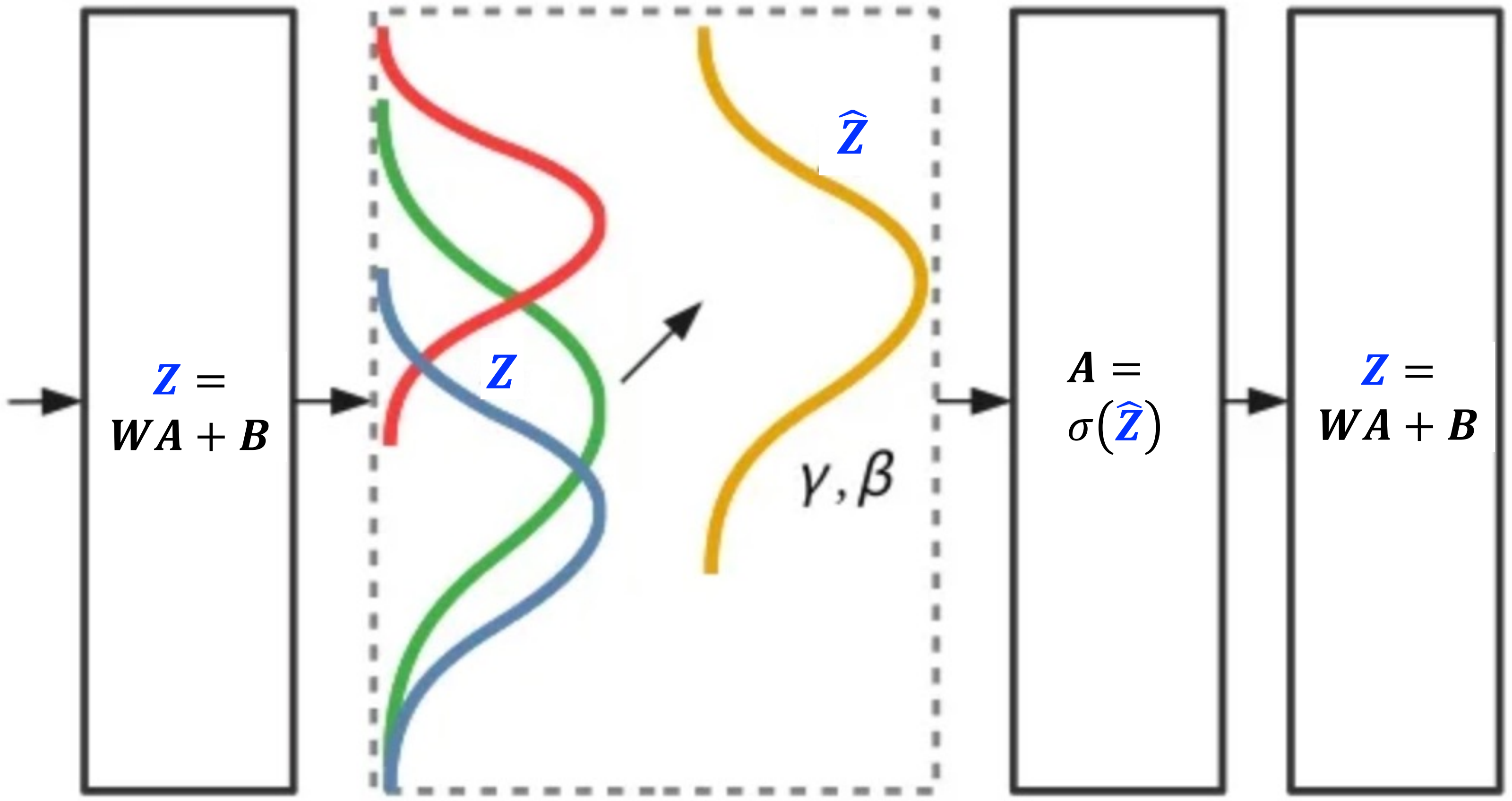
$$a_{i,j}^{(1)} = \sigma\left(\hat{z}_{i,j}^{(1)}\right)$$

Running average for Mean and Std. Dev.

Batch Normalization

◈Why Does Batch Normalization Work?

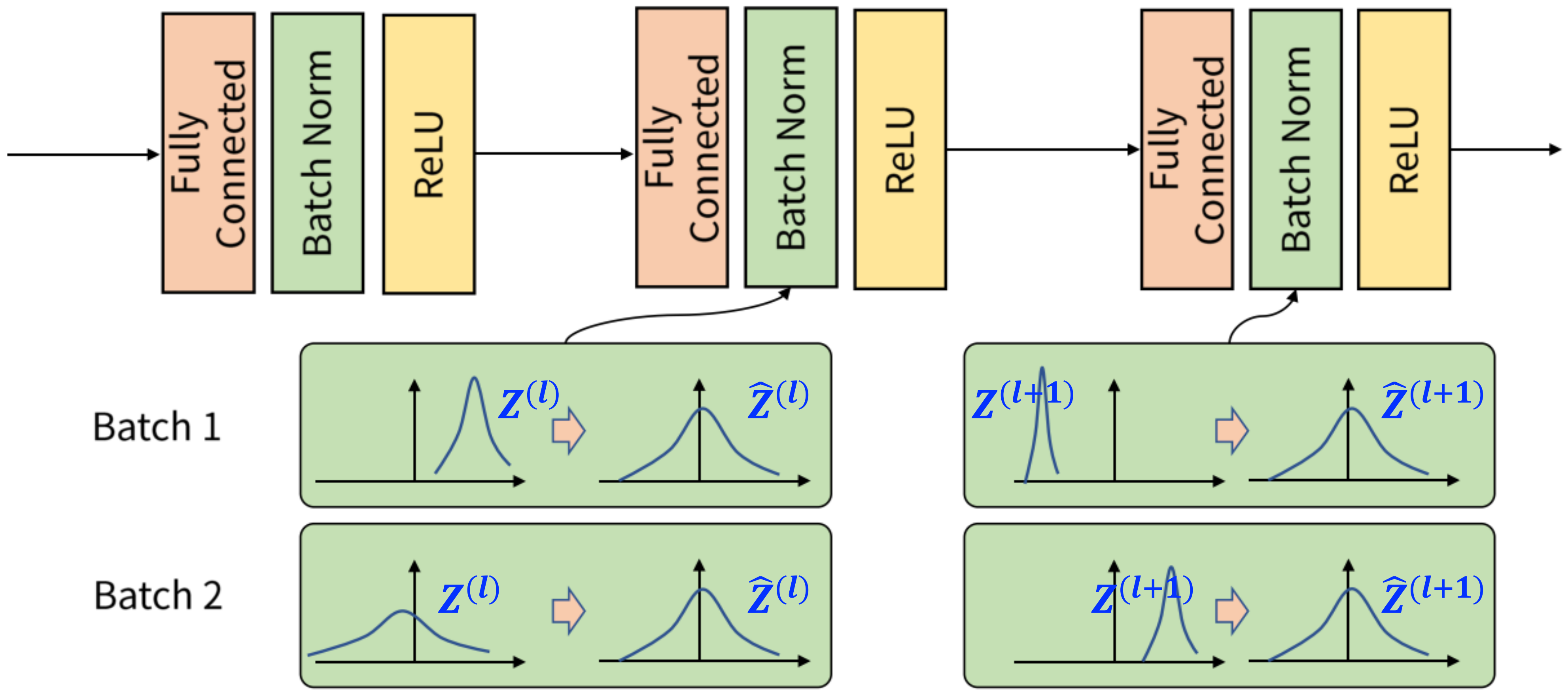
– Ensure the layer output distribution is adjusted, so that it is helpful for the training



Batch Normalization

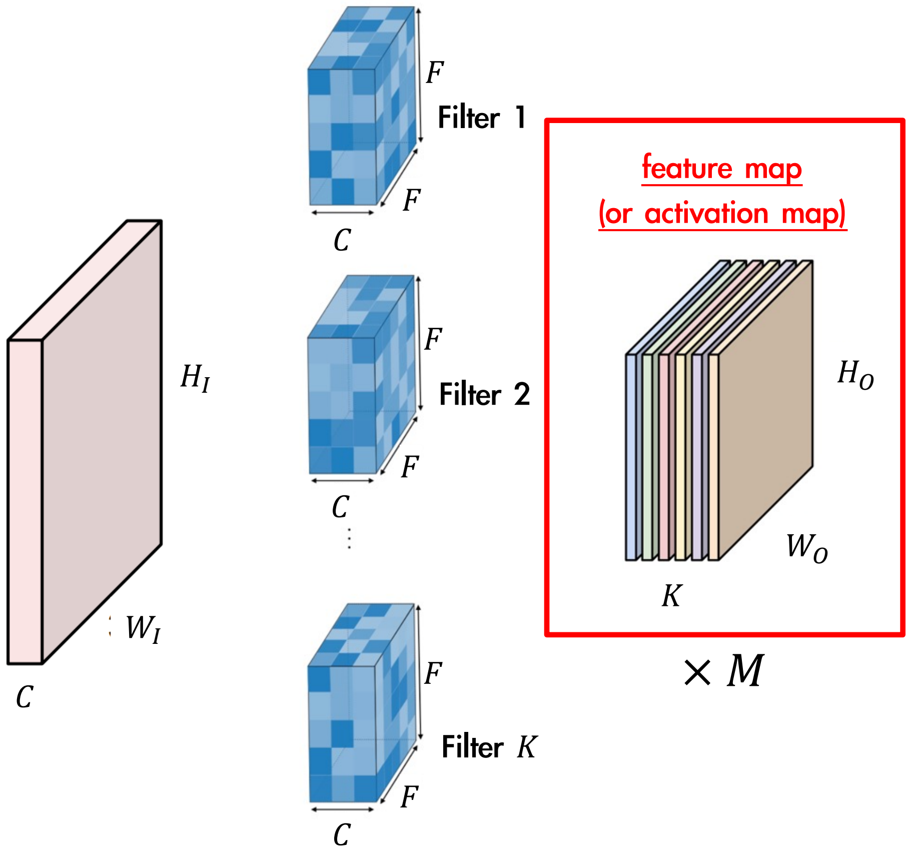
◈ Why Does Batch Normalization Work?

