

UNIVERSITY OF
TORONTO

Applied Deep Learning

Lecture 7. *Normalization and Residual Learning*

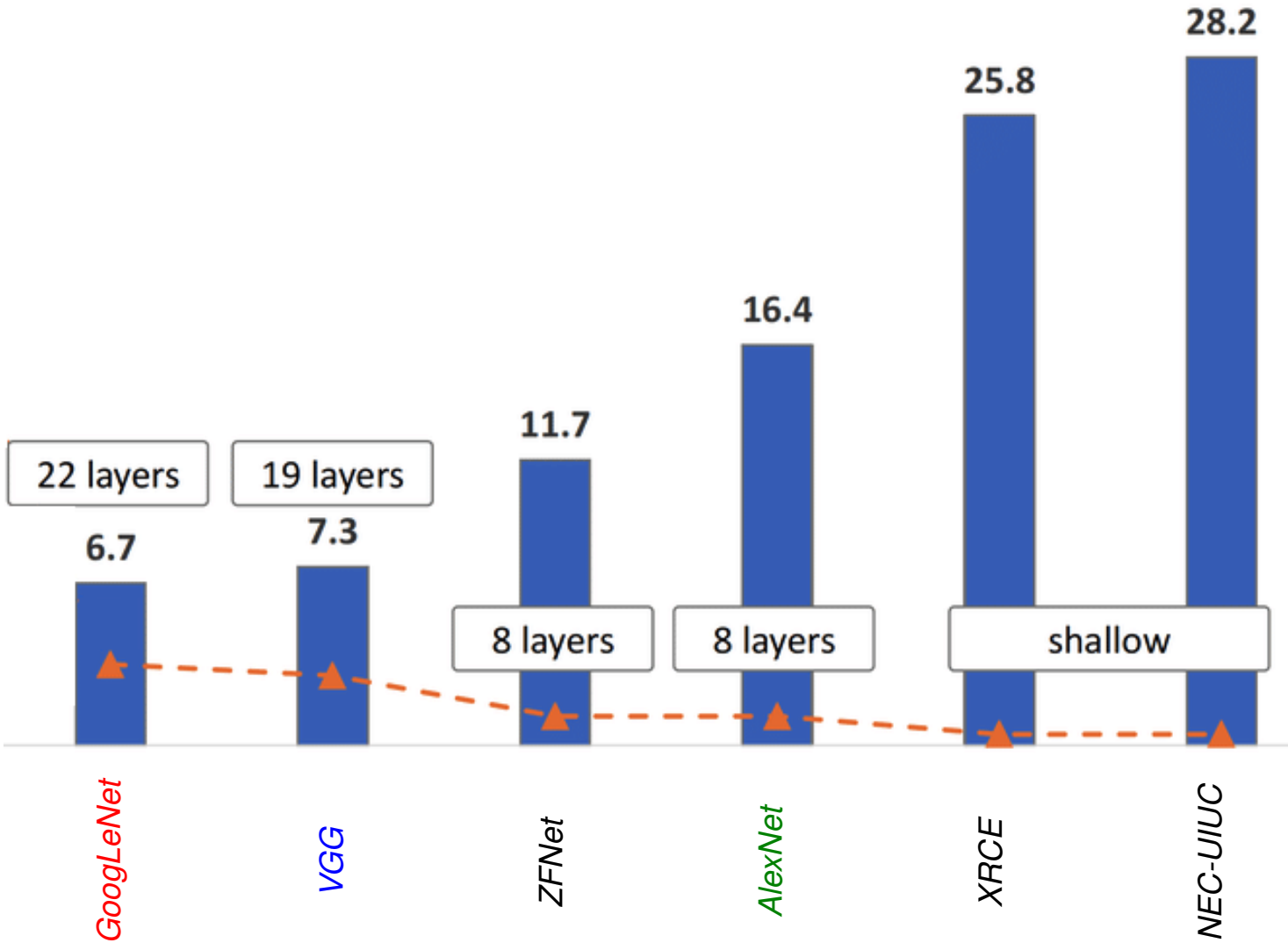
Ali Bereyhi
ECE Department, University of Toronto
Fall 2026



UNIVERSITY OF
TORONTO

Lecture Overview

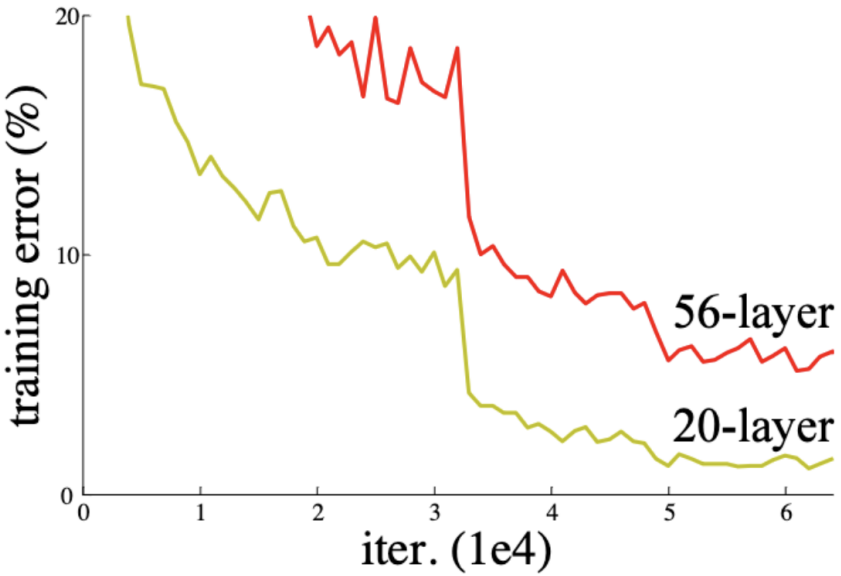
Depth vs Accuracy: *ILSVRC*

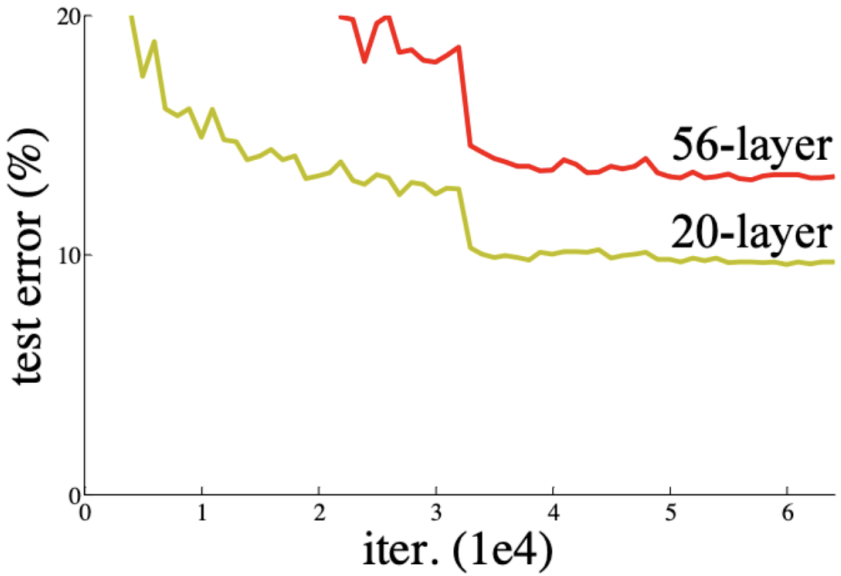


A Hurdle in Going Deeper

Studies showed that *much deeper CNNs start to perform worse!*

? *Overfitting was ruled out by Microsoft Research Lab*

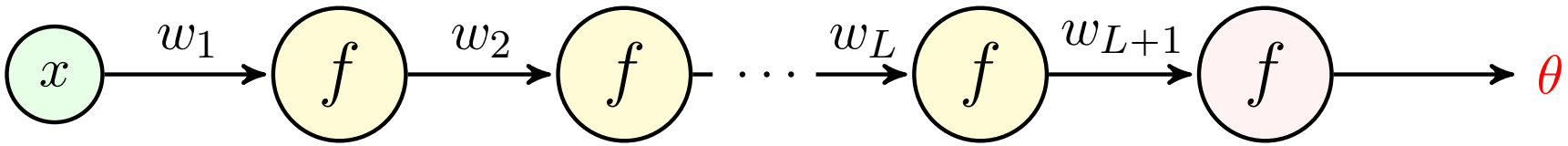




! *If it's coming from overfitting we should see better training and worst test risk!*

↳ *It's indeed vanishing gradient behaviour of deep NNs*

Vanishing and Exploding Gradients



Derivative of loss with respect to the first weight w_1 is

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} = \overleftarrow{\theta}\ \underbrace{x\dot{f}\left(w_1 x\right)\prod_{\ell=2}^{L+1} w_\ell \dot{f}\left(w_\ell y_{\ell-1}\right)}_{\text{accumulated in Backpropagation}}$$

Lecture 7. *Normalization and Residual Learning*

Today, we learn two major techniques that enable us go very deep

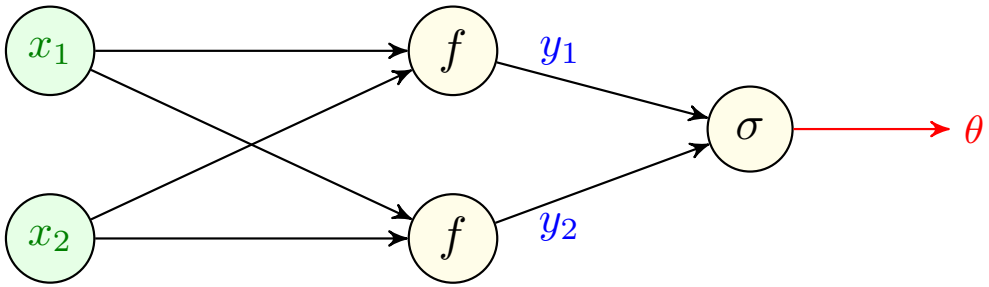
1. Normalization
   - *Input normalization*
   - *Batch normalization*
   - *Layer normalization*
2. Residual Learning
   - *Skip connection and it's role*
   - *Residual unit*
   - *Residual networks*



UNIVERSITY OF
TORONTO

Normalization

Feature Scaling: *Example*

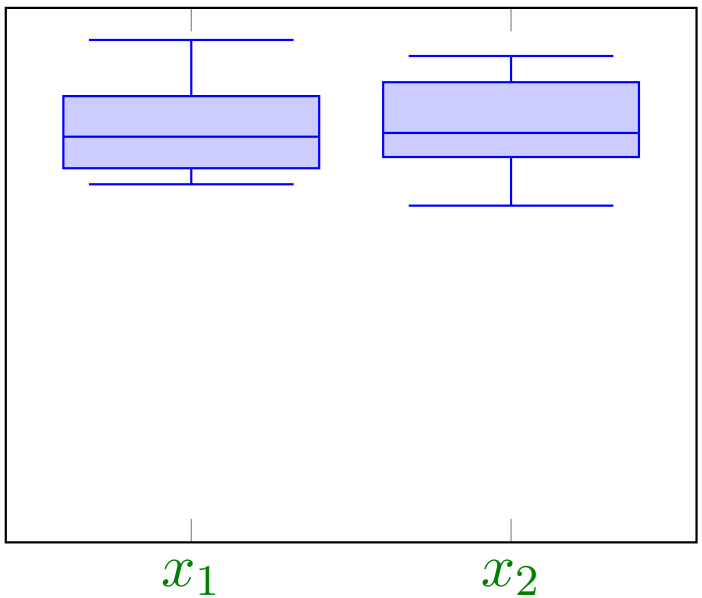


The inputs could be anything, for instance

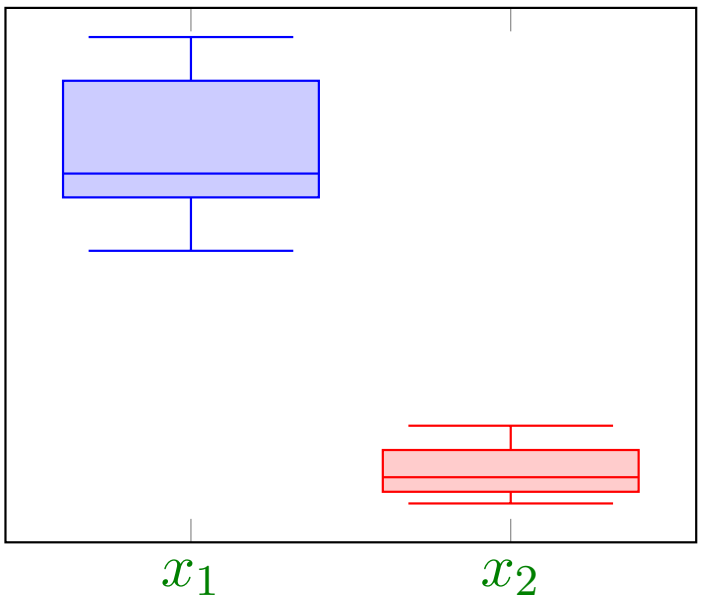
- ***Example A.*** *both are heights in centimeters*
  - ↳ *they are in the same range*
  - ↳ *distributions of variables have close means and variances*
- ***Example B.*** x_1 *is height centimeters and* x_2 *is number of children*
  - ↳ *they are in the widely-different range*
  - ↳ *distributions of variables have strongly-different means and variances*

Feature Scaling: *Example*

Example A



Example B



Feature Scaling

Input data or features at hidden layers can scale in different range

1. *It can deteriorates learning $\rightsquigarrow$ some features dominate due to their scale*
2. *This impacts gradient computations $\rightsquigarrow$ various derivatives scale differently*

Normalization

- ? *It seems we should re-scale them to a common range!*
- That's right! We can normalize features

Normalization

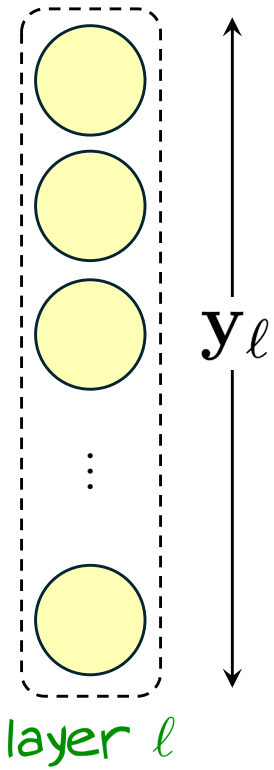
At layer ℓ, we can normalize features $\mathbf{y}_\ell$ before passing it to the next layer as

$$\bar{\mathbf{y}}_\ell = \frac{\mathbf{y}_\ell - \boldsymbol{\mu}_\ell}{\boldsymbol{\sigma}_\ell}$$

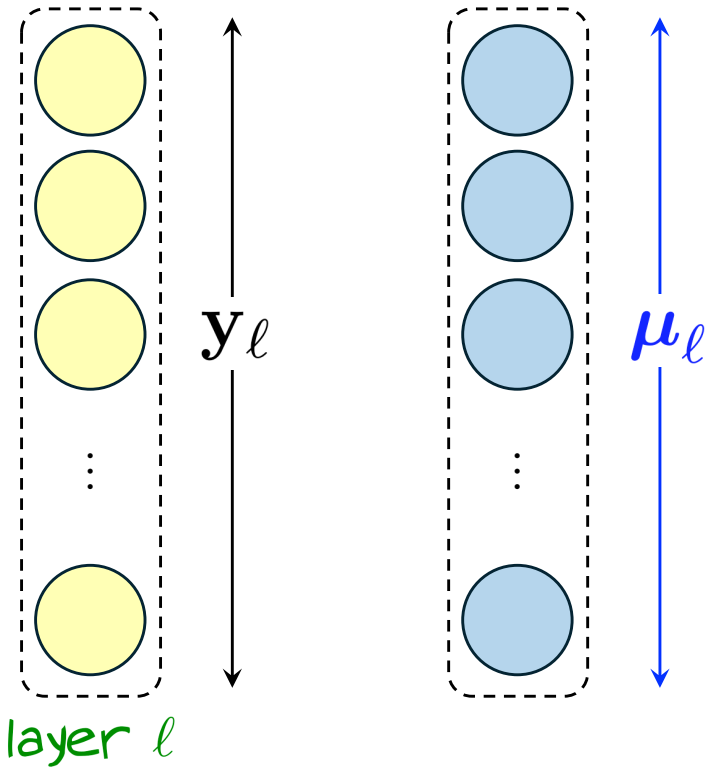
The next layer will perform computations on normalized features $\bar{\mathbf{y}}_\ell$

- ? *How shall we choose $\boldsymbol{\mu}_\ell$ and $\boldsymbol{\sigma}_\ell$?*
- ! We can do it in various forms!