– Ensure the layer output distribution is adjusted, so that it is helpful for the training



Batch Normalization (BatchNorm2D) at CNN

◈ Batch Normalization Training (Batch Data Shape: $M \times K \times H_O \times W_O$)

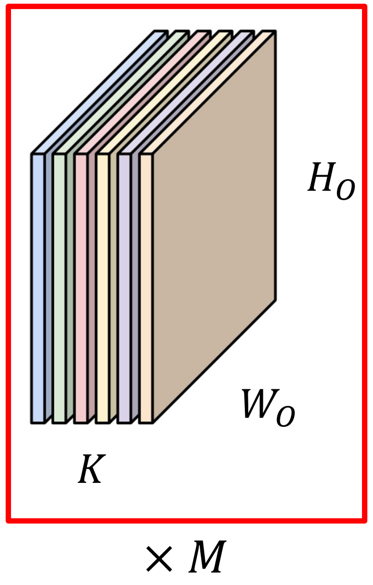


```
for each epoch do:
  for each batch $b$ do:
    $Z^{[b](0)} = A^{[b](0)} = X^{[b]}$
    for $l = 0, 1, 2, \dots, L-1$ do:
      $Z^{[b](l+1)} = Conv2D(A^{[b](l)})$
      $\hat{Z}^{[b](l+1)} = BN(Z^{[b](l+1)})$
        $\mu_j^{[b](l+1)} = \frac{1}{M}\sum_i^M z_{i,j}^{[b](l+1)}$
        $v_j^{[b](l+1)} = \frac{1}{M}\sum_i^M \left(z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}\right)^2$
        $u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}}{\sqrt{v_j^{[b](l+1)} + \epsilon}}$
        $\hat{z}_{i,j}^{[b](l+1)} = \gamma_j^{(l+1)} u_{i,j}^{[b](l+1)} + \beta_j^{(l+1)}$
        $\hat{\mu}_j^{(l+1)} = (1-\tau)\hat{\mu}_j^{(l+1)} + \tau\mu_j^{[b](l+1)}$
        $\hat{v}_j^{(l+1)} = (1-\tau)v_j^{(l+1)} + \tau v_j^{[b](l+1)}$
      $A^{[b](l+1)} = \sigma(\hat{Z}^{[b](l+1)})$
```

Learnable Parameters

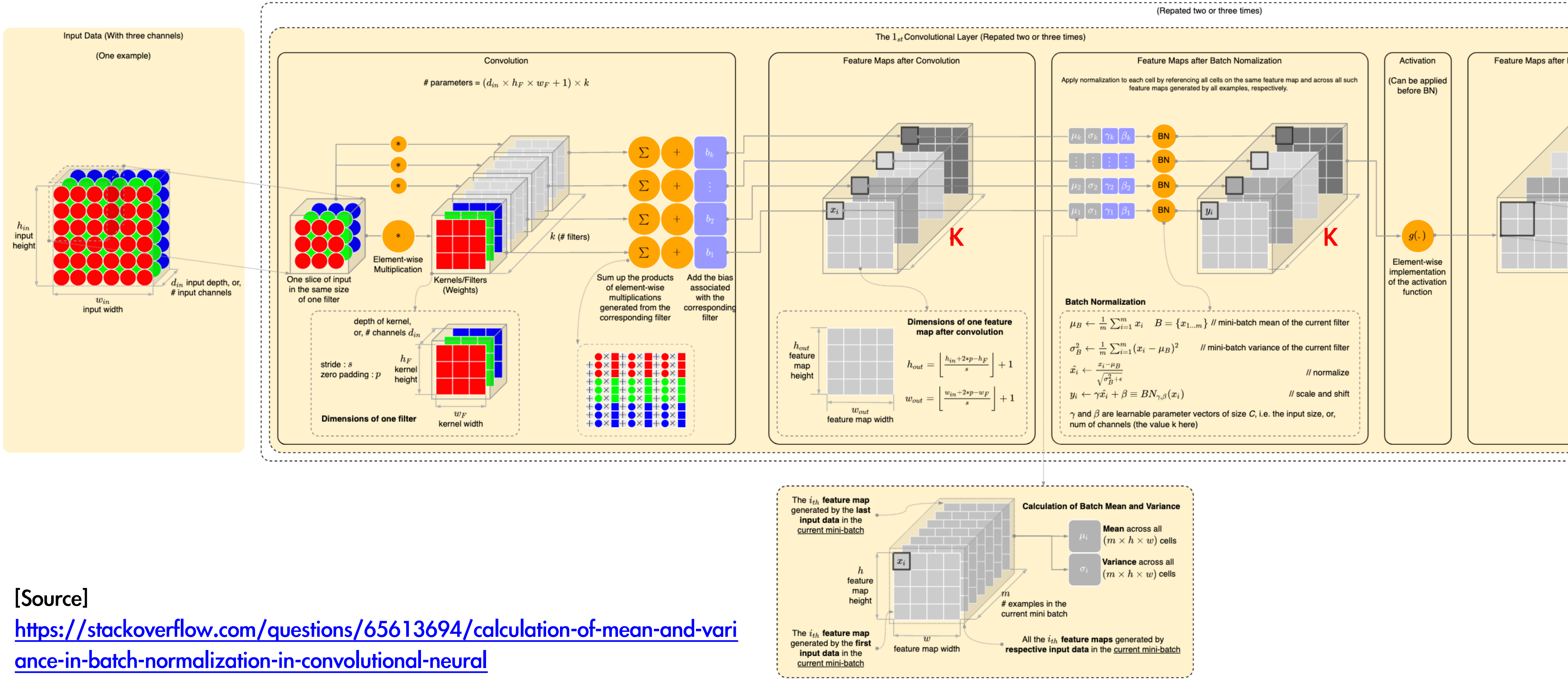
Batch Normalization (BatchNorm2D) at CNN

- Batch Normalization with Convolutional Layer (Batch Data Shape: M x K x H_O x W_O)
  - Four values per layer are determined by batched values seen during training
    - Calculated by forward pass
      - Running (exponentially weighted) average over each batch
      - $\hat{\mu}_j^{(l)}$: 1 x K x 1 x 1 per layer $(j \in K)$
      - $\hat{v}_j^{(l)}$: 1 x K x 1 x 1 per layer $(j \in K)$
    - Trained by backward pass over each batch
      - $\gamma_j^{(l)}$: 1 x K x 1 x 1 per layer $(j \in K)$
      - $\beta_j^{(l)}$: 1 x K x 1 x 1 per layer $(j \in K)$



Batch Normalization (BatchNorm2D) at CNN

◈Batch Normalization with Convolutional Layer (Batch Data Shape: $M \times K \times H_O \times W_O$)



(Repated two or three times)
Input Data (With three channels)
(One example)
h_{in} input height
w_{in} input width
d_{in} input depth, or, # input channels
The 1_{st} Convolutional Layer (Repated two or three times)
Convolution
# parameters = $(d_{in} \times h_F \times w_F + 1) \times k$
One slice of input in the same size of one filter
Element-wise Multiplication
Kernels/Filters (Weights)
k (# filters)
Sum up the products of element-wise multiplications generated from the corresponding filter
Add the bias associated with the corresponding filter
b_k
b_2
b_1
depth of kernel, or, # channels d_{in}
stride : s
zero padding : p
h_F kernel height
w_F kernel width
Dimensions of one filter
Feature Maps after Convolution
x_i
K
Dimensions of one feature map after convolution
h_{out} feature map height
w_{out} feature map width
$h_{out} = \left\lfloor \frac{h_{in}+2*p-h_F}{s} \right\rfloor + 1$
$w_{out} = \left\lfloor \frac{w_{in}+2*p-w_F}{s} \right\rfloor + 1$
Feature Maps after Batch Nomalization
Apply normalization to each cell by referencing all cells on the same feature map and across all such feature maps generated by all examples, respectively.
μ_k σ_k γ_k β_k
μ_2 σ_2 γ_2 β_2
μ_1 σ_1 γ_1 β_1
BN
y_i
K
Batch Normalization
$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad B = \{x_{1...m}\}$ // mini-batch mean of the current filter
$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ // mini-batch variance of the current filter
$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ // normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ // scale and shift
γ and β are learnable parameter vectors of size C, i.e. the input size, or, num of channels (the value k here)
Activation
(Can be applied before BN)
$g(.)$
Element-wise implementation of the activation function
Feature Maps after
The i_{th} feature map generated by the last input data in the current mini-batch
Calculation of Batch Mean and Variance
Mean across all $(m \times h \times w)$ cells
Variance across all $(m \times h \times w)$ cells
μ_i
σ_i
h feature map height
w feature map width
m # examples in the current mini batch
The i_{th} feature map generated by the first input data in the current mini-batch
All the i_{th} feature maps generated by respective input data in the current mini-batch

[Source]
https://stackoverflow.com/questions/65613694/calculation-of-mean-and-variance-in-batch-normalization-in-convolutional-neural

link_dl/_01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py

Batch Normalization with PyTorch

◈BatchNorm1d & BatchNorm2d

```python
class MyModel(nn.Module):
  def __init__(self, in_channels, n_output):
    super().__init__()
    self.model = nn.Sequential(
      # B x 1 x 28 x 28 --> B x 6 x 24 x 24
      nn.Conv2d(
        in_channels=in_channels, out_channels=6,
        kernel_size=(5, 5), stride=(1, 1)
      ),
      # B x 6 x 24 x 24 --> B x 6 x 12 x 12
      nn.MaxPool2d(kernel_size=2, stride=2),

      nn.BatchNorm2d(
        num_features=6, eps=1e-05, momentum=0.1
      ),

      nn.ReLU(),
      ...
```

```
for each epoch do:
  for each batch $b$ do:
    $Z^{[b](0)} = A^{[b](0)} = X^{[b]}$
    for $l = 0, 1, 2, \dots, L-1$ do:
      $Z^{[b](l+1)} = Conv2D(A^{[b](l)})$
        $\hat{Z}^{[b](l+1)} = BN(Z^{[b](l+1)})$
        $\mu_j^{[b](l+1)} = \frac{1}{M}\sum_i^M z_{i,j}^{[b](l+1)}$
        $v_j^{[b](l+1)} = \frac{1}{M}\sum_i^M \left(z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}\right)^2$
        $u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}}{\sqrt{v_j^{[b](l+1)} + \epsilon}}$
        $\hat{z}_{i,j}^{[b](l+1)} = \gamma_j^{(l+1)} u_{i,j}^{[b](l+1)} + \beta_j^{(l+1)}$
          Learnable Parameters
        $\hat{\mu}_j^{(l+1)} = (1-\tau)\hat{\mu}_j^{(l+1)} + \tau\mu_j^{[b](l+1)}$
        $\hat{v}_j^{(l+1)} = (1-\tau)v_j^{(l+1)} + \tau v_j^{[b](l+1)}$
      $A^{[b](l+1)} = \sigma(\hat{Z}^{[b](l+1)})$
```

link_dl/_01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py

Batch Normalization with PyTorch

◈BatchNorm1d & BatchNorm2d

```python
class MyModel(nn.Module):
  def __init__(self, in_channels, n_output):
    super().__init__()
    self.model = nn.Sequential(
      ...

      # B x 6 x 12 x 12 --> B x 864
      nn.Flatten(),
      nn.Linear(864, 256),

      nn.BatchNorm1d(
        num_features=256, eps=1e-05, momentum=0.1
      ),

      nn.ReLU(),
      nn.Linear(256, n_output),
    )
```

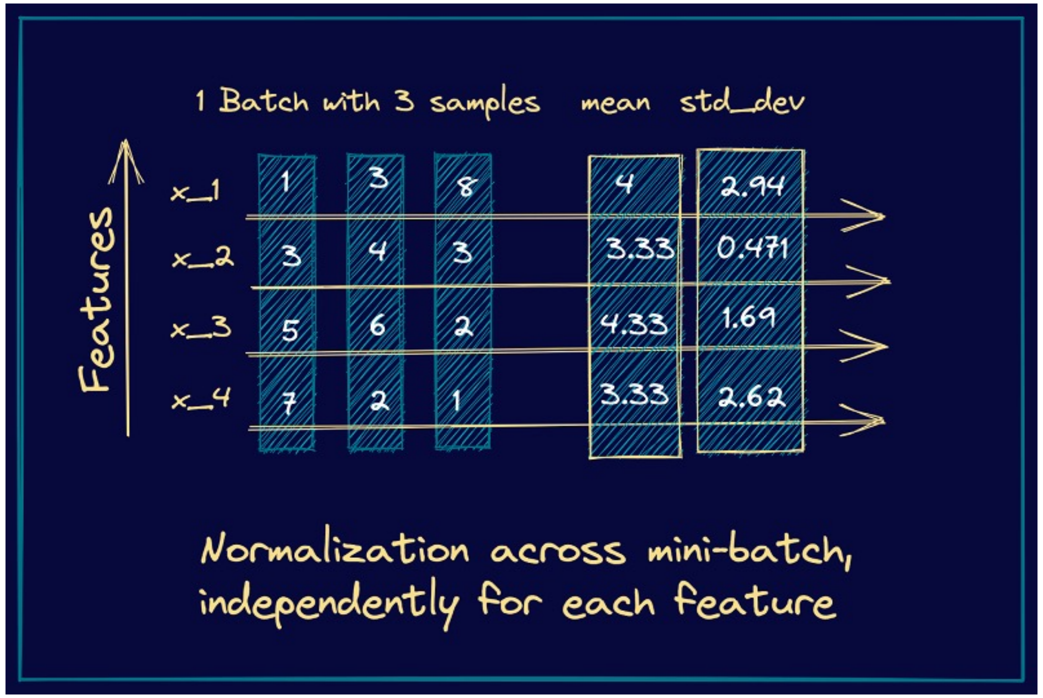
```
for each epoch do:
  for each batch $b$ do:
    $Z^{[b](0)} = A^{[b](0)} = X^{[b]}$
    for $l = 0, 1, 2, \dots, L-1$ do:
      $Z^{[b](l+1)} = W^{(l+1)} A^{[b](l)} + B^{(l+1)}$
      $\boxed{\hat{Z}^{[b](l+1)} = BN\left(Z^{[b](l+1)}\right)}$
      $\mu_j^{[b](l+1)} = \frac{1}{M} \sum_i^M z_{i,j}^{[b](l+1)}$
      $v_j^{[b](l+1)} = \frac{1}{M} \sum_i^M \left(z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}\right)^2$
      $u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}}{\sqrt{v_j^{[b](l+1)} + \epsilon}}$
      $\hat{z}_{i,j}^{[b](l+1)} = \gamma_j^{(l+1)} u_{i,j}^{[b](l+1)} + \beta_j^{(l+1)}$
      Learnable Parameters
      $\hat{\mu}_j^{(l+1)} = (1-\tau)\hat{\mu}_j^{(l+1)} + \tau \mu_j^{[b](l+1)}$
      $\hat{v}_j^{(l+1)} = (1-\tau) v_j^{(l+1)} + \tau v_j^{[b](l+1)}$
      $A^{[b](l+1)} = \sigma\left(\hat{Z}^{[b](l+1)}\right)$
```