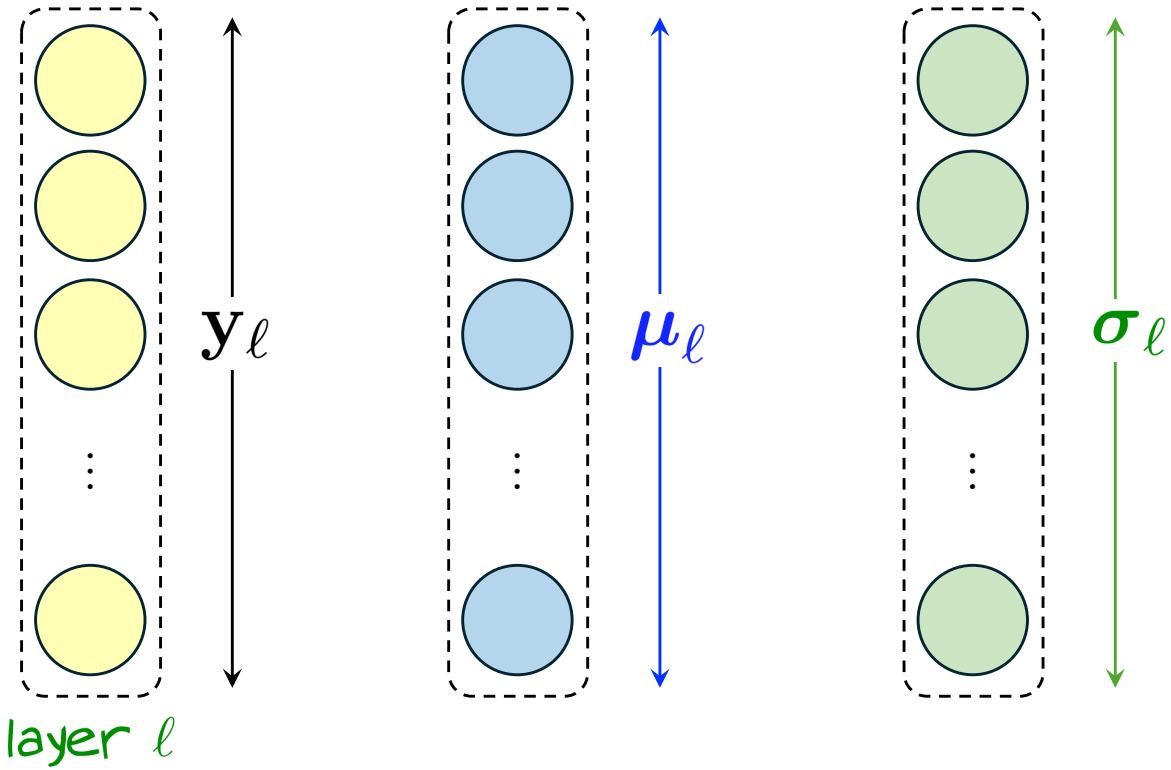
Visualization: *MLPs*



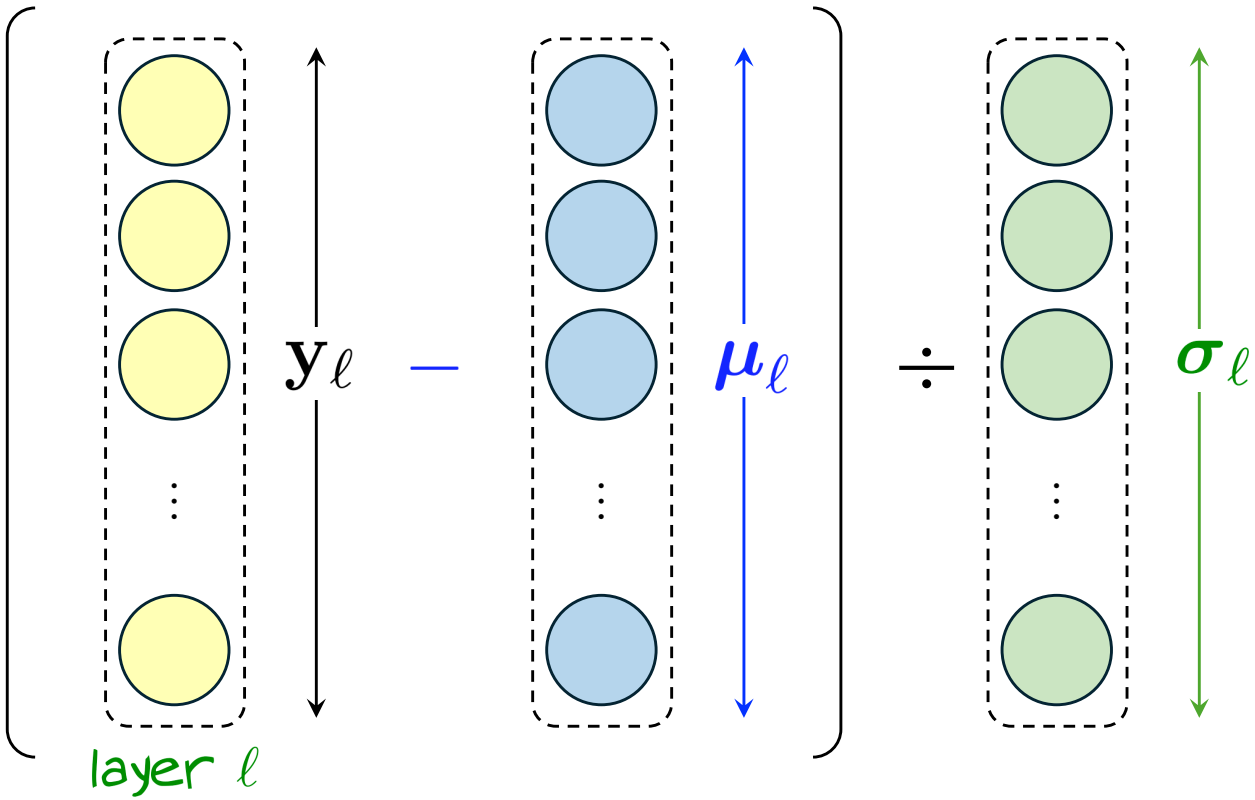
Visualization: *MLPs*



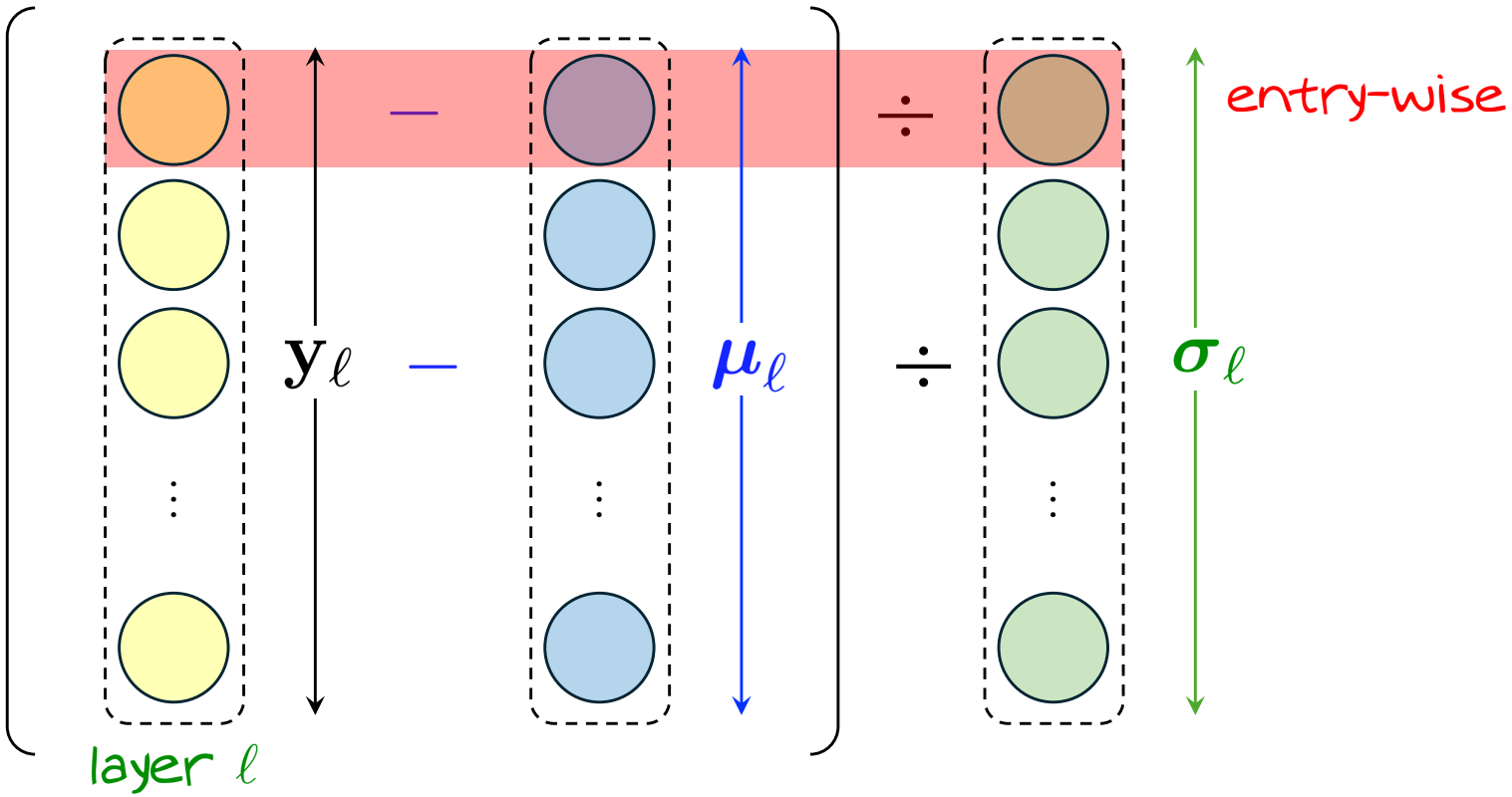
Visualization: *MLPs*



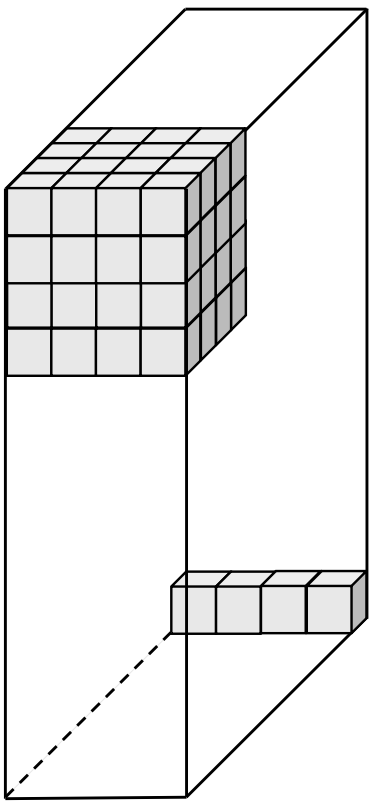
Visualization: *MLPs*



Visualization: *MLPs*

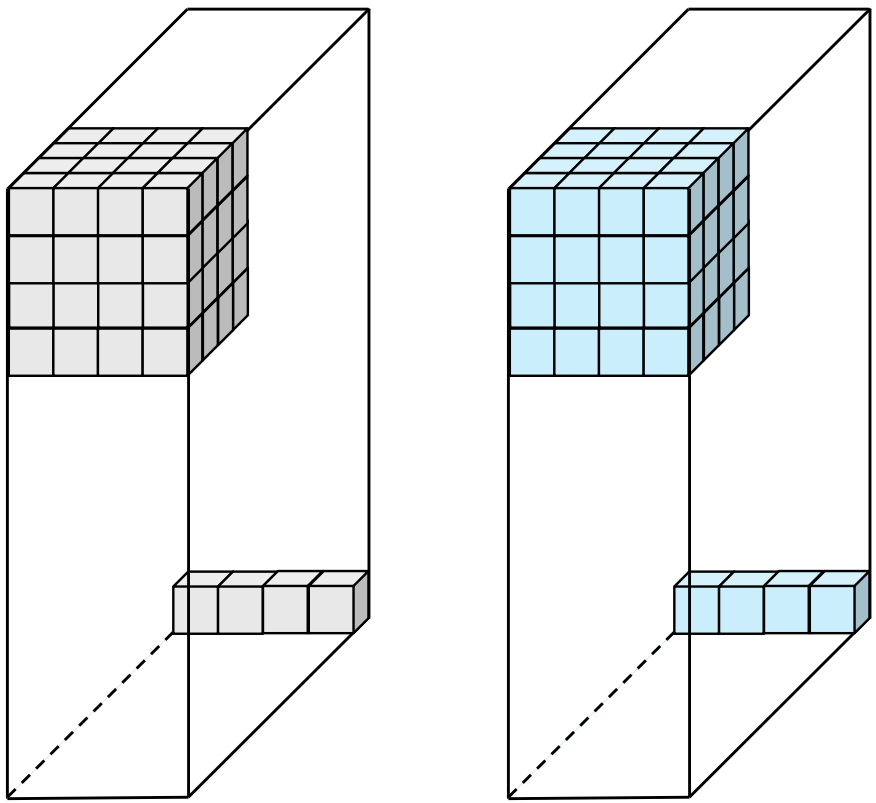


Visualization: *CNNs*



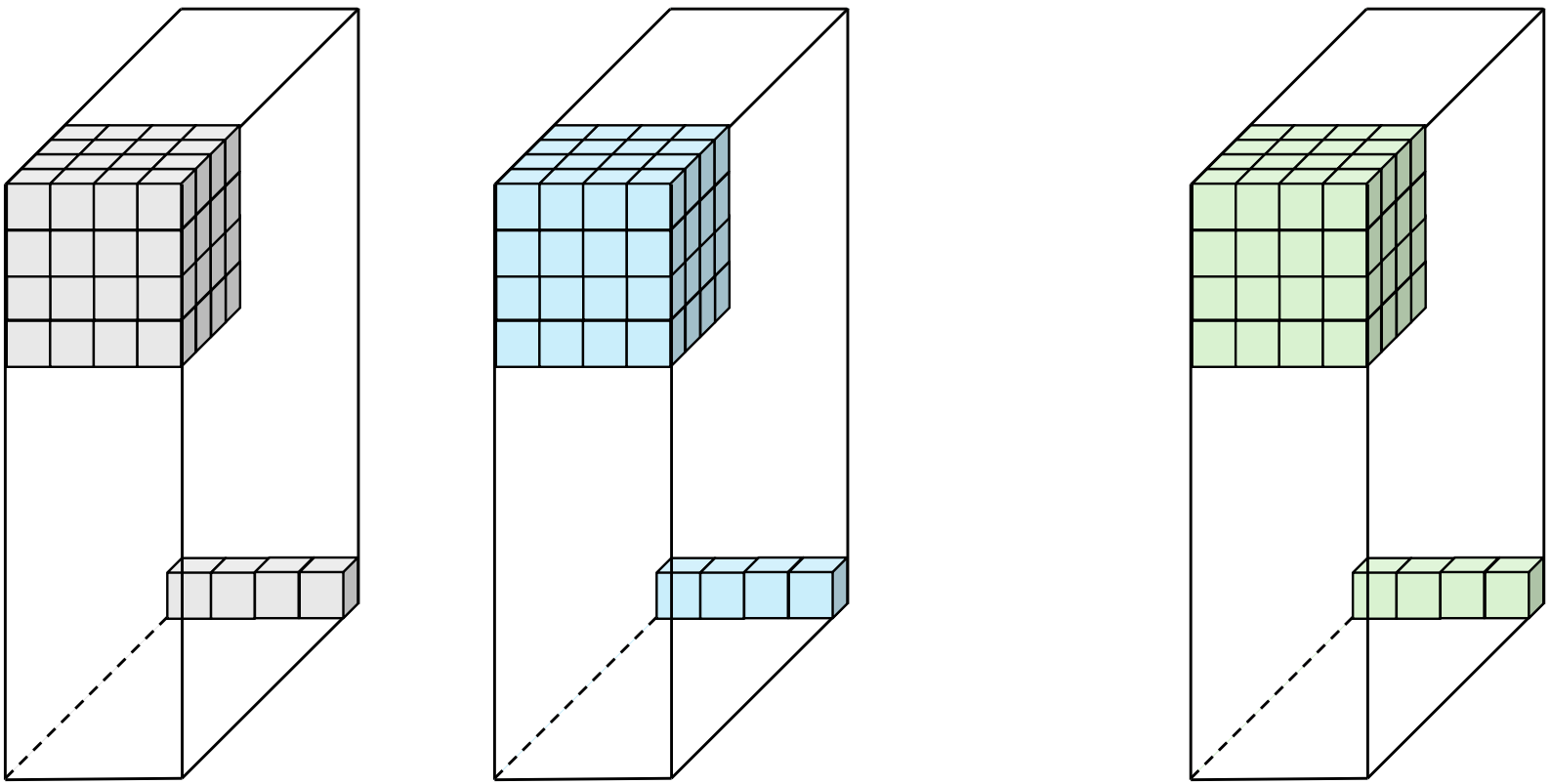
$\mathbf{y}_\ell$

Visualization: *CNNs*



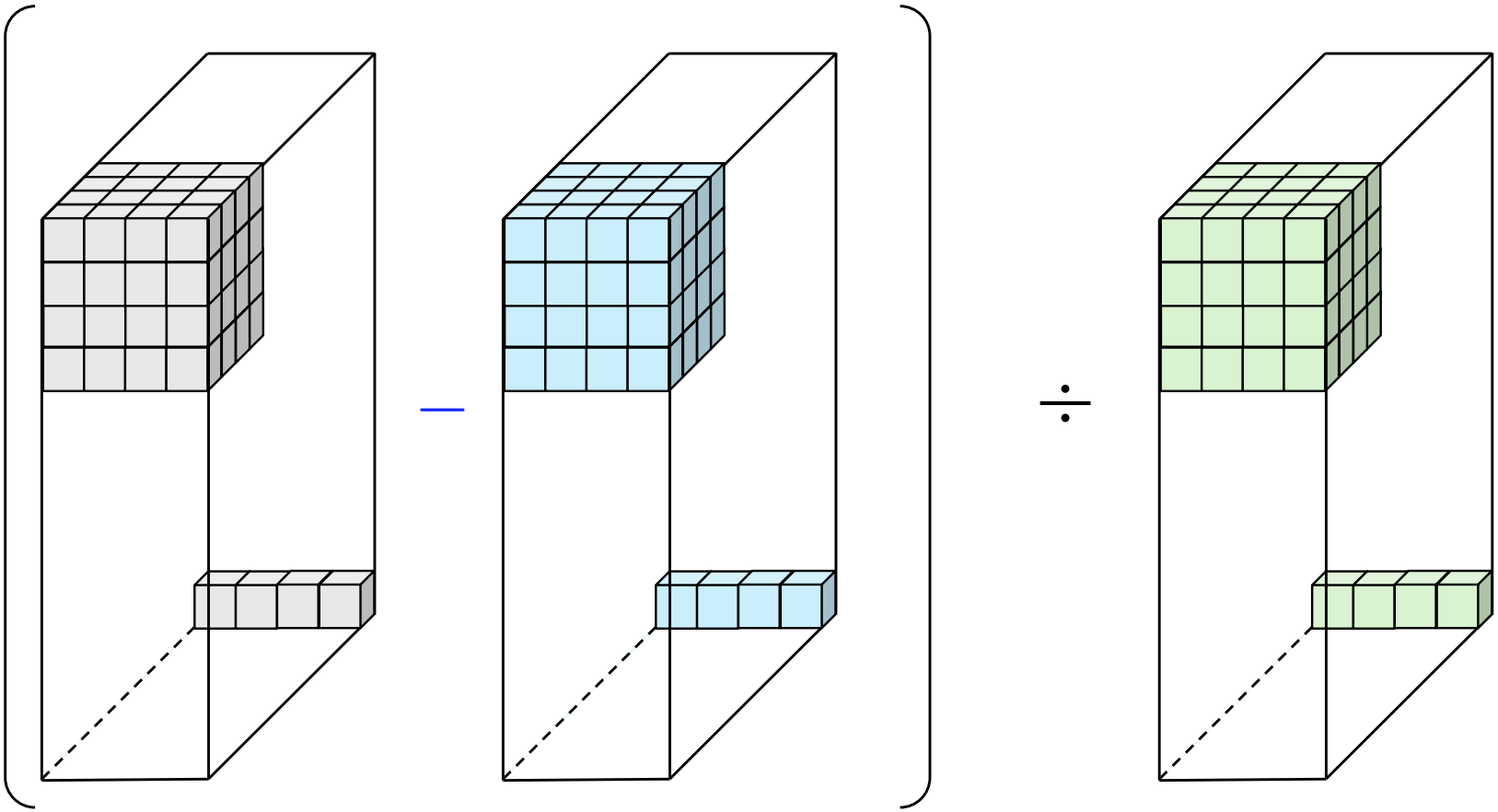
$\mathbf{y}_\ell$ $\quad$ $\boldsymbol{\mu}_\ell$

Visualization: *CNNs*

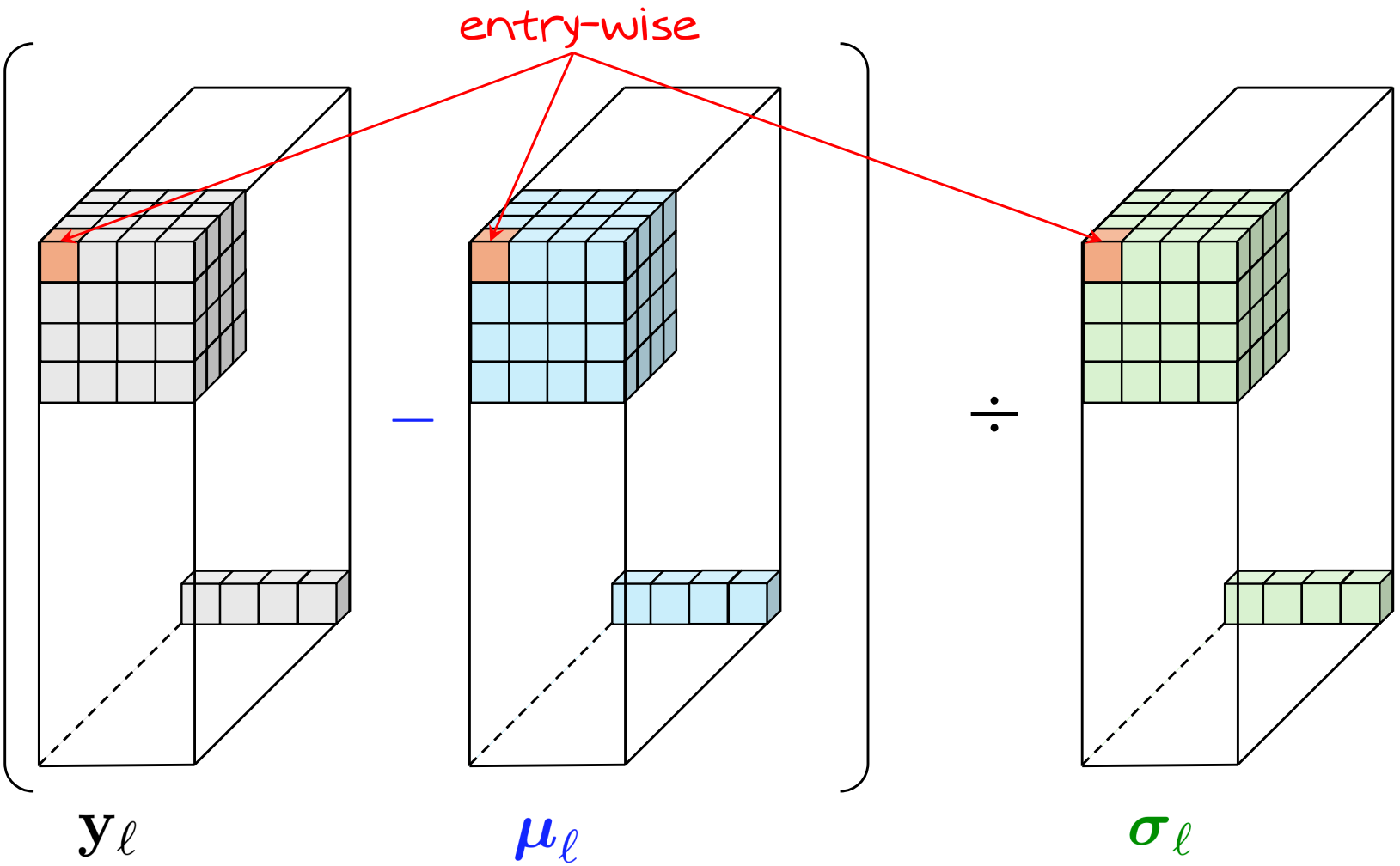


$\mathbf{y}_\ell$ $\quad$ $\boldsymbol{\mu}_\ell$ $\quad$ $\boldsymbol{\sigma}_\ell$

Visualization: *CNNs*



Visualization: *CNNs*



Input Normalization

At the input layer, *we can find* $\boldsymbol{\mu}_0$ *and* $\boldsymbol{\sigma}_0$ *directly from data, i.e., we normalize each data sample as*

$$\bar{\boldsymbol{x}}_b = \frac{\boldsymbol{x}_b - \boldsymbol{\mu}_0}{\boldsymbol{\sigma}_0}$$

for $\boldsymbol{\mu}_0$ *and* $\boldsymbol{\sigma}_0$ *that are computed from the dataset with* I *samples as*

$$\boldsymbol{\mu}_0 = \frac{1}{I}\sum_{b=1}^{I} \boldsymbol{x}_b$$

$$\boldsymbol{\sigma}_0 = \sqrt{\frac{1}{I}\sum_{b=1}^{I} (\boldsymbol{x}_b - \boldsymbol{\mu}_0)^2}$$

Forward Propagation with *Input Normalization*

```
ForwardProp():
```
1: Initiate with $\mathbf{y}_0 = \mathcal{U}\left(\boldsymbol{x}_b | \mathbb{D}\right)$