Layer Normalization

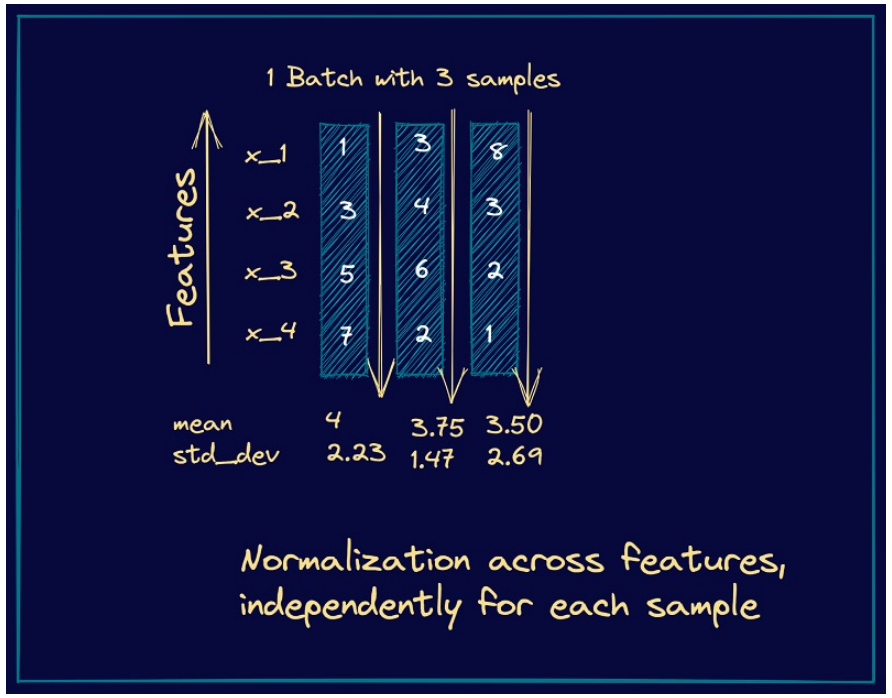
◈Layer Normalization

- Assume that an input has d features (d-dimensional vector) and there are B inputs in a batch
- Layer normalization is done along the length of the d-dimensional vector and not across the batch of size B
  - Normalizing across all features for each of the inputs to a layer removes the dependence on batches
  - This makes layer normalization well suited for sequence models (e.g., transformers or RNNs)

Batch Norm



Layer Norm



Layer Normalization

◈Layer Normalization (Batch Data Shape: M x F)

- $z_{i,j}^{[b](l)}$: the jth feature of ith sample of a batch b at l layer
- J: feature size
- ~~M: batch size (Not Used)~~

3 Features ($j = 1, 2, 3$)

M Samples at a batch b

$z_{1,1}^{[b](1)}$	$z_{1,2}^{[b](1)}$	$z_{1,3}^{[b](1)}$
$z_{2,1}^{[b](1)}$	$z_{2,2}^{[b](1)}$	$z_{2,3}^{[b](1)}$
…	…	…
$z_{M,1}^{[b](1)}$	$z_{M,2}^{[b](1)}$	$z_{M,3}^{[b](1)}$

```
for each epoch do:
  for each batch $b$ do:
    $Z^{[b](0)} = A^{[b](0)} = X^{[b]}$
    for $l = 0, 1, 2, \dots, L-1$ do:
      $Z^{[b](l+1)} = W^{(l+1)} A^{[b](l)} + B^{(l+1)}$
      $\hat{Z}^{[b](l+1)} = LN\left(Z^{[b](l+1)}\right)$
      $A^{[b](l+1)} = \sigma\left(\hat{Z}^{[b](l+1)}\right)$
```

$$\boxed{\hat{Z}^{[b](l+1)} = LN\left(Z^{[b](l+1)}\right)}$$

$$\mu_i^{[b](l+1)} = \frac{1}{J}\sum_j^J z_{i,j}^{[b](l+1)}$$

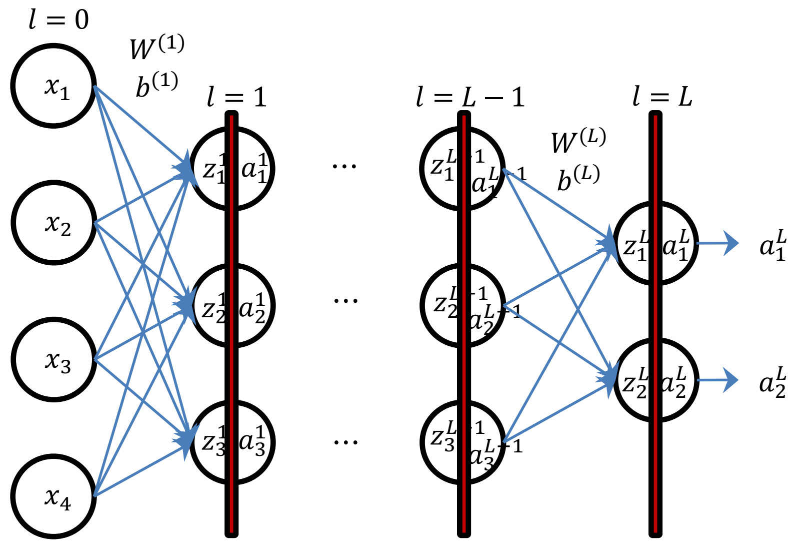
$$v_i^{[b](l+1)} = \frac{1}{J}\sum_j^J \left(z_{i,j}^{[b](l+1)} - \mu_i^{[b](l+1)}\right)^2$$

$$u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_i^{[b](l+1)}}{\sqrt{v_i^{[b](l+1)} + \epsilon}}$$

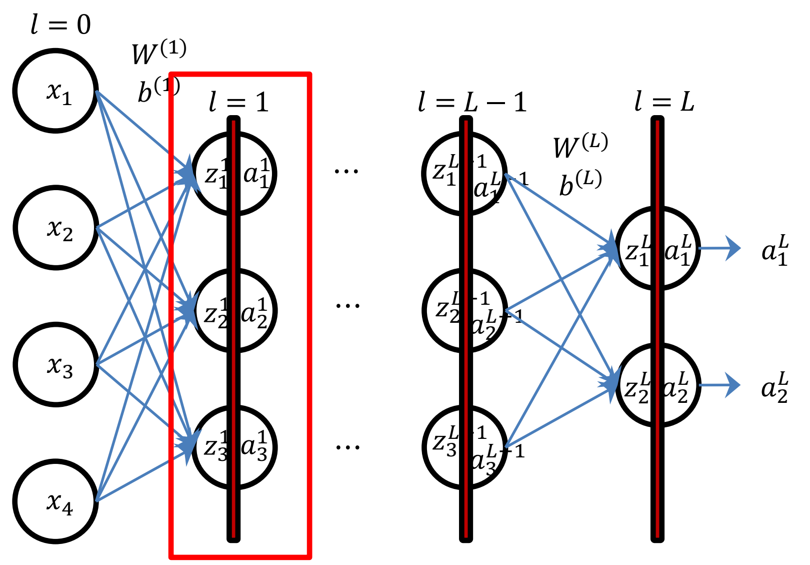
$$\hat{z}_{i,j}^{(l+1)} = \gamma^{(l+1)} u_{i,j}^{[b](l+1)} + \beta^{(l+1)}$$

Learnable Parameters

$$A^{[b](l+1)} = \sigma\left(\hat{Z}^{[b](l+1)}\right)$$



Layer Normalization



◈Training, Validation, and Testing with Layer Normalization (Batch Data Shape: M x F)

– $z_j^{(l)}$: the jth feature of ith sample of a batch b at l layer

– J: feature size

3 Features ($j = 1, 2, 3$)

M Samples at a batch b

$z_{1,1}^{[b](1)}$	$z_{1,2}^{[b](1)}$	$z_{1,3}^{[b](1)}$
$z_{2,1}^{[b](1)}$	$z_{2,2}^{[b](1)}$	$z_{2,3}^{[b](1)}$
…	…	…
$z_{M,1}^{[b](1)}$	$z_{M,2}^{[b](1)}$	$z_{M,3}^{[b](1)}$

Mean and Std. Dev.

$$\mu_i^{[b](1)} = \frac{1}{J}\sum_j^J z_{i,j}^{[b](1)}$$

$$v_i^{[b](1)} = \frac{1}{J}\sum_j^J \left(z_{i,j}^{[b](1)} - \mu_i^{[b](1)}\right)^2$$

Normalize

$$u_{i,j}^{[b](1)} = \frac{z_{i,j}^{[b](1)} - \mu_i^{[b](1)}}{\sqrt{v_i^{[b](1)} + \epsilon}}$$

Scale and Shift

$$\hat{z}_{i,j}^{[b](1)} = \gamma^{(1)} u_{i,j}^{[b](1)} + \beta^{(1)}$$

Activate

$$a_{i,j}^{[b](1)} = \sigma\left(\hat{z}_{i,j}^{[b](1)}\right)$$

~~$\hat{\mu}_j^{(1)} = (1-\tau)\hat{\mu}_j^{(1)} + \tau\mu_j^{[b](1)}$~~

~~$\hat{v}_j^{(1)} = (1-\tau)v_j^{(1)} + \tau v_j^{[b](1)}$~~

~~- Running average for Mean and Std. Dev.~~

~~- It is averaged over each minibatch~~

Layer Normalization

◈Advantages of Layer Normalization (against Batch Normalization) - 1/2

- Batch Independence
  - It is not dependent on batch size
  - This is particularly useful for models or situations with a small batch or even a batch size of one
- Sequence Models
  - LN is more straightforward to apply in recurrent models like Transformer or RNNs than BN
- No Need for Momentum or Running Statistics
  - Unlike BN, LN doesn't require tracking running statistics (mean and variance) for inference, which simplifies the process and reduces potential sources of error
- Consistent Behavior during Training and Inference
  - BN's behavior can change between training and inference due to the use of running statistics during inference
  - In contrast, LN behaves the same way during both training and inference

link_dl/_01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py

Layer Normalization with PyTorch

◈ Layer Norm (1/2)

```
def get_cnn_model_with_dropout_and_layer_normalization():
  class MyModel(nn.Module):
    def __init__(self, in_channels, n_output):
      super().__init__()

      self.model = nn.Sequential(
        # 3 x 32 x 32 --> 6 x (32 - 5 + 1) x (32 - 5 + 1) = 6 x 28 x 28
        nn.Conv2d(in_channels=in_channels, out_channels=6, kernel_size=(5, 5), stride=(1, 1)),
        # 6 x 28 x 28 --> 6 x 14 x 14
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.LayerNorm(normalized_shape=[6, 14, 14], eps=1e-05),
        nn.ReLU(),
        # 6 x 14 x 14 --> 16 x (14 - 5 + 1) x (14 - 5 + 1) = 16 x 10 x 10
        nn.Conv2d(in_channels=6, out_channels=16, kernel_size=(5, 5), stride=(1, 1)),
        # 16 x 10 x 10 --> 16 x 5 x 5
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.LayerNorm(normalized_shape=[16, 5, 5], eps=1e-05),
        nn.ReLU(),
```

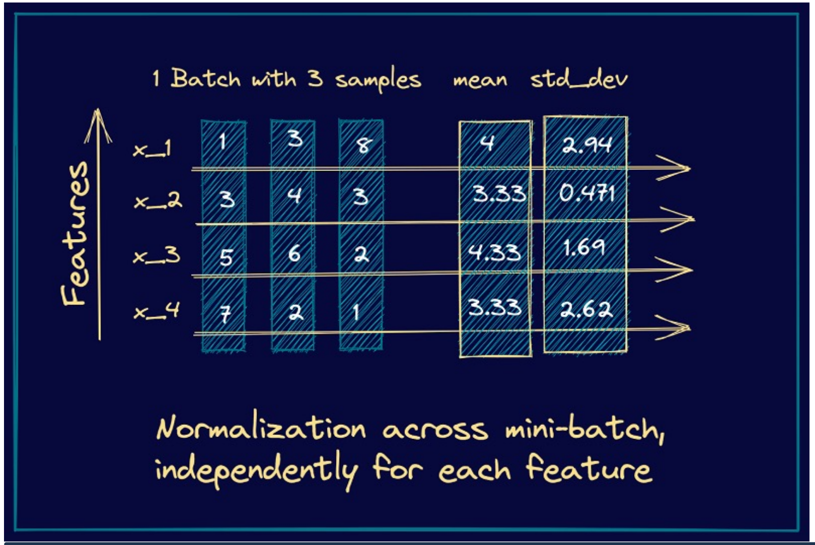
link_dl/_01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py

Layer Normalization with PyTorch

◈ Layer Norm (2/2)

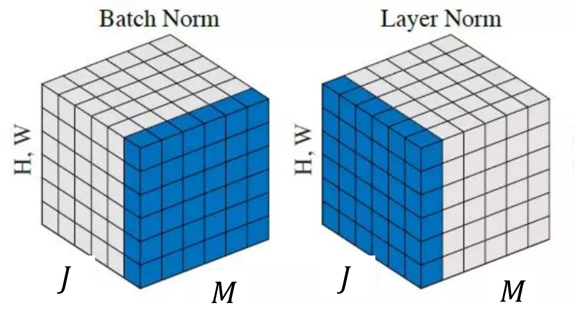
```
def get_cnn_model_with_dropout_and_batch_normalization():
  class MyModel(nn.Module):
    def __init__(self, in_channels, n_output):
      ...
      self.model = nn.Sequential(
        ...
        nn.Flatten(),
        nn.Dropout(p=0.5),      # p: dropout probability
        nn.Linear(400, 128),
        nn.LayerNorm(normalized_shape=[128], eps=1e-05),
        nn.ReLU(),
        nn.Dropout(p=0.5),      # p: dropout probability
        nn.Linear(128, n_output),
    )

    def forward(self, x):
      x = self.model(x)
      return x
```

BN vs. LN

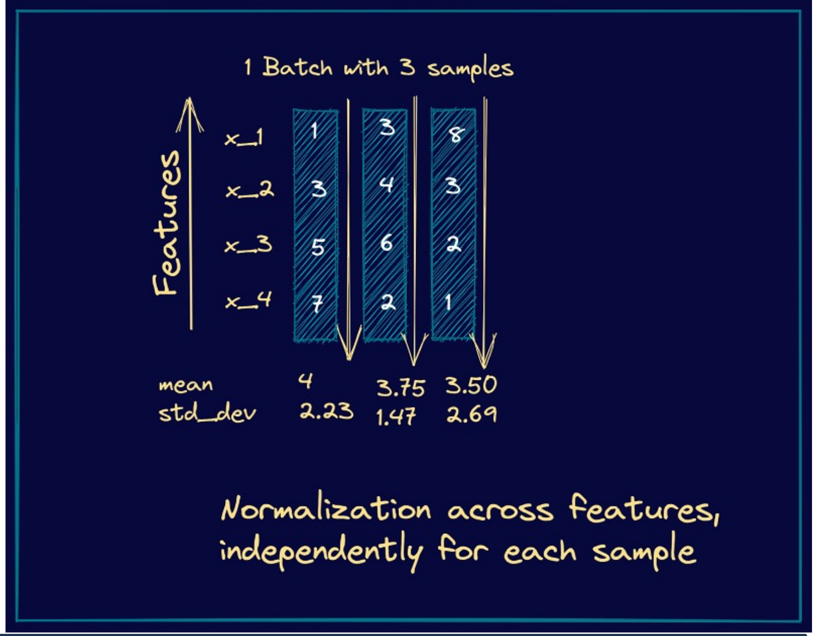


	1 Batch with 3 samples			mean	std_dev
x_1	1	3	8	4	2.94
x_2	3	4	3	3.33	0.471
x_3	5	6	2	4.33	1.69
x_4	7	2	1	3.33	2.62

Normalization across mini-batch, independently for each feature



$M = 3 \ (i = 1,2,3)$
$J = 4 \ (j = 1,2,3,4)$



	1 Batch with 3 samples		
x_1	1	3	8
x_2	3	4	3
x_3	5	6	2
x_4	7	2	1
mean	4	3.75	3.50
std_dev	2.23	1.47	2.69

Normalization across features, independently for each sample

$M = 3 \ (i = 1,2,3)$
$J = 4 \ (j = 1,2,3,4)$

Train

$$\boxed{\hat{Z}^{[b](l+1)} = BN\left(Z^{[b](l+1)}\right)}$$

$$\mu_j^{[b](l+1)} = \frac{1}{M}\sum_{i}^{M} z_{i,j}^{[b](l+1)}$$

$$v_j^{[b](l+1)} = \frac{1}{M}\sum_{i}^{M} \left(z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}\right)^2$$

$$u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_j^{[b](l+1)}}{\sqrt{v_j^{[b](l+1)} + \epsilon}} \qquad \hat{z}_{i,j}^{[b](l+1)} = \gamma_j^{(l+1)} u_{i,j}^{[b](l+1)} + \beta_j^{(l+1)}$$

Learnable Parameters

$$\hat{\mu}_j^{(l+1)} = (1-\tau)\hat{\mu}_j^{(l+1)} + \tau\mu_j^{[b](l+1)} \qquad \hat{v}_j^{(l+1)} = (1-\tau)v_j^{(l+1)} + \tau v_j^{[b](l+1)}$$

Inference

$$\boxed{\hat{Z}^{[b](l+1)} = BN\left(Z^{[b](l+1)}\right)}$$

$$u_{i,j}^{(l+1)} = \frac{z_{i,j}^{(l+1)} - \hat{\mu}_j^{(l+1)}}{\sqrt{\hat{v}_j^{(l+1)} + \epsilon}} \qquad \hat{z}_{i,j}^{(l+1)} = \gamma_j^{(l+1)} u_{i,j}^{(l+1)} + \beta_j^{(l+1)}$$

```
nn.BatchNorm2d(num_features=6)
nn.BatchNorm1d(num_features=256)
```

Train & Inference

$$\boxed{\hat{Z}^{[b](l+1)} = LN\left(Z^{[b](l+1)}\right)}$$

$$\mu_i^{[b](l+1)} = \frac{1}{J}\sum_{j}^{J} z_{i,j}^{[b](l+1)}$$

$$v_i^{[b](l+1)} = \frac{1}{J}\sum_{j}^{J} \left(z_{i,j}^{[b](l+1)} - \mu_i^{[b](l+1)}\right)^2$$

$$u_{i,j}^{[b](l+1)} = \frac{z_{i,j}^{[b](l+1)} - \mu_i^{[b](l+1)}}{\sqrt{v_i^{[b](l+1)} + \epsilon}} \qquad \hat{z}_{i,j}^{(l+1)} = \gamma^{(l+1)} u_{i,j}^{[b](l+1)} + \beta^{(l+1)}$$

Learnable Parameters

```
nn.LayerNorm(normalized_shape=[6, 14, 14])
nn.LayerNorm(normalized_shape=[256])
```

Normalization with PyTorch

◈CIFAR10 + CNN + Adam Optimizer + Weight Decay 0.002 + Dropout + [No Normalization, Batch Normalization, Layer Normalization]

– No Normalization

- python _01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py --wandb -v 1 -o 3 -w 0.002 --dropout -n 0

– Batch Normalization

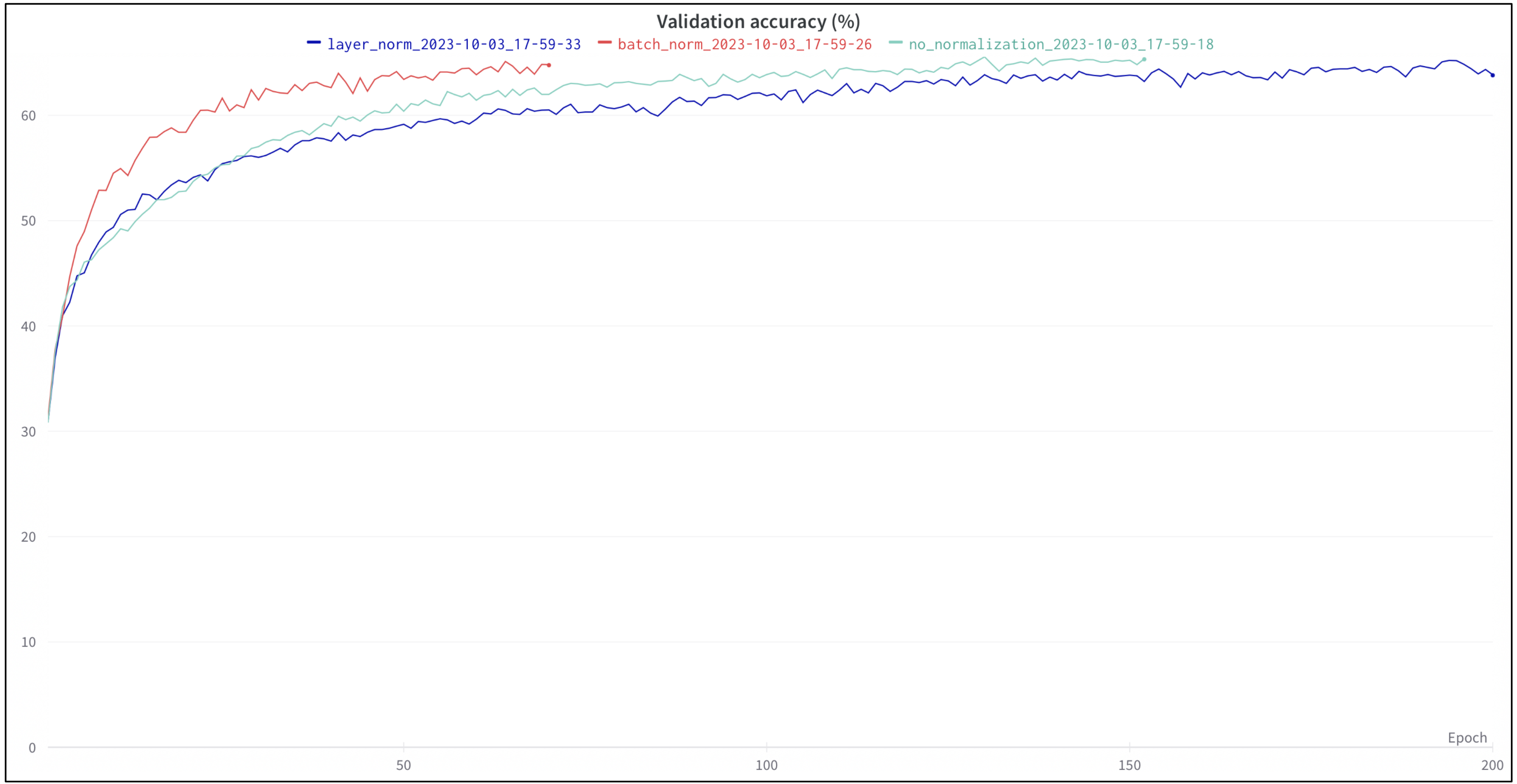
- python _01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py --wandb -v 1 -o 3 -w 0.002 --dropout -n 1