2: **for** $\ell = 0, \ldots, L$ **do**
3: Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$ `# forward affine`
4: Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$ `# forward activation`
5: **end for**
6: **for** $\ell = 1, \ldots, L+1$ **do**
7: Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
8: **end for**

+ *Shall we do the same for every layer?*
– Well, we could try, *but we end up with computation complexity close to full-batch training!*

Normalization at Hidden Layers

At hidden layers, i.e., $\ell > 1$, *the input depends on the wights and biases of previous layer: for instance, after normalizing input we compute*

$$\mathbf{z}_1 = \mathbf{W}_1 \bar{\mathbf{x}} \rightsquigarrow \mathbf{y}_1 = f_1(\mathbf{z}_1)$$

if we approximate mean and variance with same approach, we should compute

$$\boldsymbol{\mu}_1 = \frac{1}{I} \sum_{b=1}^{I} \mathbf{y}_{1,b}$$

$$\boldsymbol{\sigma}_1 = \sqrt{\frac{1}{I} \sum_{b=1}^{I} (\mathbf{y}_{1,b} - \boldsymbol{\mu}_1)^2}$$

This means after each update of weights at the end of each mini-batch, we should repeat this for the entire training dataset!

Approach I. Batch Normalization

Batch normalization extends the input normalization approach to mini-batches to incorporate it in hidden layers: the idea is simple

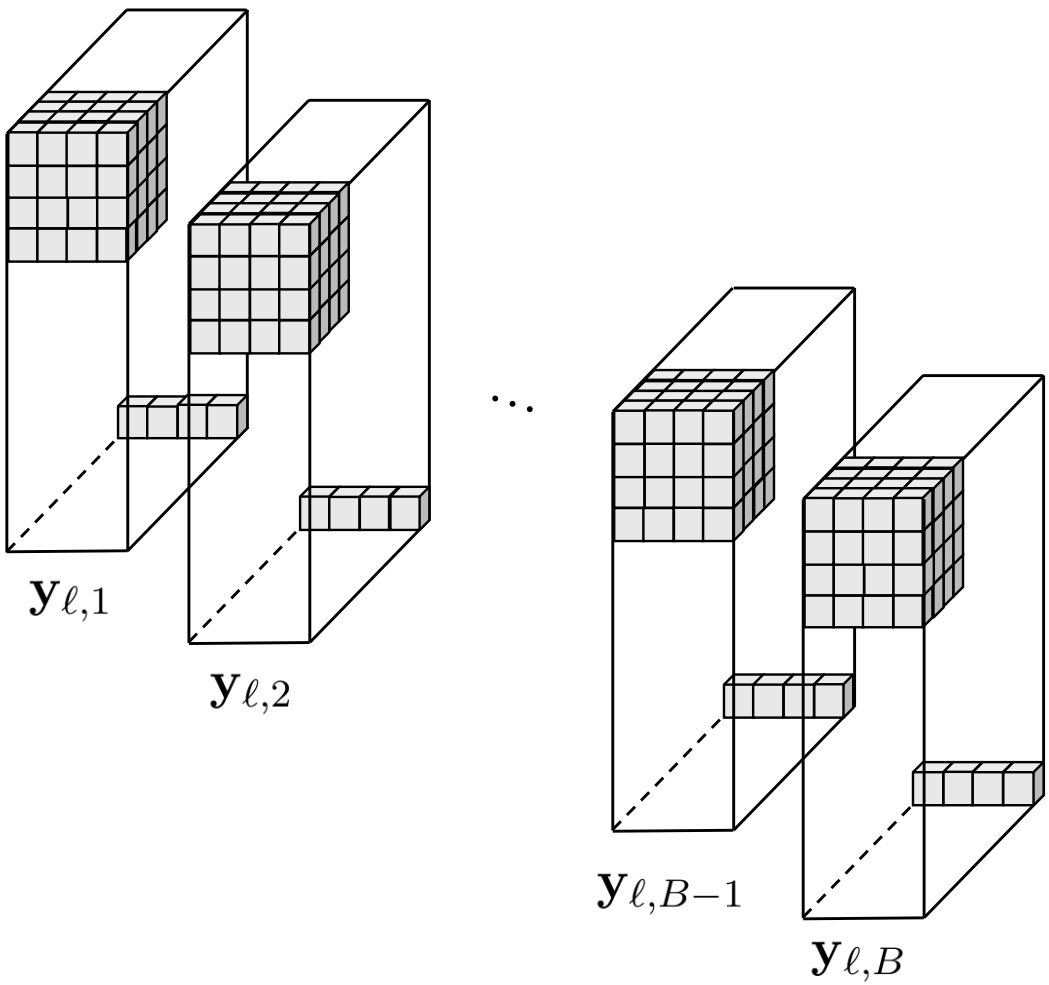
- ✔ *approximate mean and variance using the mini-batch*