– Layer Normalization

- python _01_code/_08_diverse_techniques/f_cifar10_train_cnn_with_normalization.py --wandb -v 1 -o 3 -w 0.002 --dropout -n 2

Normalization with PyTorch

- CIFAR10 + CNN + Adam Optimizer + Weight Decay 0.002 + Dropout + [No Normalization, Batch Normalization, Layer Normalization]
  - https://wandb.ai/link-koreatech/cnn_cifar10_with_normalization/workspace?workspace=user-link-koreatech



Data Augmentation

Data Augmentation

◈**What is Data Augmentation?**

- A technique of artificially increasing the training set by creating modified copies of a dataset using existing data

◈**Augmented (증강) vs. Synthetic (합성) data**

- Augmented data is driven from original data with some minor changes
  - Augmentation techniques are not limited to images
  - You can augment audio, video, text, and other types of data too
- Synthetic data is generated artificially without using the original dataset
  - GANs (Generative Adversarial Networks) are used to generate synthetic data

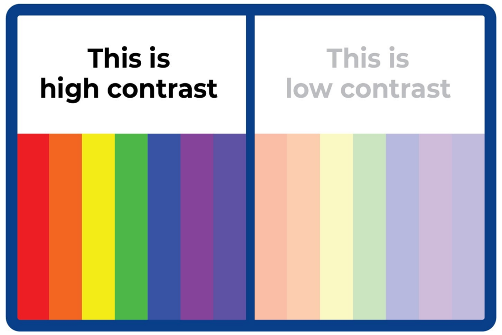
Data Augmentation

◈Data Augmentation Techniques

– Audio Data Augmentation
  - Noise injection: add gaussian or random noise to the audio dataset to improve the model performance
  - Shifting: shift 'some part of audio' left or right with random seconds
  - Changing the speed: stretches times series by a fixed rate
  - Changing the pitch: randomly change the pitch of the audio

– Text Data Augmentation
  - Word replacement: replace words with synonyms (동의어)
  - Text Paraphrasing: Rewriting sentences to express the same meaning but with different words and phrasing
  - Misspelling and Typo Injection: Introducing intentional misspellings and typographical errors in the text

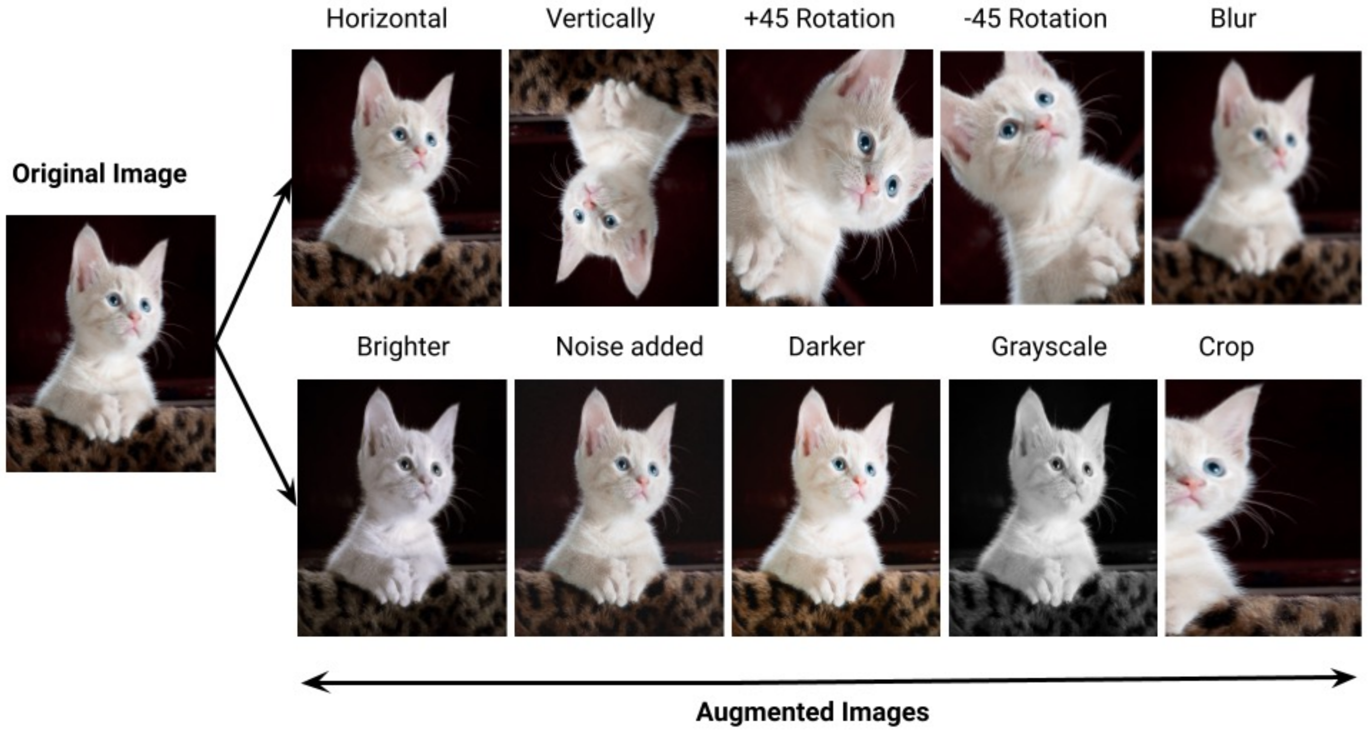
Data Augmentation



◈Data Augmentation Techniques

– Image Augmentation

- Geometric transformations
  - ➢randomly flip, crop, rotate, stretch, and zoom images
  - ➢You need to be careful about applying multiple transformations on the same images, as this can reduce model performance
- Color space transformations
  - ➢randomly change RGB color channels, contrast, and brightness
- Kernel filters
  - ➢randomly change the sharpness or blurring of the image
- Random erasing
  - ➢delete some part of the initial image
- Mixing images
  - ➢blending and mixing multiple images

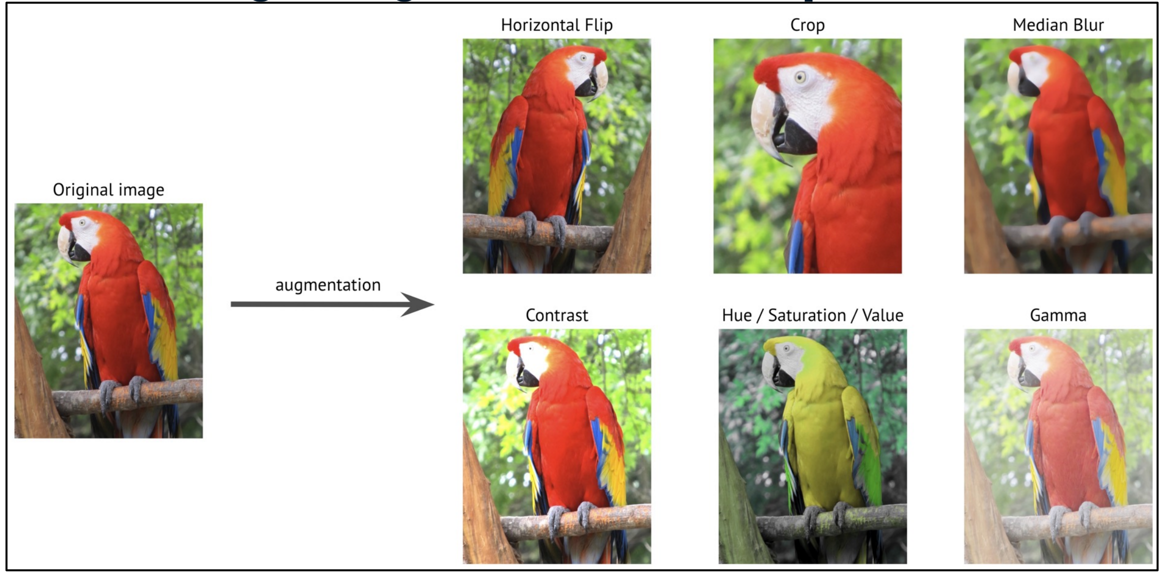


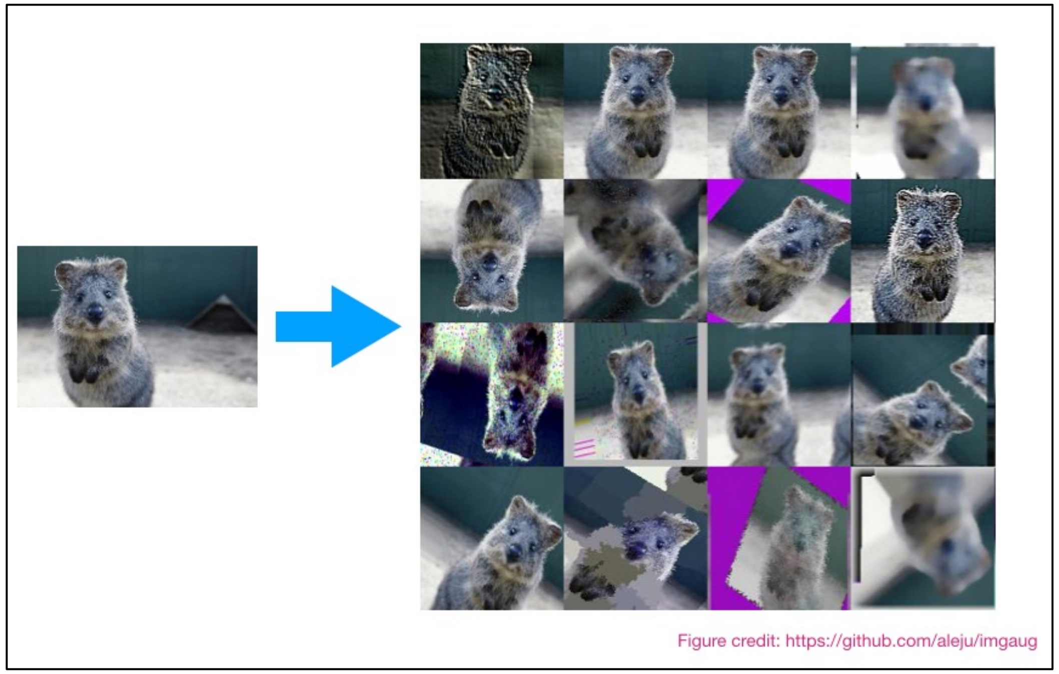
[Reference] https://giggster.com/guide/basics/hue-saturation-lightness/

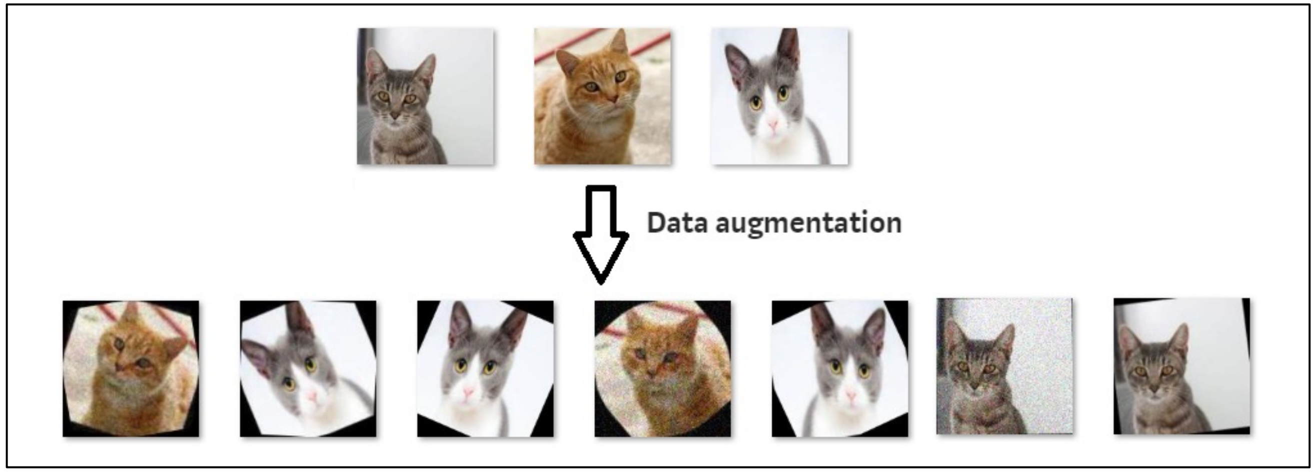
Data Augmentation

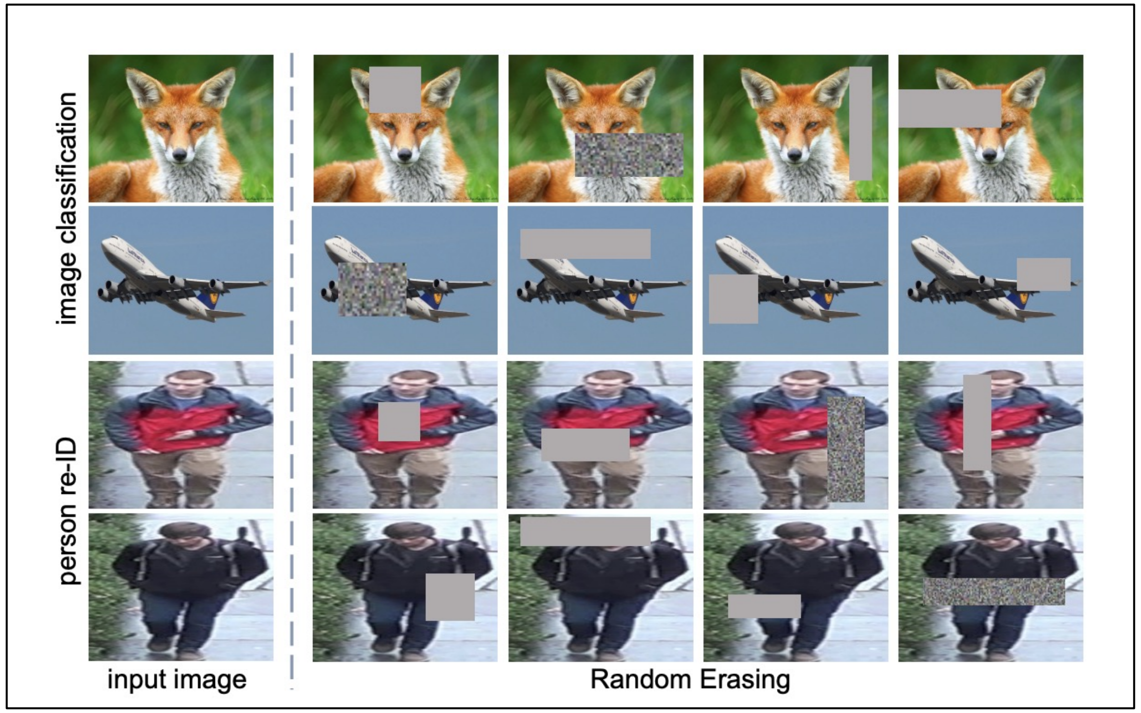
◇Data Augmentation Techniques

– More Image Augmentation Examples









Data Augmentation

◈When Should You Use Data Augmentation?

- To prevent models from overfitting
- The initial training set is too small
- To improve the model accuracy
- To reduce the operational cost of labeling and cleaning the raw dataset

◈Limitations of Data Augmentation

- Some augmentation techniques can lead to a loss of data quality
  - For example, aggressive cropping or resizing in image data augmentation may result in the loss of critical details, making it harder for the model to learn
- Data augmentation can significantly increase the computational cost of training a model
- Finding an effective data augmentation approach can be challenging

Data Augmentation with Pytorch

◈Image Transforms with Pytorch

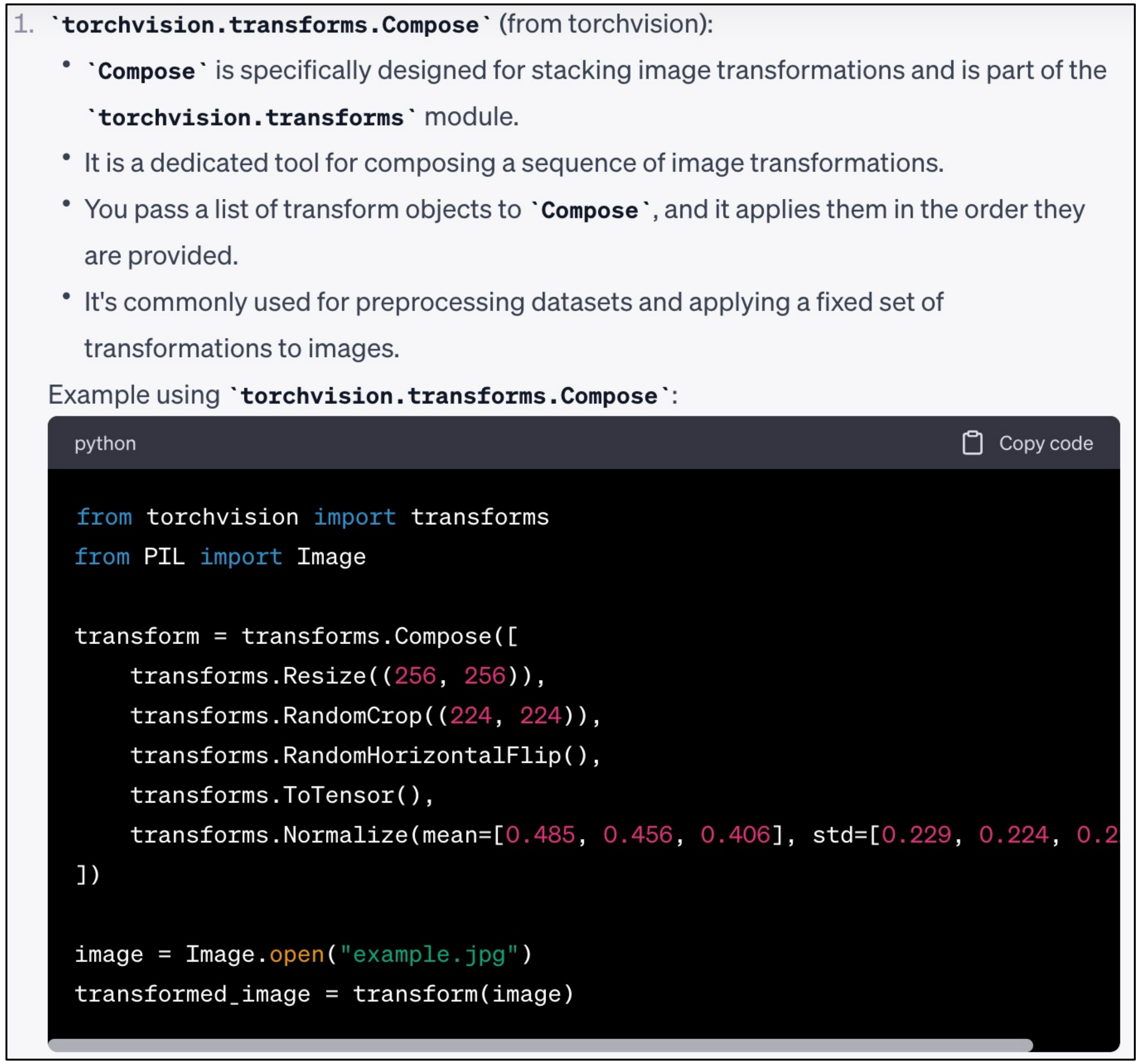
– https://pytorch.org/vision/0.16/transforms.html

◈Image Transforms with Pytorch (한글 설명)

– https://teddylee777.github.io/pytorch/pytorch-image-transforms/
– https://blog.joonas.io/193

Data Augmentation with Pytorch

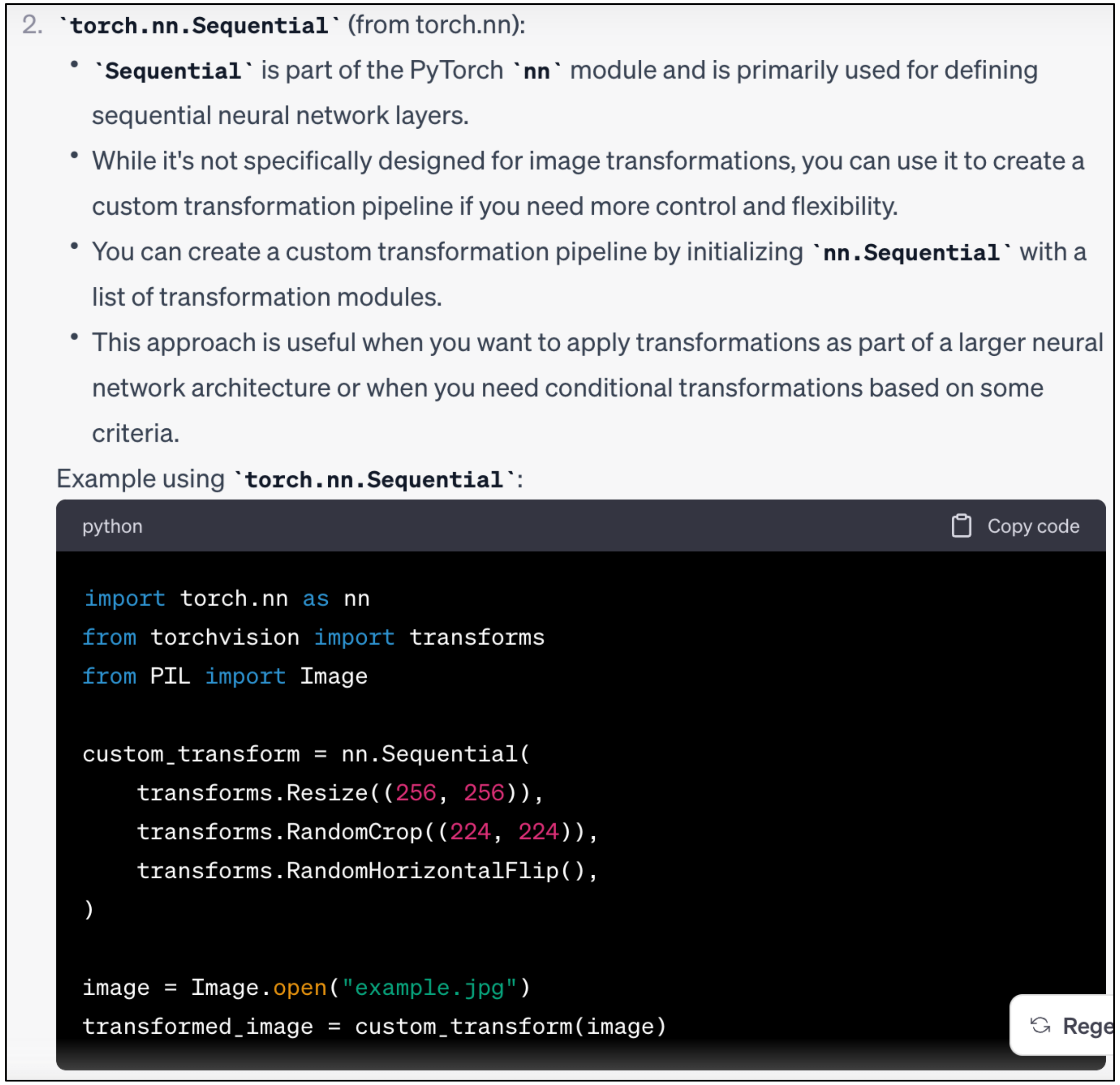
◈Compose vs. Sequential for Pytorch image transforms



1. `torchvision.transforms.Compose` (from torchvision):
   - `Compose` is specifically designed for stacking image transformations and is part of the `torchvision.transforms` module.
   - It is a dedicated tool for composing a sequence of image transformations.
   - You pass a list of transform objects to `Compose`, and it applies them in the order they are provided.
   - It's commonly used for preprocessing datasets and applying a fixed set of transformations to images.