$$\boldsymbol{\mu}_\ell = \frac{1}{I}\sum_{b=1}^{B} \mathbf{y}_{\ell,b}$$

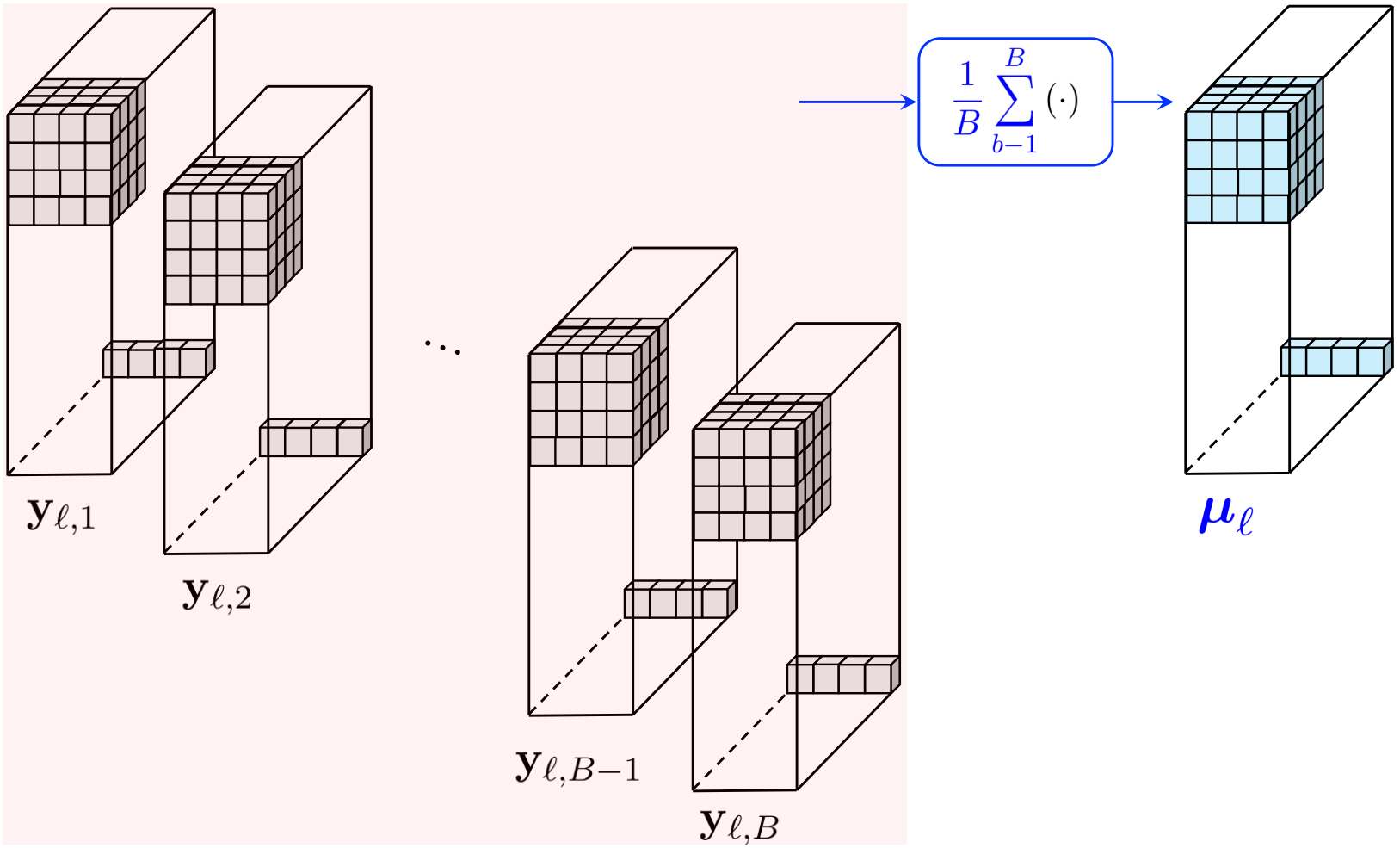
$$\boldsymbol{\sigma}_\ell = \sqrt{\frac{1}{B}\sum_{b=1}^{B} (\mathbf{y}_{\ell,b} - \boldsymbol{\mu}_\ell)^2}$$

- ! *compute derivatives by taking normalization into account*
  - Note than $\boldsymbol{\mu}_\ell$ and $\boldsymbol{\sigma}_\ell$ also depend on weights

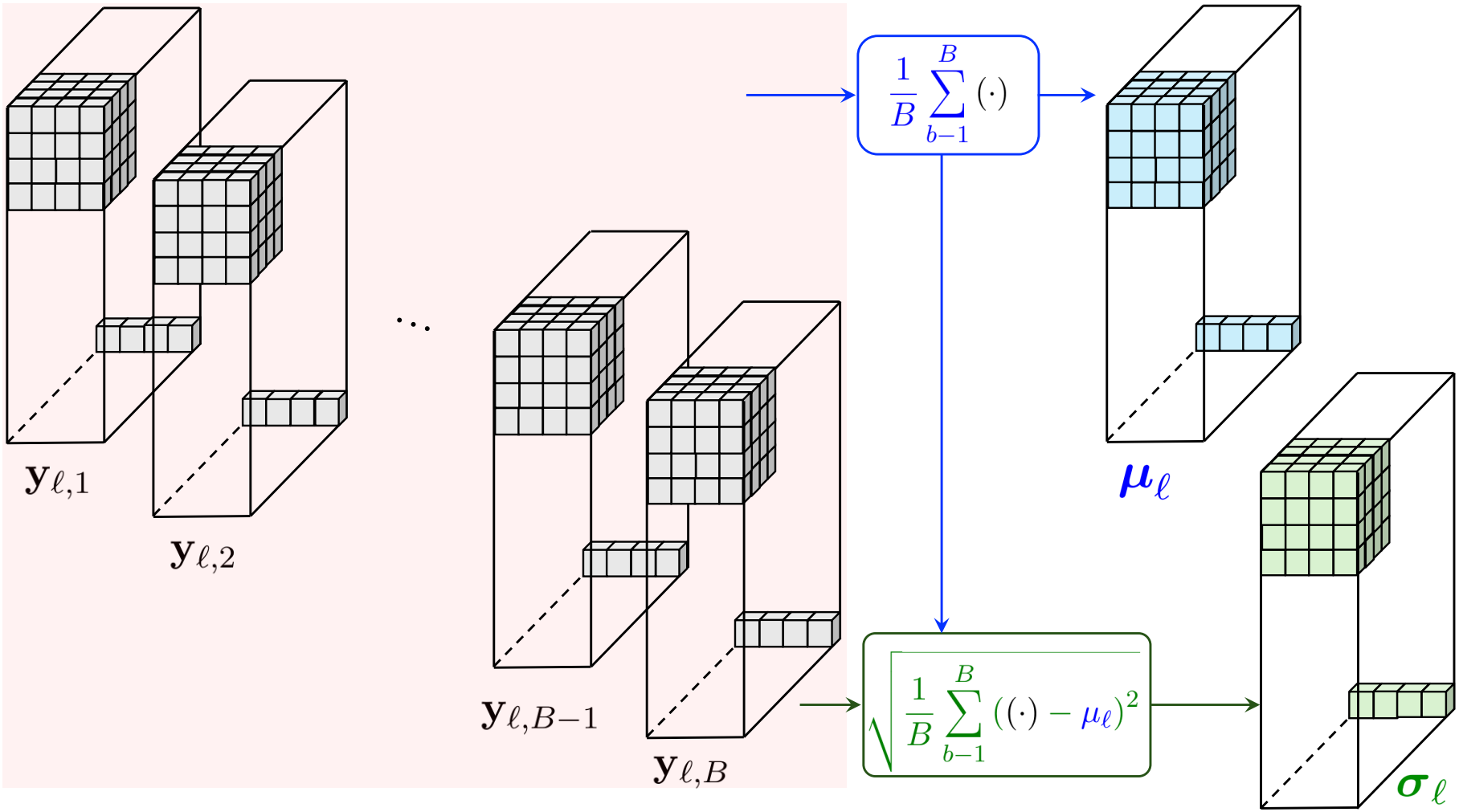
Visualization: *Batch Normalization*



Visualization: *Batch Normalization*

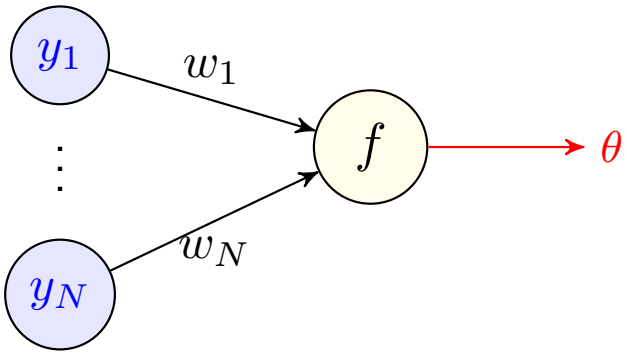


Visualization: *Batch Normalization*



Batch Normalization: *Single Neuron*

Let's focus on a *single neuron* in a *hidden layer*



Assume we train via mini-batches with batch size Ω

Batch Normalization: *Forward Pass*

We wait till we have $\mathbf{y}_\omega$ for $\omega = 1, \ldots, \Omega$

1. *approximate mean and variance as*

$$\boldsymbol{\mu} = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega}\mathbf{y}_\omega \qquad \boldsymbol{\sigma} = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega}(\mathbf{y}_\omega - \boldsymbol{\mu})^2}$$

2. *normalize* $\mathbf{y}_\omega$ *for* $\omega = 1, \ldots, \Omega$ *as*

$$\mathbf{u}_\omega = \frac{\mathbf{y}_\omega - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

→ 3. *scale and shift to a common place*

$$\bar{\mathbf{y}}_\omega = \boldsymbol{\gamma} \odot \mathbf{u}_\omega + \boldsymbol{\beta}$$

4. *compute* $z_\omega = \mathbf{w}^\mathsf{T}\bar{\mathbf{y}}_\omega$ *and activate as* $\theta_\omega = f(z_\omega)$

Batch Normalization: *Learnable Shift and Scale*

- ? *Why do we scale and shift after normalization?*
- There is no reason that going center with unit variance is the best

Depending on activation, it might be better *to center all features at another place with some different variances*

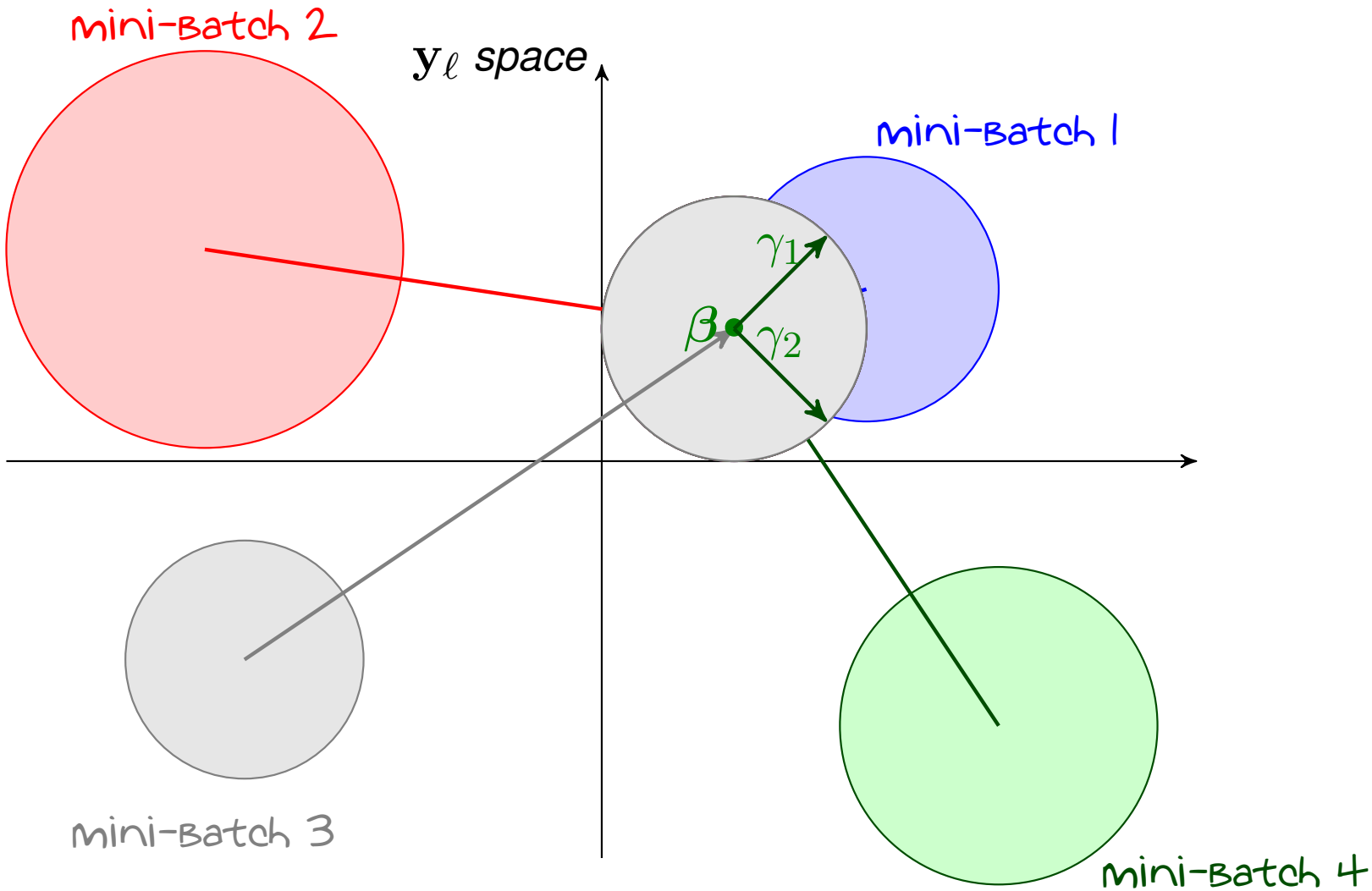
- ! *It seems hard to engineer this point and the variance*
- *We hence introduce a general point* $\boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathbb{R}^N$
  - *We treat them as learnable parameters*
  - *We update them in backward pass*