Example using `torchvision.transforms.Compose`:

```python
from torchvision import transforms
from PIL import Image

transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomCrop((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.2
])

image = Image.open("example.jpg")
transformed_image = transform(image)
```



2. `torch.nn.Sequential` (from torch.nn):
   - `Sequential` is part of the PyTorch `nn` module and is primarily used for defining sequential neural network layers.
   - While it's not specifically designed for image transformations, you can use it to create a custom transformation pipeline if you need more control and flexibility.
   - You can create a custom transformation pipeline by initializing `nn.Sequential` with a list of transformation modules.
   - This approach is useful when you want to apply transformations as part of a larger neural network architecture or when you need conditional transformations based on some criteria.

Example using `torch.nn.Sequential`:

```python
import torch.nn as nn
from torchvision import transforms
from PIL import Image

custom_transform = nn.Sequential(
    transforms.Resize((256, 256)),
    transforms.RandomCrop((224, 224)),
    transforms.RandomHorizontalFlip(),
)

image = Image.open("example.jpg")
transformed_image = custom_transform(image)
```

link_dl/_01_code/_08_diverse_techniques/g_cifar10_train_cnn_with_image_augmentation_and_batch_normalization.py

Image Augmentation with PyTorch

◈Image Augmentation (1/2)

```
def get_augmented_cifar10_data():
  data_path = os.path.join(BASE_PATH, "_00_data", "i_cifar10")

  cifar10_train = datasets.CIFAR10(data_path, train=True, download=True, transform=transforms.ToTensor())

  cifar10_train, cifar10_validation = random_split(cifar10_train, [45_000, 5_000])

  cifar10_train_transforms = nn.Sequential(
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop([32, 32], padding=4),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
  )

  transformed_train_data = []
  for image, label in cifar10_train:
    transformed_image = cifar10_train_transforms(image)
    transformed_train_data.append((transformed_image, label))
```

link_dl/_01_code/_08_diverse_techniques/g_cifar10_train_cnn_with_image_augmentation_and_batch_normalization.py

Image Augmentation with PyTorch

◈ Image Augmentation (1/2)

```
def get_augmented_cifar10_data():
  ...

  cifar10_train = ConcatDataset([cifar10_train, transformed_train_data])

  print("Num Train Samples: ", len(cifar10_train))           # >>> Num Train Samples:  90000
  print("Num Validation Samples: ", len(cifar10_validation))  # >>> Num Validation Samples:  5000

  num_data_loading_workers = get_num_cpu_cores() if is_linux() or is_windows() else 0
  print("Number of Data Loading Workers:", num_data_loading_workers)
  # >>> Number of Data Loading Workers: 0

  train_data_loader = DataLoader(
    dataset=cifar10_train, batch_size=wandb.config.batch_size, shuffle=True,
    pin_memory=True, num_workers=num_data_loading_workers
  )
```

link_dl/_01_code/_08_diverse_techniques/g_cifar10_train_cnn_with_image_augmentation_and_batch_normalization.py

Image Augmentation with PyTorch

◈ Image Augmentation (1/2)

```
def get_augmented_cifar10_data():
  ...

  validation_data_loader = DataLoader(
    dataset=cifar10_validation, batch_size=wandb.config.batch_size,
    pin_memory=True, num_workers=num_data_loading_workers
  )

  cifar10_transforms = nn.Sequential(
    transforms.ConvertImageDtype(torch.float),
    transforms.Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.2470, 0.2435, 0.2616)),
  )

  return train_data_loader, validation_data_loader, cifar10_transforms
```

Image Augmentation with PyTorch

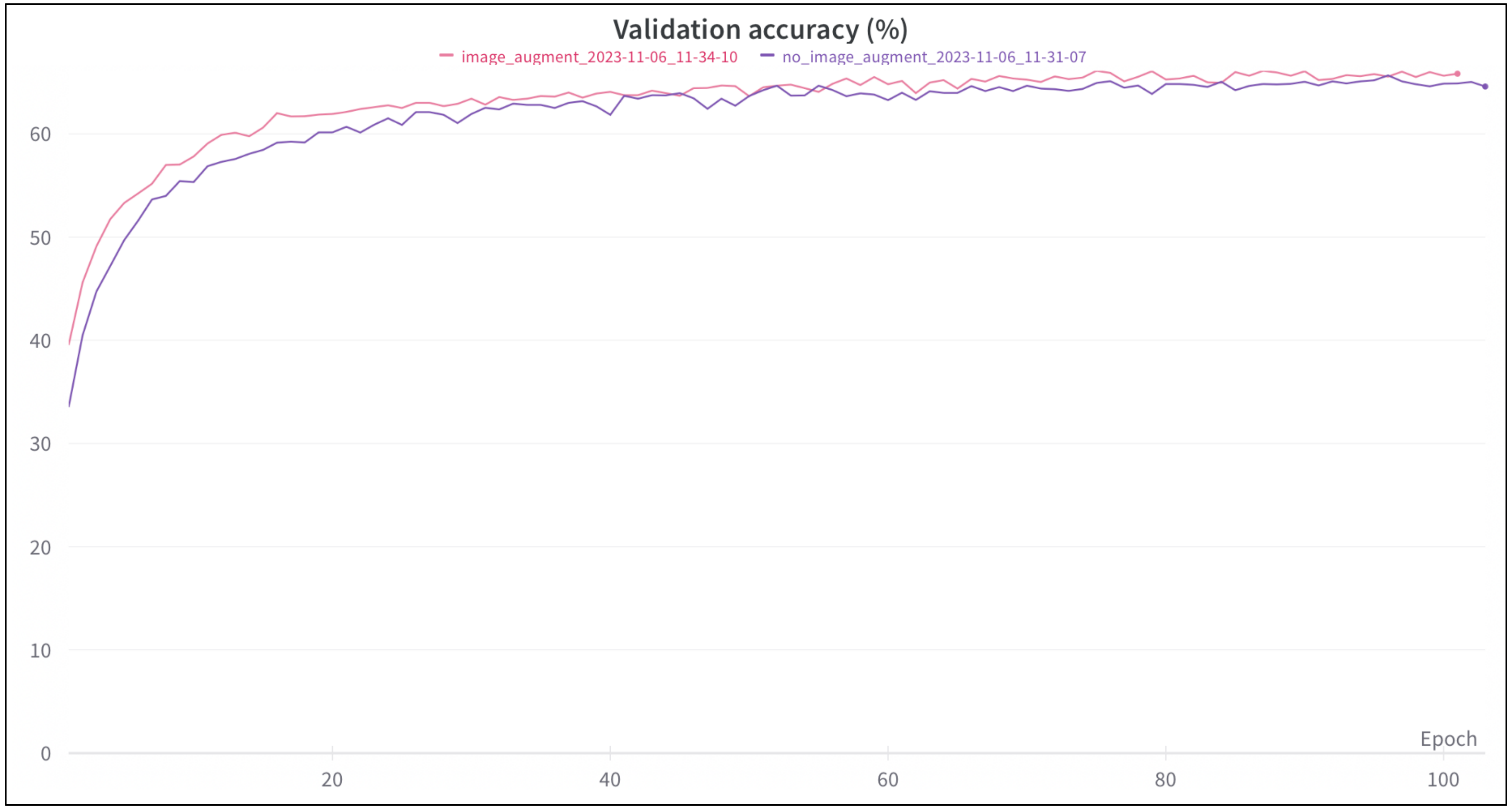
◈ CIFAR10 + CNN + Adam Optimizer + Weight Decay 0.002 + Dropout + Batch Normalization + [No Image Augmentation, Image Augmentation]

- No Image Augmentation
  - python _01_code/_08_diverse_techniques/g_cifar10_train_cnn_with_image_augmentation_and_batch_normalization.py --wandb --dropout --no-augment -v 1 -o 3 -w 0.002 -n 1

- Image Augmentation
  - python _01_code/_08_diverse_techniques/g_cifar10_train_cnn_with_image_augmentation_and_batch_normalization.py --wandb --dropout --augment -v 1 -o 3 -w 0.002 -n 1

Normalization with PyTorch

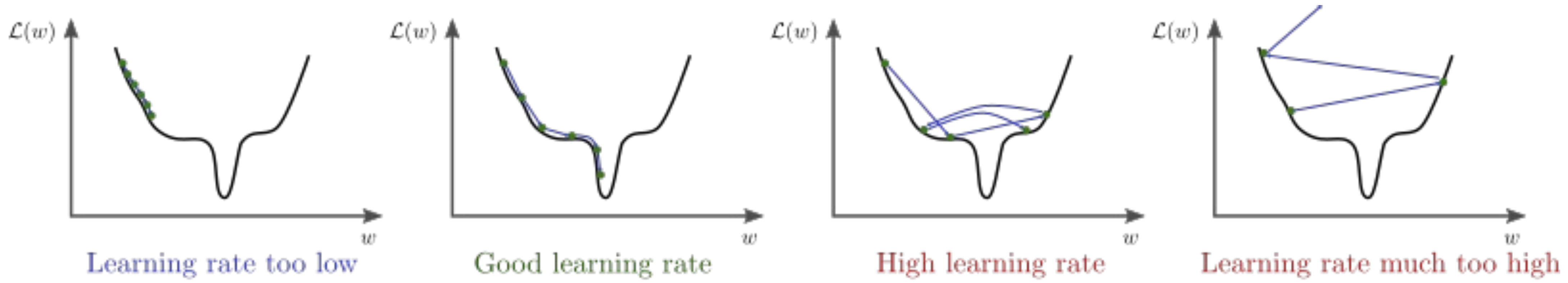
- CIFAR10 + CNN + Adam Optimizer + Weight Decay 0.002 + Dropout + Batch Normalization + [No Image Augmentation, Image Augmentation]
  - https://wandb.ai/link-koreatech/cnn_cifar10_with_image_augment/workspace?workspace=user-link-koreatech

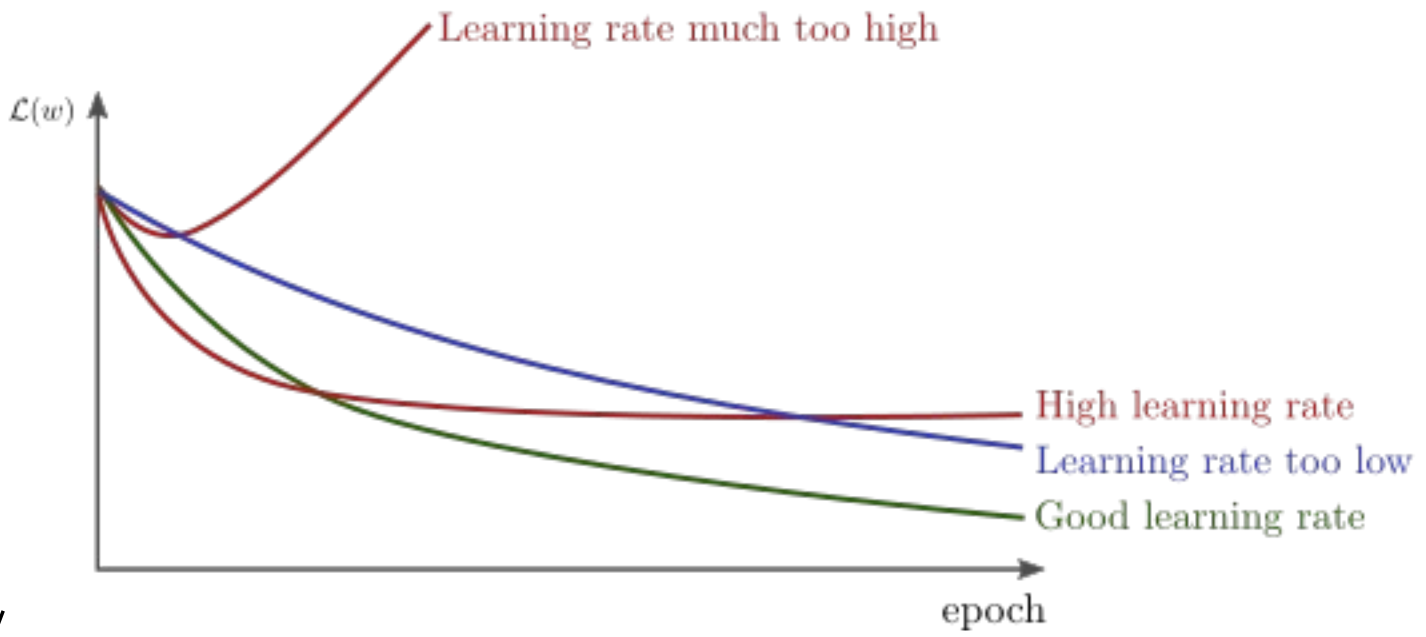


Learning Rate Decay

Learning Rate Decay

◈ Learning Rate

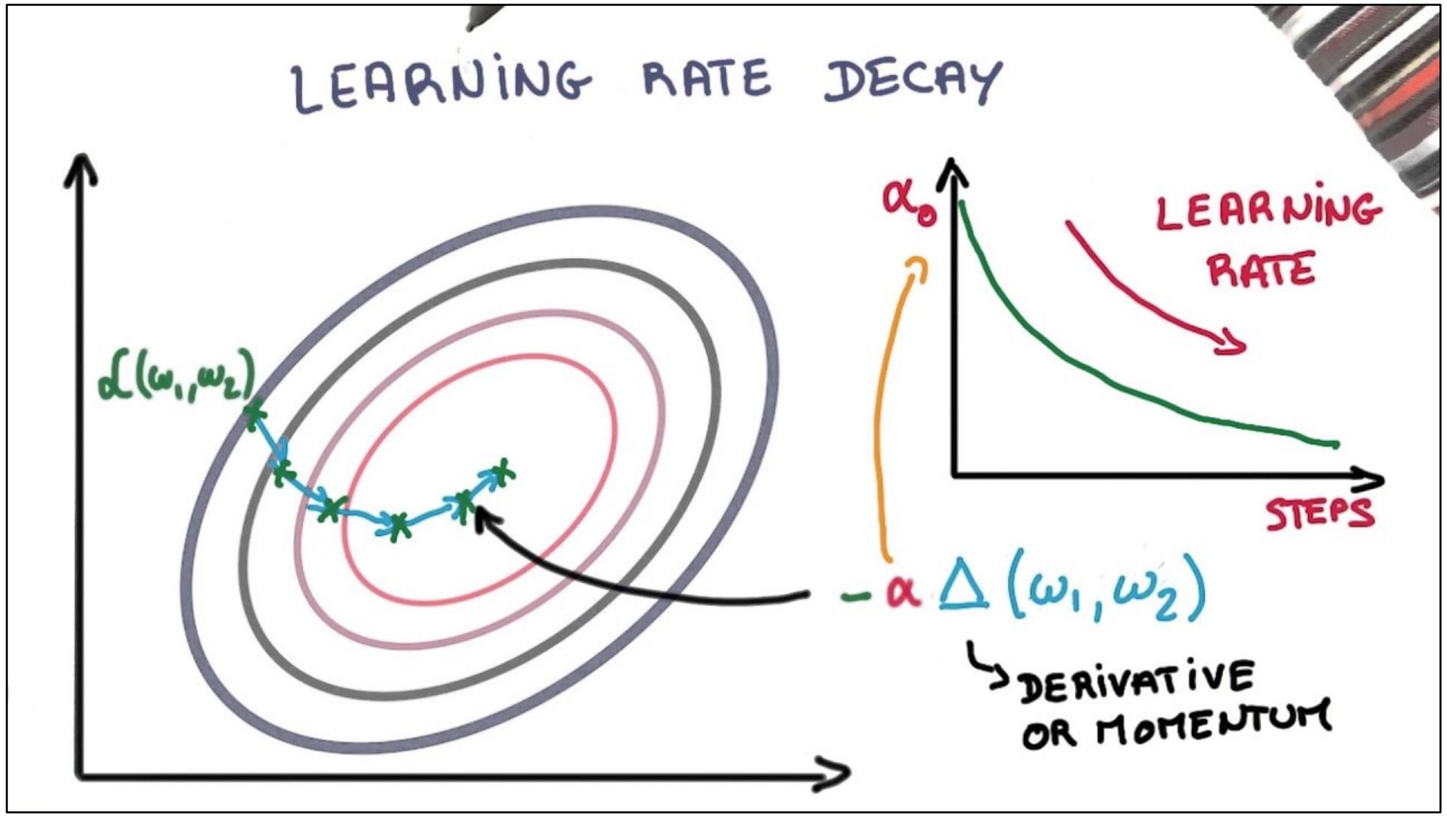




[Source] https://www.bdhammel.com/learning-rates/

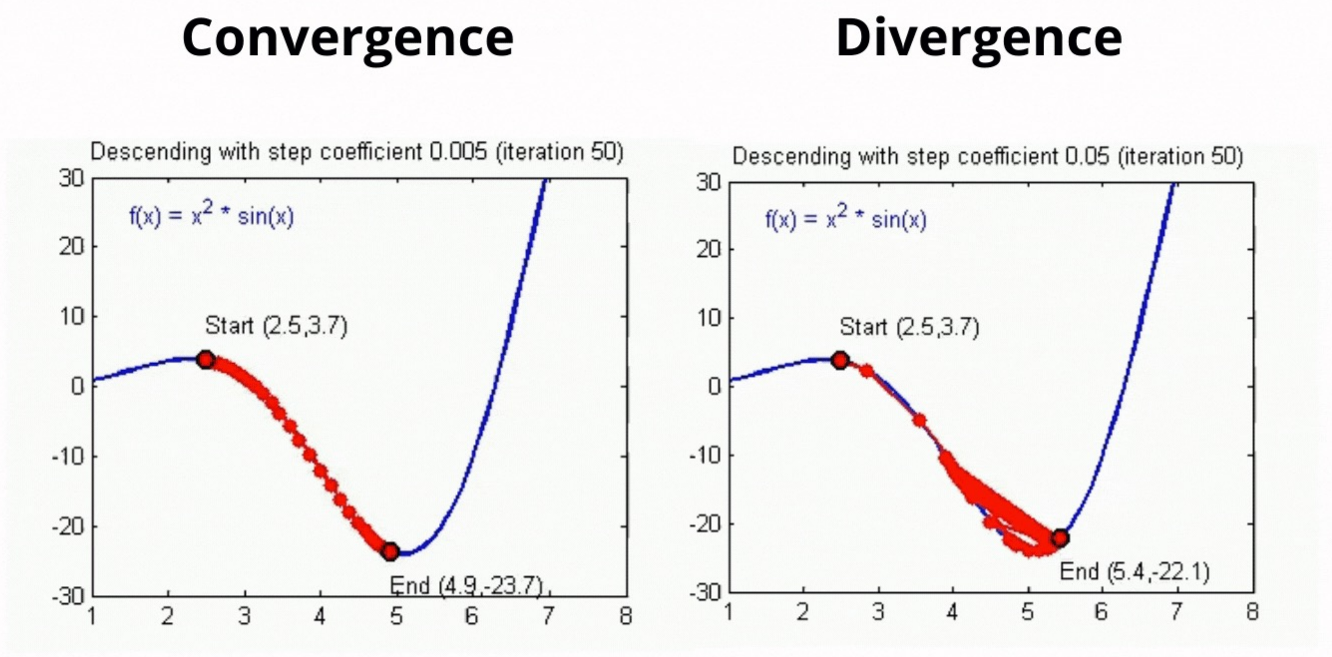
Learning Rate Decay

- **Learning Rate Decay**
  - It starts training the network with a large learning rate and then slowly reducing/decaying it until local minima is obtained
  - It is empirically observed to help both optimization and generalization



Learning Rate Decay

- Why it is useful?
  - Decreasing the learning rate helps to prevent parameters oscillations (or overshooting), leading to a more stable convergence
  - Gradual reduction in the learning rate allows the model to refine its learned patterns and not overfit to the training data



Learning Rate Decay

◈ Methods of Learning Rate Decay

– Pytorch supports six methods of learning rate decays

lr_scheduler.StepLR	Decays the learning rate of each parameter group by gamma every step_size epochs.
lr_scheduler.MultiStepLR	Decays the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestones.
lr_scheduler.ConstantLR	Decays the learning rate of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: total_iters.
lr_scheduler.LinearLR	Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: total_iters.
lr_scheduler.ExponentialLR	Decays the learning rate of each parameter group by gamma every epoch.
lr_scheduler.PolynomialLR	Decays the learning rate of each parameter group using a polynomial function in the given total_iters.

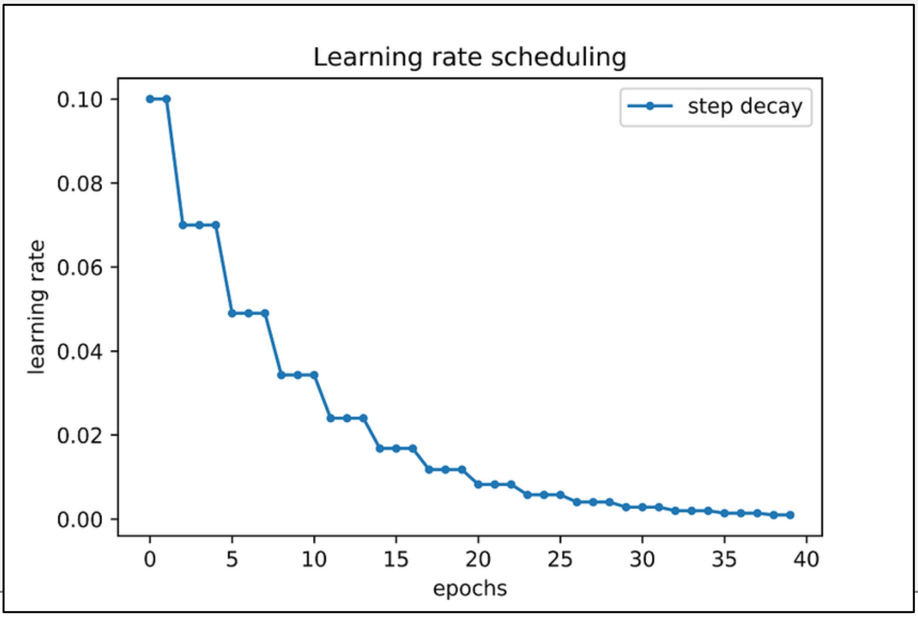
Learning Rate Decay

◈Methods of Learning Rate Decay

– Pytorch supports six methods of learning rate decays

```
import torch.optim as optim
from torch.optim import lr_scheduler
optimizer = optim.SGD(model.parameters(), lr=0.1)

# >>> Decay LR by a factor of 0.7 every 3 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.7)
...
…
for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

$$lr_{\text{epoch}} = \begin{cases} Gamma * lr_{\text{epoch - 1}}, & \text{if epoch \% step\_size} = 0 \\ lr_{\text{epoch - 1}}, & \text{otherwise} \end{cases}$$


Learning rate scheduling
step decay
learning rate
epochs