Batch Normalization as a Layer

We can treat batch normalization as a layer with learnable parameters

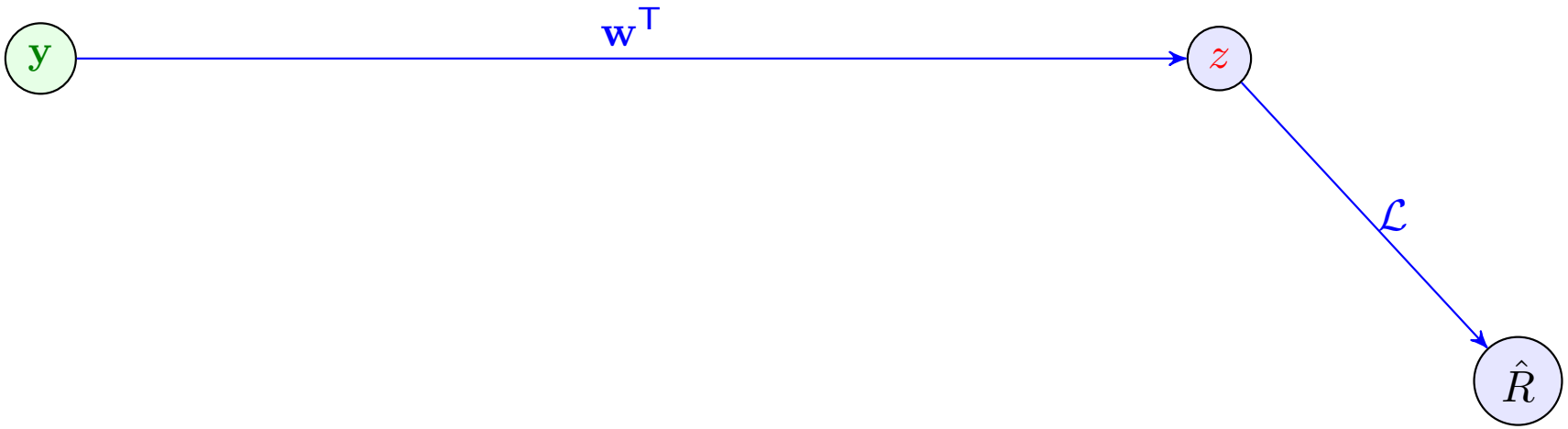
$$\mathcal{B}_{\mathcal{N}}\left(\mathbf{y}|\boldsymbol{\gamma}, \boldsymbol{\beta}\right) = \boldsymbol{\gamma} \odot \mathbf{u} + \boldsymbol{\beta}$$

Batch Normalization: *Learnable Shift and Scale*



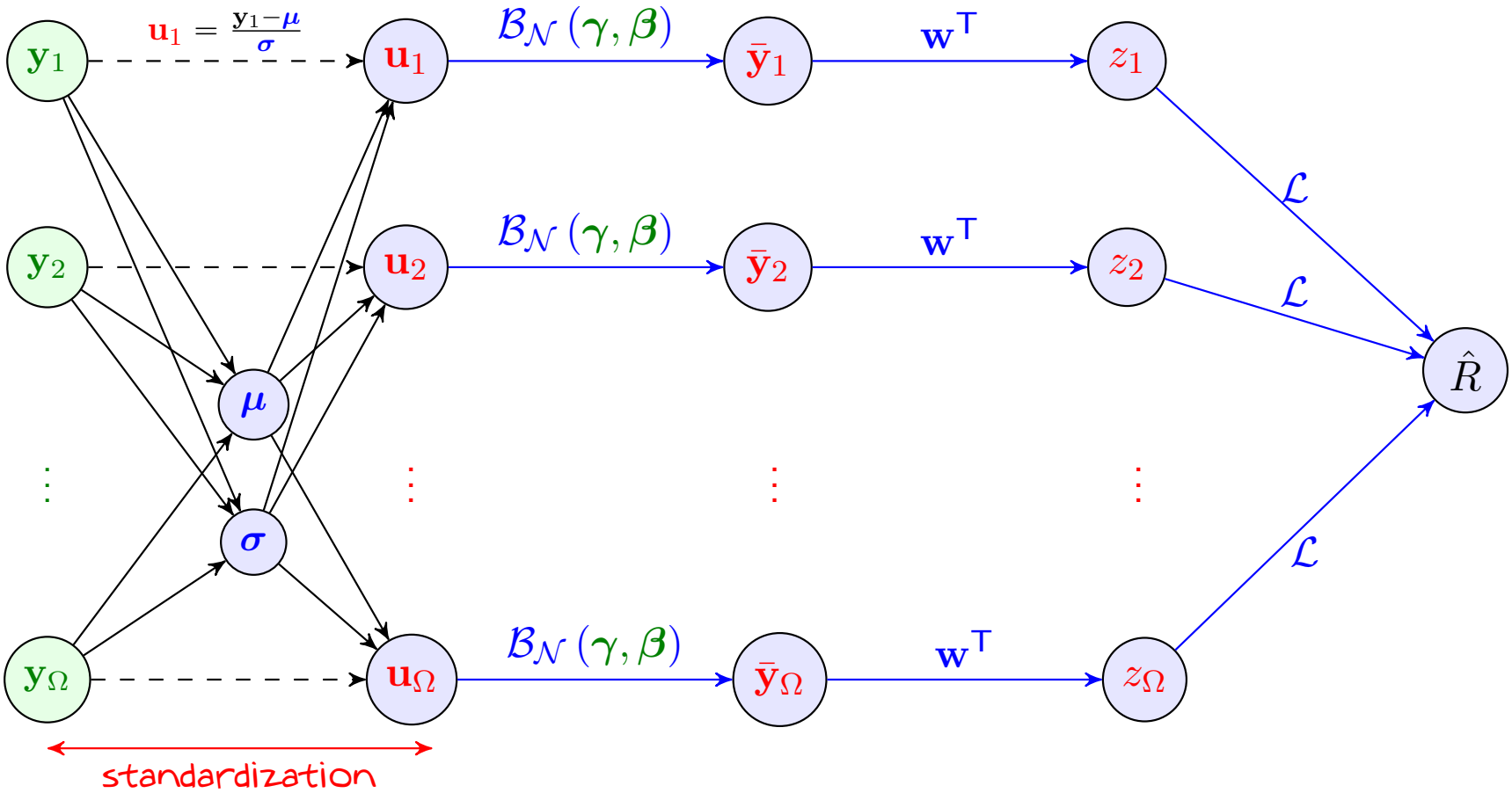
Batch Normalization: *Computation Graph*

In good old days without batch-normalization, we had



But, it gets more complicated now!

Batch Normalization: *Computation Graph*



$$\boldsymbol{\mu} = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega} \mathbf{y}_\omega \qquad \boldsymbol{\sigma} = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega} (\mathbf{y}_\omega - \boldsymbol{\mu})^2}$$

Batch Normalization: *Backpropagation*

Now assume that *forward pass* *is over for the complete mini-batch*

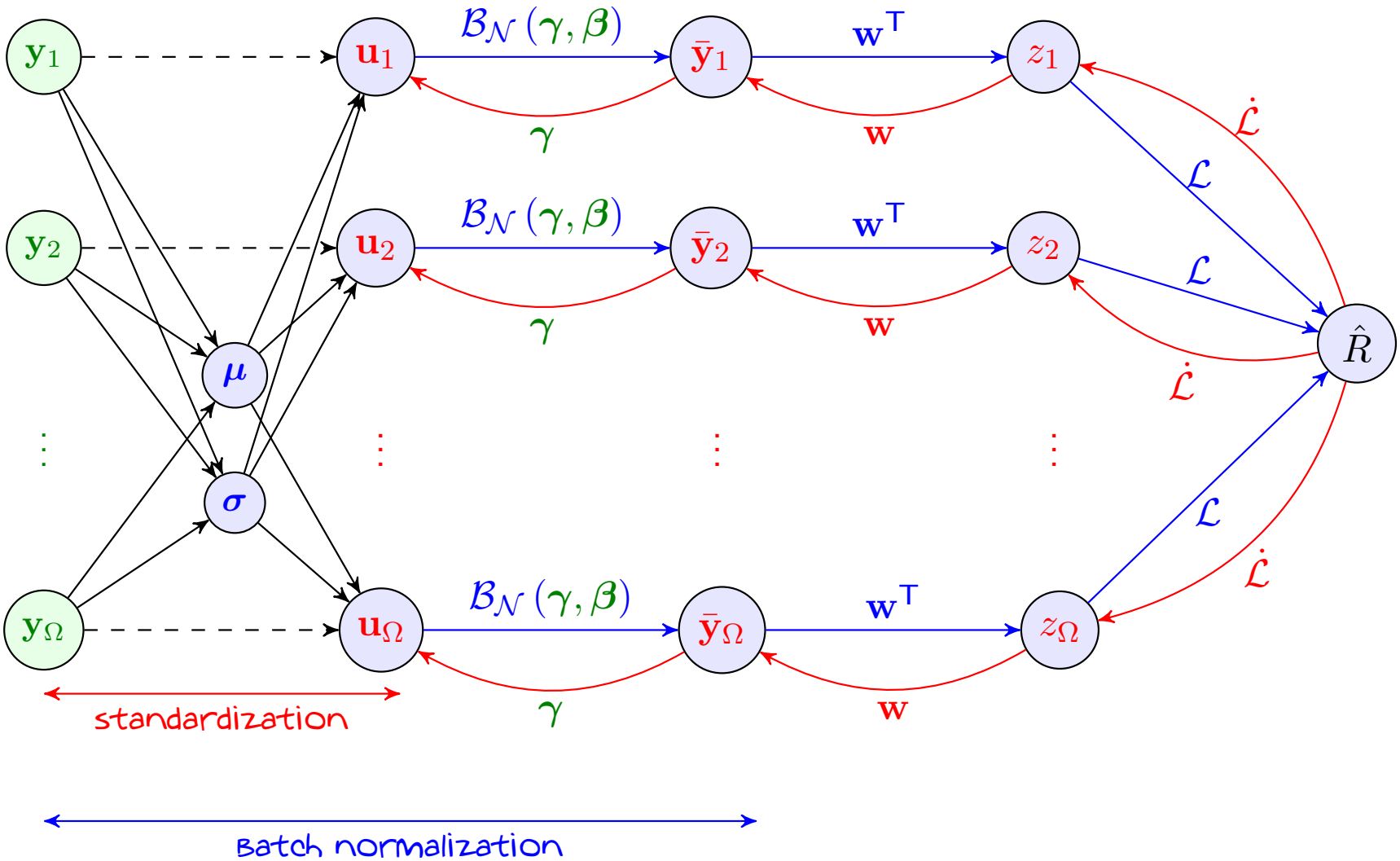
we could easily get back to normalized variables, i.e., $\bar{\mathbf{y}}_\omega$

First, we note that *our empirical risk reads*

$$\hat{R} = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega} \underbrace{\mathcal{L}\left(\theta_\omega, v_\omega\right)}_{\hat{R}_\omega} = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega} \hat{R}_\omega$$

- *We compute* $\nabla_{z_\omega} \hat{R}_\omega$
- *We compute* $\nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$
- *We compute* $\nabla_{\mathbf{u}_\omega} \hat{R}_\omega$

Batch Normalization: *Backpropagation*



Batch Normalization: *Backpropagation*

For $\omega = 1, \ldots, \Omega$, we have

$$\nabla_{\boldsymbol{\gamma}} \hat{R}_\omega = \mathbf{u}_\omega \odot \nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$$
$$\nabla_{\boldsymbol{\beta}} \hat{R}_\omega = \nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$$

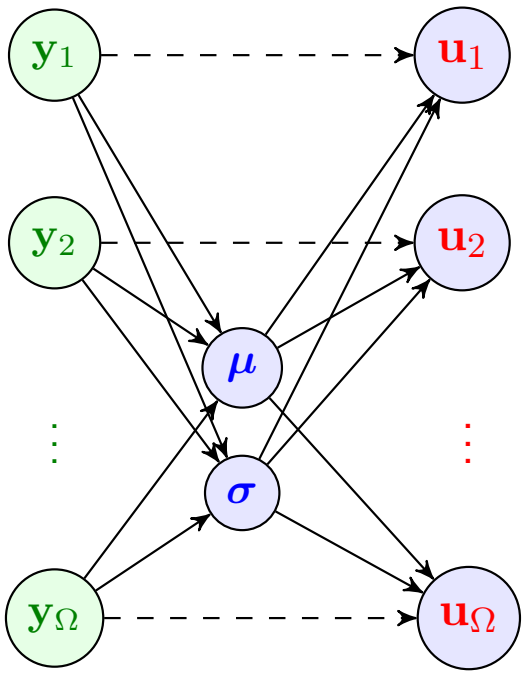
In our optimizer, we then update $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ using $\nabla_{\boldsymbol{\gamma}} \hat{R}_\omega$ and $\nabla_{\boldsymbol{\beta}} \hat{R}_\omega$: *for instance with standard SGD we have*

$$\boldsymbol{\gamma}^{(t+1)} = \boldsymbol{\gamma}^{(t)} - \frac{\eta}{\Omega} \sum_{\omega=1}^{\Omega} \nabla_{\boldsymbol{\gamma}} \hat{R}_\omega$$

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \frac{\eta}{\Omega} \sum_{\omega=1}^{\Omega} \nabla_{\boldsymbol{\beta}} \hat{R}_\omega$$

Batch Normalization: *Backpropagation*

The most challenging block is standardization: we open expand everything



Let's write again forward pass

$$\mathbf{u}_\omega = \frac{\mathbf{y}_\omega - \boldsymbol{\mu}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}{\boldsymbol{\sigma}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}$$

The chain rule would give us

$$\nabla_{\mathbf{y}_\tau} \mathbf{u}_\omega = \nabla_{\mathbf{y}_\tau} \mathbf{u}_\omega \circ 1 + \nabla_{\boldsymbol{\mu}} \mathbf{u}_\omega \circ \nabla_{\mathbf{y}_\tau} \boldsymbol{\mu} + \nabla_{\boldsymbol{\sigma}} \mathbf{u}_\omega \circ \nabla_{\mathbf{y}_\tau} \boldsymbol{\sigma}$$

Batch Normalization: *Testing*

- ? *After training, how do we test it for a single new point? We do not have any mini-batch anymore!*
- Good point! We use *moving average* of training means and variances

Inference with Batch Normalization

During training, at each layer ℓ, we start with some initial $\bar{\boldsymbol{\mu}}_\ell$ and $\bar{\boldsymbol{\sigma}}_\ell$ and update

$$\bar{\boldsymbol{\mu}}_\ell = \alpha \bar{\boldsymbol{\mu}}_\ell + (1-\alpha)\,\boldsymbol{\mu}_\ell$$
$$\bar{\boldsymbol{\sigma}}_\ell = \alpha \bar{\boldsymbol{\sigma}}_\ell + (1-\alpha)\,\boldsymbol{\sigma}_\ell$$

after each iteration for some $0 < \alpha < 1$: typically close to 1

We use these values for normalization during inference

Batch Normalization: *Final Points*

1. We could normalize before or after activation
   - ↳ *Here, we did it after activation*
   - ↳ *In the original proposal is was before activation*
   - ↳ *In general, it is not fully known which one is better*
     - ⇝ We may try both and see which one gives better result
2. With batch-normalization, we should distinguish training from evaluation
   - ↳ *We use* `model.train()` *for training*
     - ⇝ *Here, we update the moving averages*
   - ↳ *We use* `model.eval()` *for evaluation*
     - ⇝ *Here, we only use the moving averages*

Approach II. Layer Normalization

Laye normalization proposes an alternative normalization that does not depend on batch statistics: the idea is even simpler

✔ *compute scalar mean and variance of entries in sample feature* $\in \mathbb{R}^{M \times N \times K}$

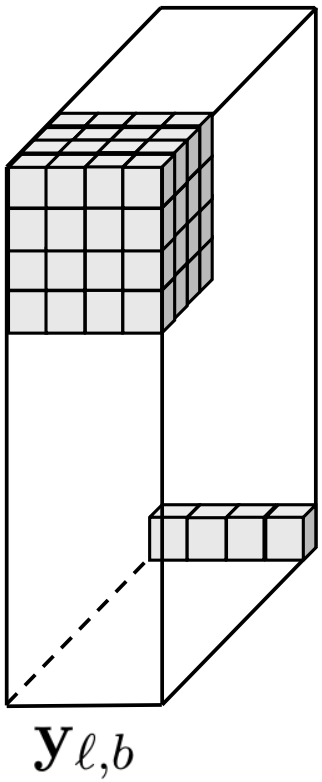
$$\mu_{\ell,b} = \frac{1}{MNK} \sum_{m,n,k} y_{\ell,b,m,n,k}$$

$$\sigma_{\ell,b} = \sqrt{\frac{1}{MNK} \sum_{m,n,k} (y_{\ell,b,m,n,k} - \mu_{\ell,b})^2}$$

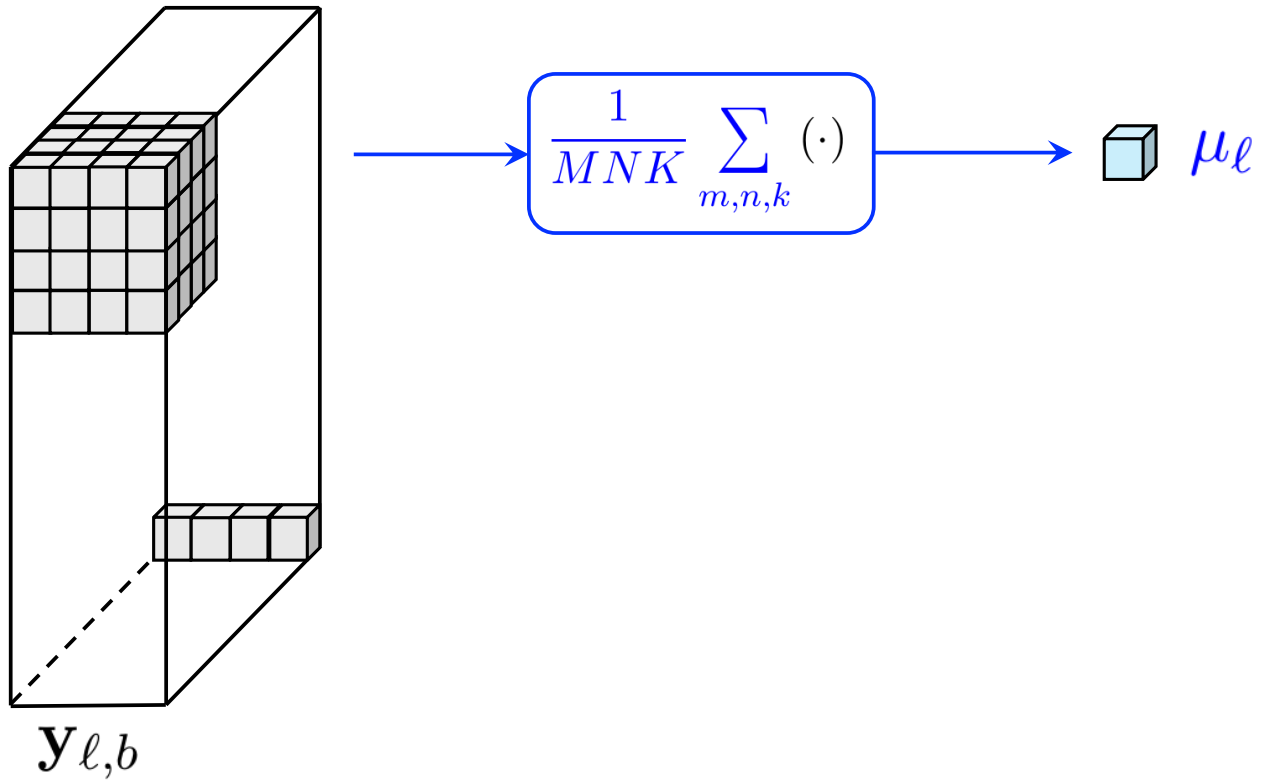
! *backpropagate by taking normalization into account*

- Again, $\mu_{\ell,b}$ and $\sigma_{\ell,b}$ depend on weights

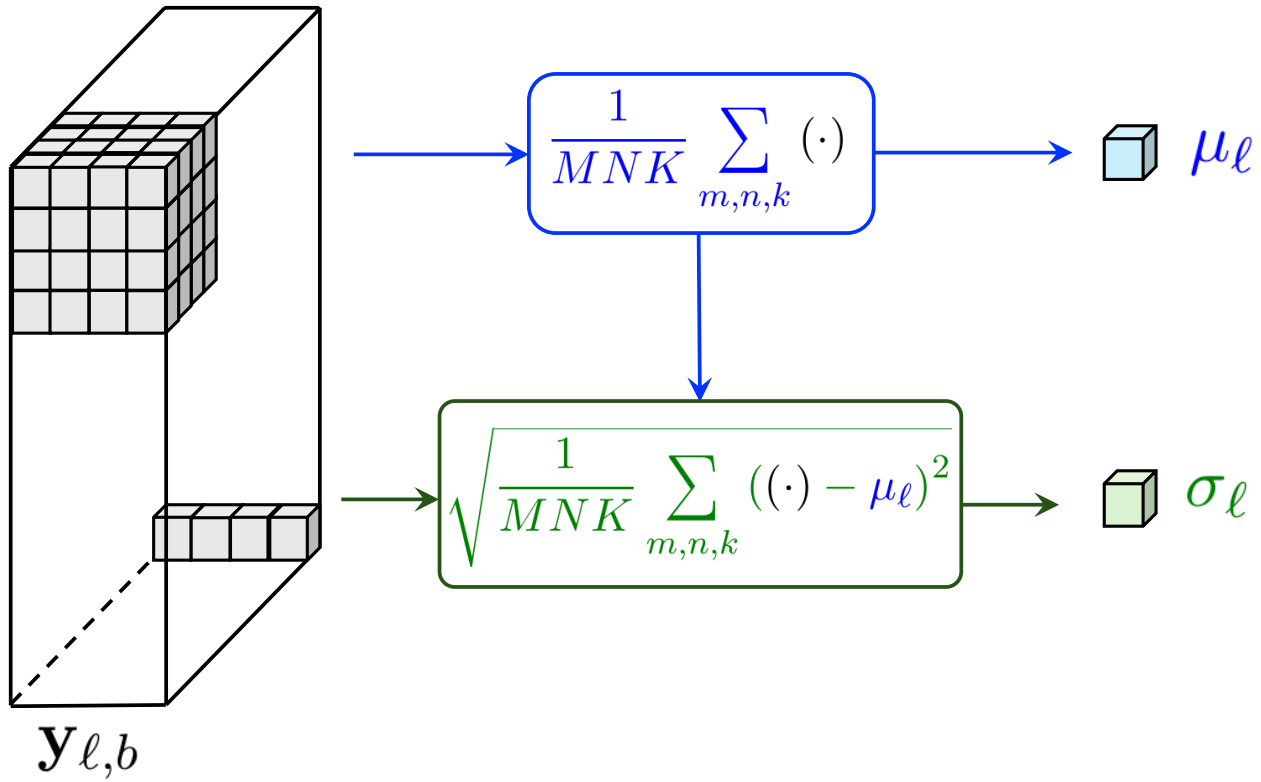
Visualization: *Layer Normalization*



Visualization: *Layer Normalization*



Visualization: *Layer Normalization*



Layer Normalization: *Forward Pass*

In layer normalization, we do not need to wait!

1. *approximate mean and variance as*

$$\mu = \frac{1}{\text{size}} \sum_{\text{all dim}} \mathbf{y} \qquad \sigma = \sqrt{\frac{1}{\text{size}} \sum_{\text{all dim}} (\mathbf{y} - \mu)^2}$$

2. *normalize all entries in* $\mathbf{y}$ *as*

$$\mathbf{u} = \frac{\mathbf{y} - \mu}{\sigma}$$

3. *scale and shift to a common place*

$$\bar{\mathbf{y}} = \boldsymbol{\gamma} \odot \mathbf{u} + \boldsymbol{\beta}$$

4. *compute* $z = \mathbf{w}^{\mathsf{T}} \bar{\mathbf{y}}$ *and activate as* $\theta = f(z)$

Layer Normalization as Layer

Similar to batch normalization, *we learn parameters* $\boldsymbol{\beta}, \boldsymbol{\gamma}$

- *We treat them as learnable parameters*
- *We update them in backward pass*

Layer Normalization as a Layer

We can treat batch normalization as a layer with learnable parameters

$$\mathcal{L}_{\mathcal{N}}\left(\mathbf{y}|\boldsymbol{\gamma}, \boldsymbol{\beta}\right) = \boldsymbol{\gamma} \odot \mathbf{u} + \boldsymbol{\beta}$$

Inference with Layer Normalization

Since the normalization uses a single sample only: *the inference runs exactly the same as training*

Layer Normalization: *Final Points*

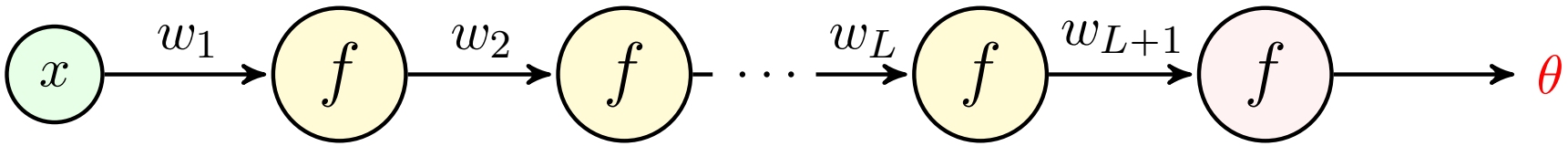
1. Again, with layer normalization, backpropagation is impacted
   - ↳ *You could try writing it as in batch normalization*
2. We could normalize before or after activation
   - ↳ *We may try both and see which one gives better result*
3. Layer normalization is a better approach if batch statistics change notably
   - ↳ *It does not depend on batch statistics*
   - ↳ *It behaves the same in training and inference*
4. Layer normalization is the favorite approach in sequence learning
   - ↳ *We will see it in design of Transformers*



UNIVERSITY OF
TORONTO

Residual Learning

Recap: Vanishing and Exploding Gradient



We compute derivative of loss with respect to the first weight w_1

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} = \overleftarrow{\theta}\ \underbrace{x\dot{f}\left(w_1 x\right)\prod_{\ell=2}^{L+1} w_\ell \dot{f}\left(w_\ell y_{\ell-1}\right)}_{\text{accumulated in Backpropagation}}$$

Vanishing Gradients: *Simple Example*

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} = \overleftarrow{\theta}\ \underbrace{x\dot{f}\left(w_1 x\right)\prod_{\ell=2}^{L+1} w_\ell \dot{f}\left(w_\ell y_{\ell-1}\right)}_{\text{accumulated in Backpropagation}}$$

- **Case I.** *We have a sigmoid activation and all weight are smaller than 1*

$$\lim_{L\uparrow\infty}\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} = 0$$

- **Case II.** *We have a ReLU activation and all weight are larger than 1*

$$\lim_{L\uparrow\infty}\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} \to \infty$$

Exploding-Vanishing Gradients: *Solutions*

- ? *How can we resolve this issue?*
- There are various solutions and we look at two key ones
  - ✔ Normalization
  - ! Residual Learning

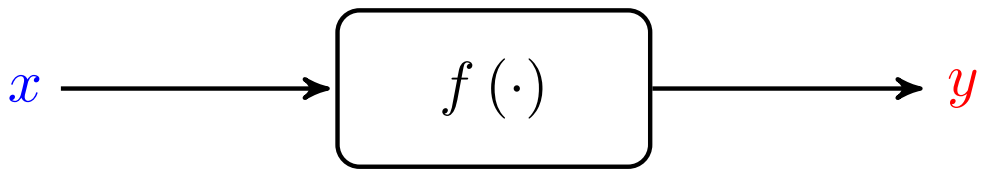
Residual Learning

The root idea is as follows: *instead of learning a function*

we can learn its residual and add it up with the input

Say we have input x and label y

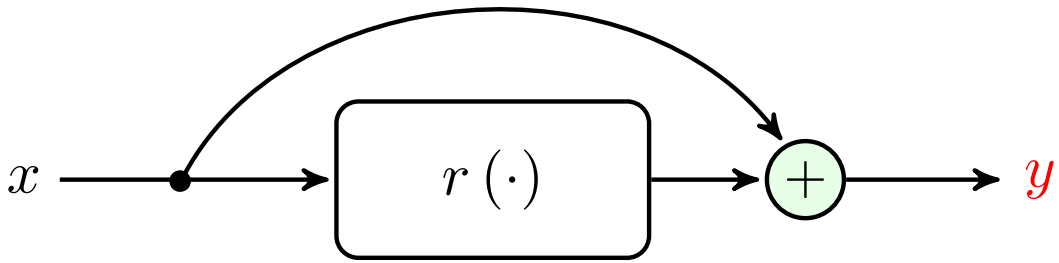
With plain NNs we learn a function $f(\cdot)$ that relates x to y



The NN approximates $f(\cdot)$ as best as it could

Residual Learning

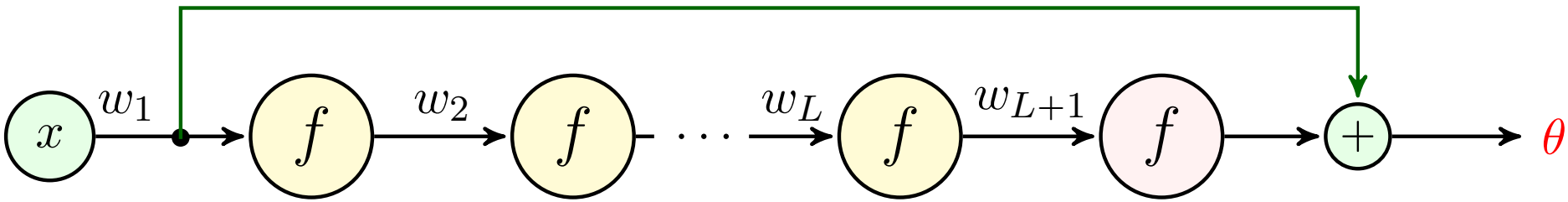
We can also learn a residual function $r\left(\cdot\right)$ that relates x to $y - x$: the end-to-end function is then given by



The NN now approximates the residual $r\left(\cdot\right)$ as best as it could

- ? *But, why should it be any different this time?*
- Let's get back to our simple example

Skip Connection: *Simple Example*



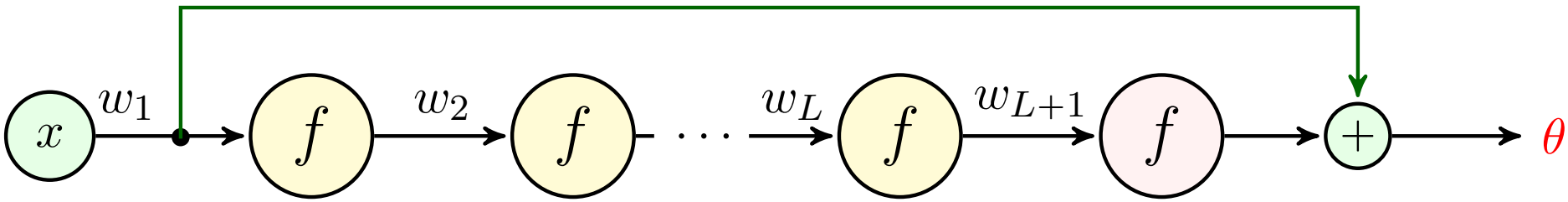
Let's see forward pass

- $y_1 = f(w_1 x)$
- $y_2 = f(w_2 y_1)$

  $\cdots$
- $y_L = f(w_L y_{L-1})$
- $y_{L+1} = f(w_{L+1} y_L)$

$\rightarrow \theta = y_{L+1} + w_1 x$

Skip Connection: *Simple Example*



What happens backward pass? We start with $\overleftarrow{\theta} = \mathrm{d}\hat{R}/\mathrm{d}\theta$ *again*

- $\theta = y_{L+1} + w_1 x$

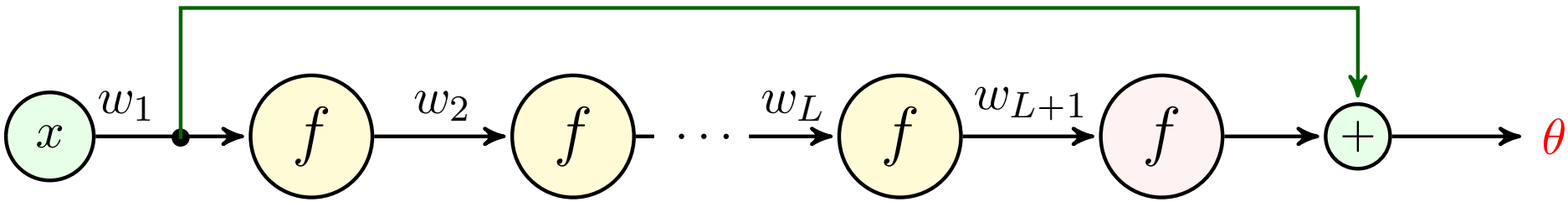
$$\overleftarrow{y}_{L+1} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_{L+1}} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}\theta}\frac{\mathrm{d}\theta}{\mathrm{d}y_{L+1}} = \overleftarrow{\theta}$$

- $y_{L+1} = f\left(w_{L+1} y_L\right)$

$$\overleftarrow{y}_L = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_L} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_{L+1}}\frac{\mathrm{d}y_{L+1}}{\mathrm{d}y_L} = \overleftarrow{y}_{L+1} w_{L+1} \dot{f}\left(w_{L+1} y_L\right)$$

$$= \overleftarrow{\theta}\, w_{L+1} \dot{f}\left(w_{L+1} y_L\right)$$

Skip Connection: *Simple Example*



What happens backward pass? We start with $\overleftarrow{\theta} = \mathrm{d}\hat{R}/\mathrm{d}\theta$ *again*

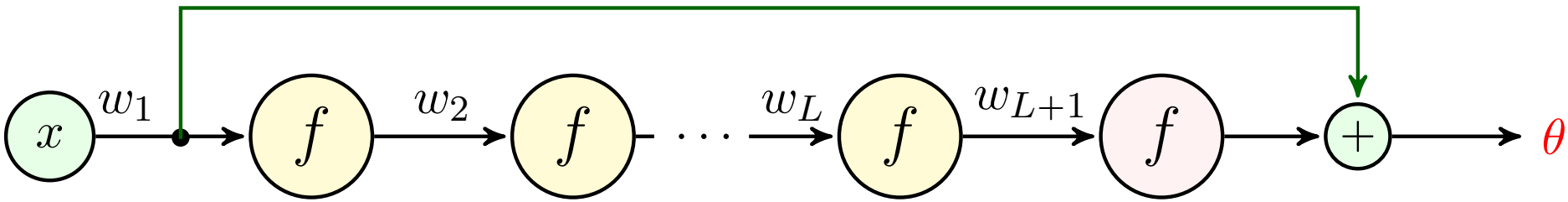
- $y_L = f\left(w_L y_{L-1}\right)$

$$\overleftarrow{y}_{L-1} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_{L-1}} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_L}\frac{\mathrm{d}y_L}{\mathrm{d}y_{L-1}} = \overleftarrow{y}_L w_L \dot{f}\left(w_L y_{L-1}\right)$$

$$= \overleftarrow{\theta}\, w_{L+1} w_L \dot{f}\left(w_L y_{L-1}\right) \dot{f}\left(w_{L+1} y_L\right)$$

- $\ldots$

Skip Connection: *Simple Example*



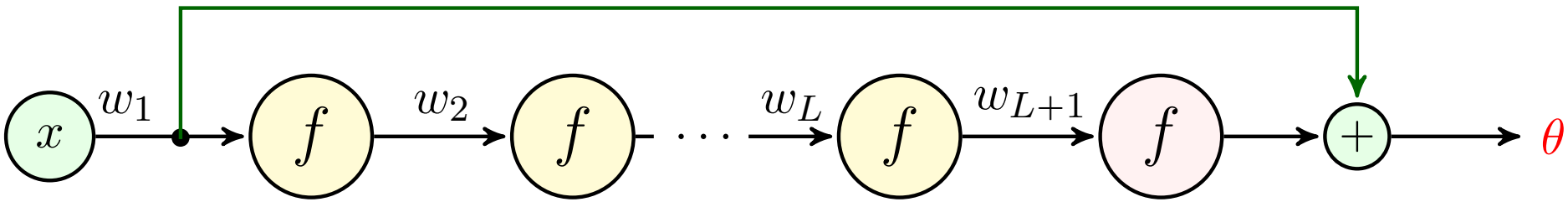
The backward pass looks the same up to the skip connection

- $y_2 = f\left(w_2 y_1\right)$

$$\overleftarrow{y}_1 = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_1} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_2}\frac{\mathrm{d}y_2}{\mathrm{d}y_1} = \overleftarrow{y}_2 w_2 \dot{f}\left(w_2 y_1\right)$$

$$= \overleftarrow{\theta} \prod_{\ell=2}^{L+1} w_\ell \dot{f}\left(w_\ell y_{\ell-1}\right)$$

Skip Connection: *Simple Example*



Now, let's compute derivative of loss with respect to the first weight w_1

- *Here, we have two links connected to* w_1 *in the network*
  - ↳ $y_1 = f(w_1 x)$
  - ↳ $\theta = y_{L+1} + w_1 x$

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} = \frac{\mathrm{d}\hat{R}}{\mathrm{d}y_1}\frac{\mathrm{d}y_1}{\mathrm{d}w_1} + \frac{\mathrm{d}\hat{R}}{\mathrm{d}\theta}\frac{\mathrm{d}\theta}{\mathrm{d}w_1}$$

$$= \overleftarrow{y_1} x \dot{f}(w_1 x) + \overleftarrow{\theta}\, x = \overleftarrow{\theta}\, x \left(\dot{f}(w_1 x) \prod_{\ell=2}^{L+1} w_\ell \dot{f}(w_\ell y_{\ell-1}) + 1 \right)$$

Skip Connection: *Simple Example*

With skip connection, the derivative of loss with respect to the first weight w_1 does not vanish

$$\frac{\mathrm{d}\hat{R}}{\mathrm{d}w_1} = \overleftarrow{\theta}\, x \left(\underbrace{\dot{f}\left(w_1 x\right) \prod_{\ell=2}^{L+1} w_\ell \dot{f}\left(w_\ell y_{\ell-1}\right)}_{\text{accumulated in backpropagation}} \quad +1 \rightarrow \text{skip connection} \right)$$

Even if all weights and derivatives are smaller than one, as we get very deep $\overleftarrow{\theta}$ is still backpropagating

Skip Connection and Residual Unit

Skip Connection

Skip connection refers to links that carry information from layer $\ell - s$ to layer ℓ for $s > 1$

Let $\mathbf{Z}_\ell$ and $\mathbf{Y}_\ell = f(\mathbf{Z}_\ell)$ be outputs of layer ℓ before and after activation

- ✔ *This layer can be a convolution or fully-connected layer*

Residual Unit

With a skip connection of depth s, the output of layer ℓ is

$$\mathbf{Y}_\ell = f(\mathbf{Z}_\ell + \mathbf{W} \circ \mathbf{Y}_{\ell-s})$$

- ↳ $\mathbf{W}$ *is a linear transform that can be potentially learned*
- ↳ $\circ$ *is a kind of product that matches the dimensions*

Skip Connection and Residual Unit

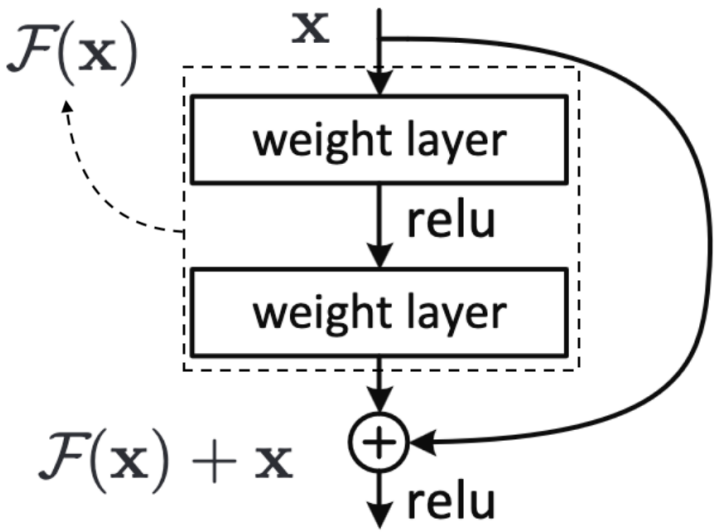
- + *What is exactly* $\mathbf{W}$ *?*
- – It is a set of weights, like other weighted components. But, *we don't really need it. We could set it to some fixed values and don't learn it at all*
- + *Why should we connect activated output to linear output?*
- – There is actually no should here also!

In original proposal it was suggested to connect *activated output of a layer to the linear output of some next layers, i.e., add* $\mathbf{Y}_{\ell-s}$ *to* $\mathbf{Z}_{\ell}$

- ↳ *This combination is only a suggestion!*
- ↳ *Later suggestions proposed to connect linear output* $\mathbf{Z}_{\ell-s}$
- ↳ *Similar to batch normalization, best combination is found by experiment*

Residual Unit: *New Building Block*

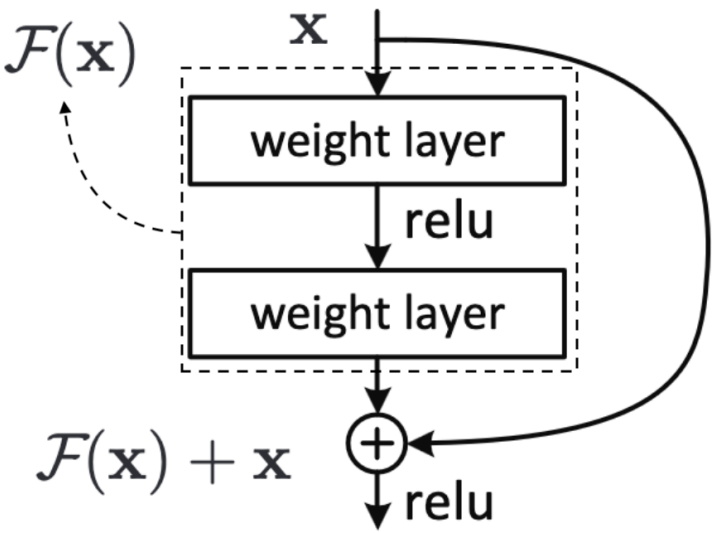
Residual unit can be seen as a new building block



Skip connection allowed for training deeper NNs: *experiments showed that*

- *Using skip connection with batch normalization makes results sensible*
  - ↳ *Deeper networks show better training performance*
- *Skip connection seems to be a crucial component for going deep*

Classical Residual Unit



Classical residual units consist of

- *multiple (typically 2) weighted layers, e.g.,*
  - ↳ *convolutional layer*
  - ↳ *standard fully-connected layer*
- *single skip connection*
  - ↳ *generally a weighted connection*

$$\textit{output} = \mathrm{ReLU}(\mathcal{F}(\mathbf{x}) + \mathbf{W}_s\mathbf{x})$$

  - ↳ *If $\mathcal{F}(\mathbf{x})$ and $\mathbf{x}$ are of same dimension $\mathbf{W}_s = \mathbf{I}$ is a good choice!*

The idea was proposed by Microsoft to win ILSVRC

Forward Pass through Residual Units

In original proposal, Residual Units are implemented by convolutional layers

1. *Input* $\mathbf{x} = \mathbf{X}$ *is a tensor with* C *channels*
2. $\mathbf{X}$ *is given to a convolutional layer with* C *output channels, i.e.,* C *filters*

$$\mathbf{Z}_1 = \text{Conv}\left(\mathbf{X}|\mathbf{W}_1^1, \ldots, \mathbf{W}_C^1\right)$$

3. $\mathbf{Z}_1$ *is then activated* $\mathbf{Y}_1 = f\left(\mathbf{Z}_1\right)$
4. $\mathbf{Y}_1$ *is given to another convolutional layer with* C *output channels*

$$\mathbf{Z}_2 = \text{Conv}\left(\mathbf{Y}_1|\mathbf{W}_1^2, \ldots, \mathbf{W}_C^2\right)$$

5. *Output* $\mathbf{Y}$ *is constructed by activating* $\mathbf{Z}_2$ *after skip connection*

$$\mathbf{Y} = f\left(\mathbf{Z}_2 + \mathbf{X}\right)$$

Backpropagation through Residual Units

Assume we have $\nabla_{\mathbf{Y}} \hat{R}$

1. $\nabla_{\mathbf{Z}_2} \hat{R}$ *is computed by entry-wise production with* $\dot{f}\left(\mathbf{Z}_2 + \mathbf{X}\right)$
2. $\nabla_{\mathbf{Y}_1} \hat{R}$ *is computed by convolution*

$$\nabla_{\mathbf{Y}_1} \hat{R} = \text{Conv}\left(\nabla_{\mathbf{Z}_2} \hat{R} | \mathbf{W}_1^{2\dagger}, \ldots, \mathbf{W}_C^{2\dagger}\right)$$

3. $\nabla_{\mathbf{Z}_1} \hat{R}$ *is computed by entry-wise production with* $\dot{f}\left(\mathbf{Z}_1\right)$
4. $\nabla_{\mathbf{X}} \hat{R}$ *is given by a new chain rule*

$$\nabla_{\mathbf{X}} \hat{R} = \underbrace{\nabla_{\mathbf{Z}_1} \hat{R} \circ \nabla_{\mathbf{X}} \mathbf{Z}_1}_{\text{Backward convolution}} + \nabla_{\mathbf{Y}} \hat{R} \underbrace{\circ}_{\odot} \underbrace{\nabla_{\mathbf{X}} \mathbf{Y}}_{\dot{f}(\mathbf{Z}_2 + \mathbf{X})}$$

$$= \text{Conv}\left(\nabla_{\mathbf{Z}_1} \hat{R} | \mathbf{W}_1^{1\dagger}, \ldots, \mathbf{W}_C^{1\dagger}\right) \underbrace{+\nabla_{\mathbf{Y}} \hat{R} \odot \dot{f}\left(\mathbf{Z}_2 + \mathbf{X}\right)}_{\text{avoids vanishing gradients}}$$

Residual Networks

We can now look at *Residual Unit* as a single block in a deep NN

- *We build an architecture by cascading them*
  - ↳ *Intuitively we can go deeper now, since we use skip connection*
- *We also add other kinds of layers that we know, e.g.,*
  - ↳ *convolutional layers, pooling layers, fully-connected layers*
- *We can do whatever we have done before to train them efficiently, e.g.,*
  - ↳ *dropout or other regularization techniques*
  - ↳ *input and batch normalization*
  - ↳ *more advanced optimizers*

ResNet

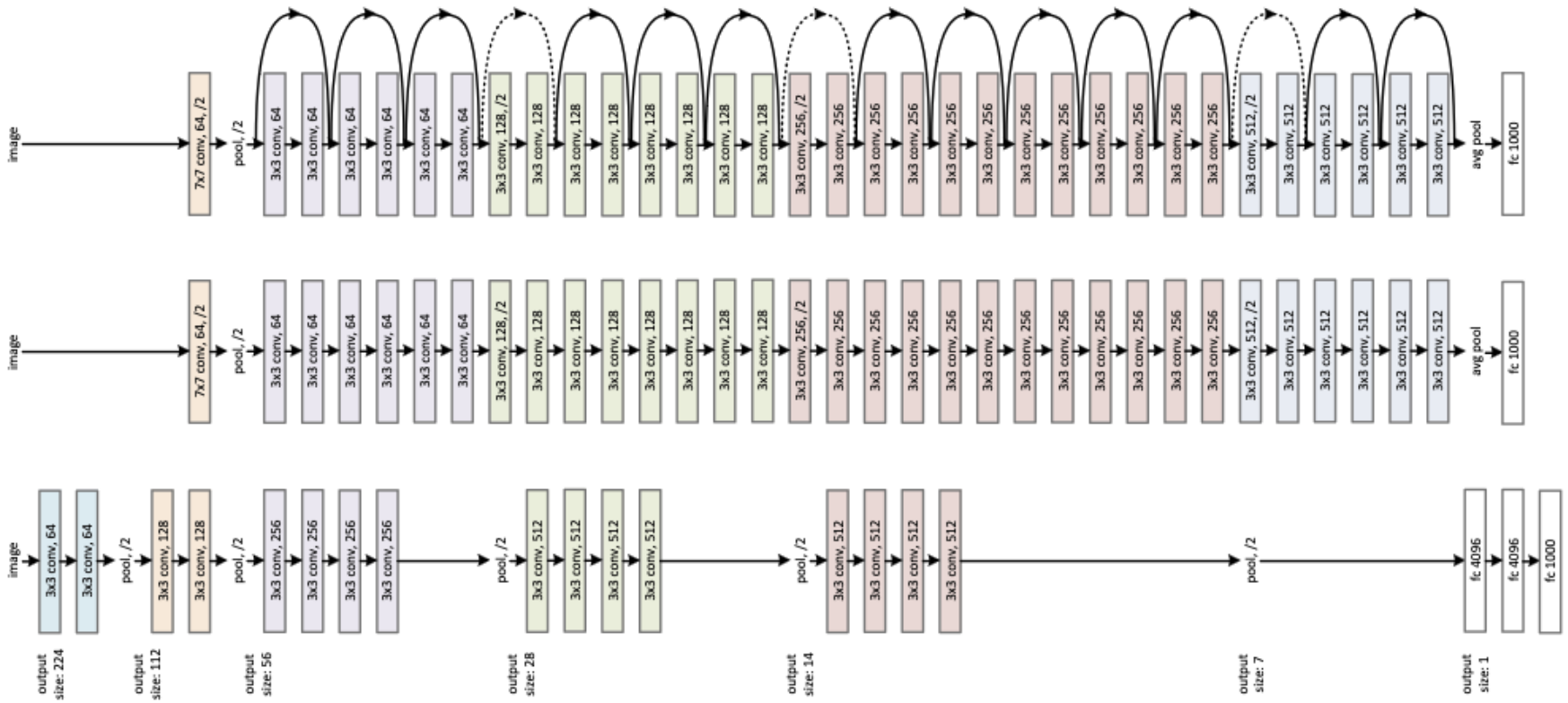
These kinds of NNs are called Residual Networks or shortly ResNet

- ↳ *they were proposed in the original paper introduced Residual Unit*
- ↳ *they are nowadays a kind of benchmark in many applications*

ResNet Breakthrough

ResNet inventors did a nice experiment to show the impact of skip connection[1]

- *They investigated 3 architectures: 34-layer ResNet, 34-layer Plain CNN (no skip connection) and benchmark VGG architectures*



[1] Find the details in their paper

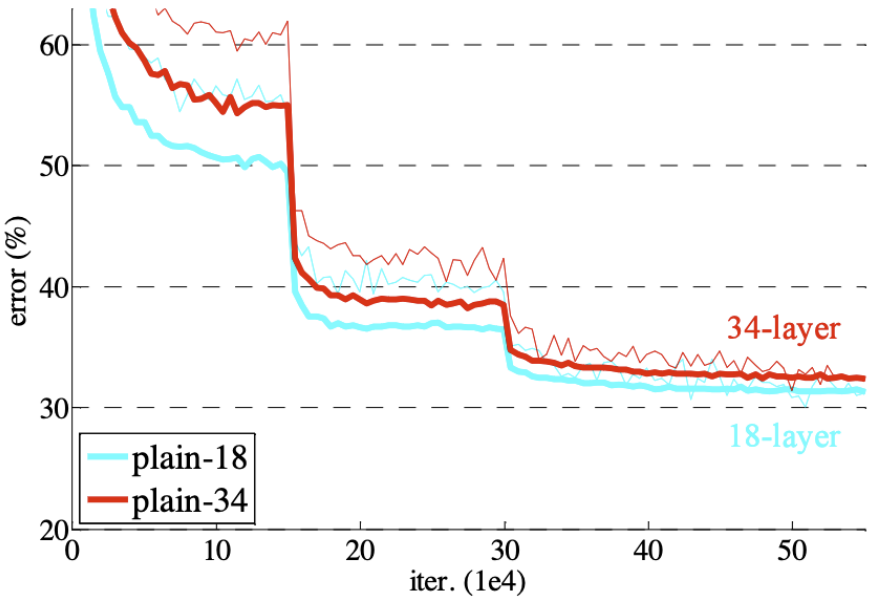
Residual Networks: *A Nice Experiment*

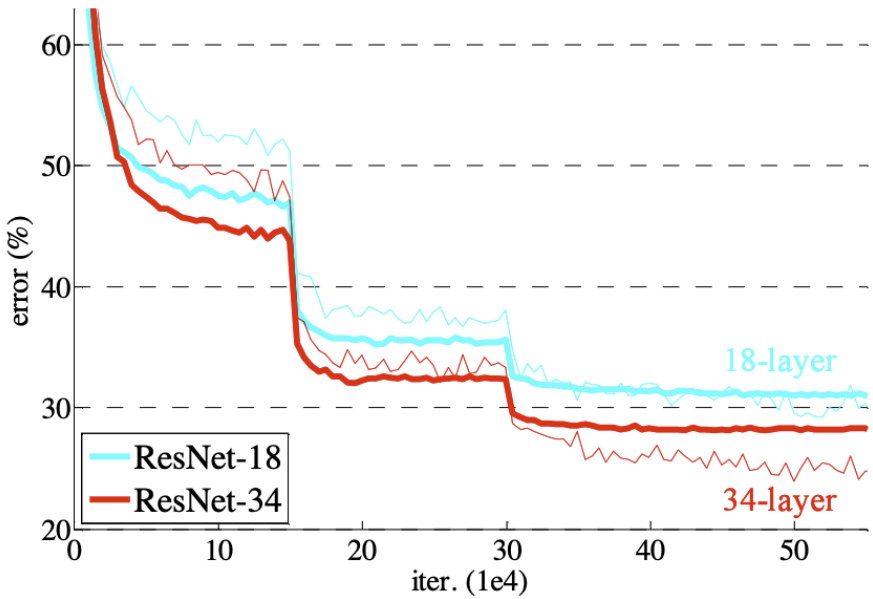
The results was interesting: *the depth challenge is now efficiently addressed, i.e., by going deeper we always get better in performance*

model	top-1 err.	top-5 err.
VGG-16 [41]	28.07	9.33
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40

Residual Networks: *A Nice Experiment*

Recall that *the depth challenge was different from overfitting! With deeper architectures, plain CNNs are showing worse "training" performance. With ResNet, this is not the case anymore*





The above figures show training error meaning that

the left figure cannot simply show overfitting!

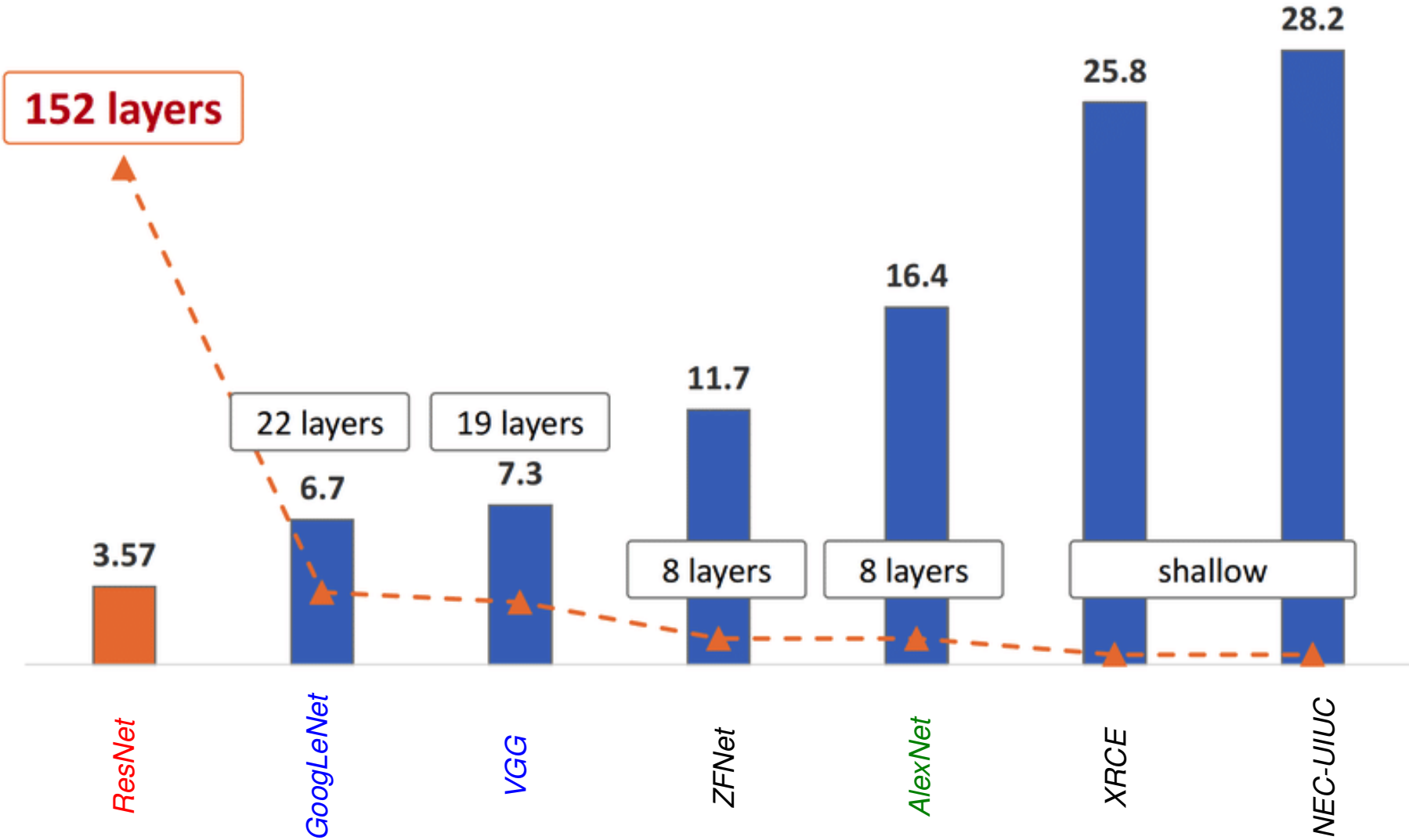
Residual Networks: *ILSVRC*

Getting rid of this undesired behavior *made it easy to go as deep as we want*

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

This way ResNet won ILSVRC in 2015!

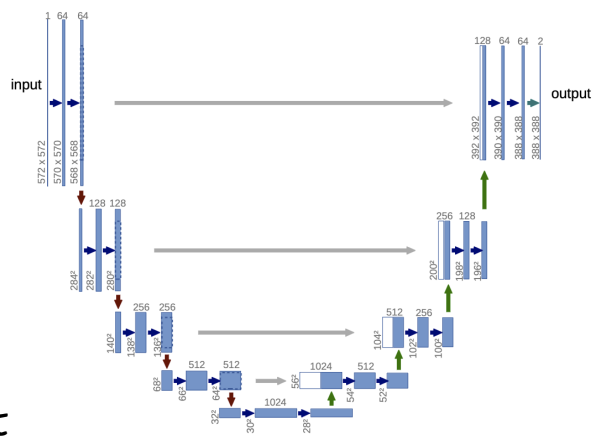
Depth vs Accuracy: *ILSVRC Winners till 2015*



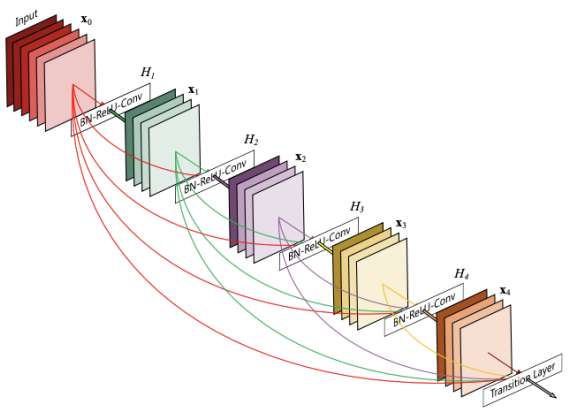
Other Architectures with Skip Connection

ResNet is not the only architecture with skip connection

- *ResNet made a breakthrough by using short skip connection*
- *U-Net uses long and short skip connections*[2]
- *DenseNet uses long and short skip connections densely to go even deeper*[3]



U-Net



DenseNet

[2] Check the original proposal of U-Net here

[3] Check the original proposal of DenseNet here

Pretrained Models

Many architectures have been already implemented and pre-trained in PyTorch

- *They are implemented using basic blocks in PyTorch*
- *They are pre-trained for ImageNet classification*
  - ↳ *They get 3-channel 224×224 tensor as input*
  - ↳ *They return 1000-dimensional output vector*

We can access them through `torchvision.models`

```
1 import torchvision.models as Models
2 model = Models.
```

```
resnet
ResNet
resnet101
ResNet101_Weights
resnet152
ResNet152_Weights
resnet18
ResNet18_Weights
resnet34
ResNet34_Weights
```

Pretrained Models

For instance, we could load ResNet-50 with its pre-trained weights

```python
from torchvision.models import resnet50, ResNet50_Weights

# Old weights with accuracy 76.130%
resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)

# New weights with accuracy 80.858%
resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)

# Best available weights (currently alias for IMAGENET1K_V2)
# Note that these weights may change across versions
resnet50(weights=ResNet50_Weights.DEFAULT)

# Strings are also supported
resnet50(weights="IMAGENET1K_V2")

# No weights - random initialization
resnet50(weights=None)
```

Finetuning Pretrained Models

+ *What if we are using it for other applications with different data size?*
– We could add or modify layer to it; *for instance,*

Say we want to use a pre-trained ResNet to classify MNIST: we could replace the first convolutional layer with single-channel 28×28 input and the same number of output channels. We further replace the output layer with a fully-connected layer with 10 classes

+ *But it does not perform well! Does it?!*
– Of course not! It has been trained for ImageNet and there is no reason to work with MNIST! But, *we can start from those weights and do normal training for several epochs*
  ↳ *It probably gets trained much faster as compared to random initialization*
  ↳ *This idea is studied in a much broader sense in Transfer Learning*



UNIVERSITY OF
TORONTO

Further Read

Further Read

Read More on These Topics

- Bishop & Bishop, *Deep Learning: Foundations and Concepts*
  - ↳ Chapter 7

Nice to Check

1. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*
   - ↳ Ioffe and Szegedy (2015)
2. *Deep Residual Learning for Image Recognition*
   - ↳ K. He et al. (2015)