

## Residual Blocks & Normalization Layers — Lecture Notes

### 1. Residual Blocks

#### 1.1 Motivation

Deep neural networks suffer from:

- **Vanishing gradients** in deep architectures
- **Degradation problem:** deeper networks often perform *worse*, not better
- Difficulty learning identity mappings when layers are unnecessary

Residual connections were introduced in **ResNet (He et al., 2015)** to solve these issues.

#### 1.2 Key Idea

Instead of forcing a stack of layers to learn a complete transformation:

$$H(x)$$

the network learns only the **residual**:

$$R(x) = H(x) - x$$

and outputs:

$$H(x) = x + R(x)$$

This makes optimization easier because:

- Learning a zero residual  $\rightarrow$  identity mapping is trivial
- Gradients have a clear bypass path ( skip connection )

---

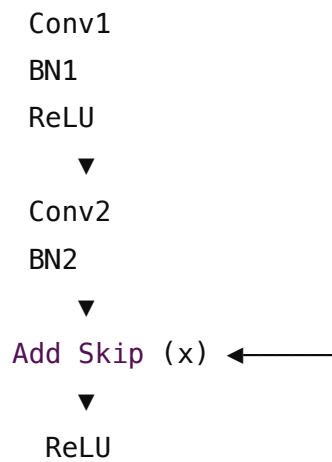
### 1.3 Structure of a Basic Residual Block

#### Diagram

sql

 Copy code





This is the **standard 2-convolution basic block** used in ResNet-18/34.

### Why two convolutions?

- A *single* convolution after the skip has limited expressive power.
- Two stacked convolutions with a nonlinearity form a **second-order transform** capable of meaningful feature refinement.
- More convolutions (3, 4+) are possible, but:
  - more expensive
  - harder to train
  - unnecessary for most tasks
  - bottleneck blocks handle depth more efficiently ( $1\times 1 \rightarrow 3\times 3 \rightarrow 1\times 1$ )

Thus **two convolutions** is the optimal compromise between expressivity and efficiency.

## 1.4 Residual Block Implementation (using custom LazyConv2d\_scratch)

python

 Copy code

```

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.conv1 = LazyConv2d_scratch(
            channels, kernel_size=3, padding=1, bias=False)
        self.bn1 = BatchNorm(channels, num_dims=4)

        self.conv2 = LazyConv2d_scratch(
            channels, kernel_size=3, padding=1, bias=False)
        self.bn2 = BatchNorm(channels, num_dims=4)

    def forward(self, x):

```

```
identity = x
out = F.relu(self.bn1(self.conv1(x)))
out = self.bn2(self.conv2(out))
return F.relu(out + identity)
```

## Why not reuse the same conv for both operations?

Because using the **same weights twice** collapses the block into a **single transformation**:

$$W(\sigma(Wx))$$

This reduces expressive power and breaks the underlying assumptions of ResNets. Each layer must learn a **different** linear transform.

---

## 1.5 Incorporating Residual Blocks into BNLeNet

Example (one residual block per stage):

nginx

 Copy code

```
Conv → BN → ReLU → ResidualBlock → MaxPool
Conv → BN → ReLU → ResidualBlock → MaxPool
Conv → BN → ReLU → ResidualBlock → MaxPool
```

This yields a “LeNet + residuals” hybrid — deeper, more expressive, and easier to train.

---

## 2. Batch Normalization (BN)

### 2.1 Motivation

- Makes optimization of deep nets more stable.
  - Controls activation distributions ("internal covariate shift").
  - Allows larger learning rates.
  - Provides a regularization effect.
  - Essential for training very deep CNNs.
- 

### 2.2 BN (Training-Time) Forward Pass

For input  $x$  of shape  $(N, C, H, W)$ :

## 1. Compute mean over batch (and spatial dims for CNN):

$$\mu_c = \frac{1}{NHW} \sum_{n,h,w} x_{n,c,h,w}$$

## 2. Compute variance:

$$\sigma_c^2 = \frac{1}{NHW} \sum_{n,h,w} (x_{n,c,h,w} - \mu_c)^2$$

## 3. Normalize:

$$\hat{x}_{n,c,h,w} = \frac{x_{n,c,h,w} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

## 4. Scale + Shift (learnable):

$$y = \gamma_c \hat{x} + \beta_c$$

---

## 2.3 Inference-time behavior

BatchNorm does **not** use batch statistics during inference.

It uses **running averages** collected during training:

$$\mu_c^{(running)}, \quad \sigma_{c,(running)}^2$$

---

## 2.4 Diagram

mathematica

 Copy code

Input  $x$

|

▼

Compute  $\mu_c, \sigma_c^2$  (over batch)

|

▼

Normalize:  $\hat{x} = (x - \mu) / \sqrt{(\sigma^2 + \epsilon)}$

|

▼

Scale+Shift:  $y = \gamma \hat{x} + \beta$

|

### 3. Layer Normalization (LN)

#### 3.1 Why BN fails for RNNs/Transformers

BN depends on **batch statistics**, but:

- RNNs process sequences step-by-step
- Each time step would have different batch stats  $\rightarrow$  unstable
- Transformers often use tiny or variable batch sizes
- Token-level independence must be preserved

Thus **BN is unsuitable** for sequence models.

#### 3.2 LayerNorm Operation

Given tensor:

$$x \in \mathbb{R}^{B \times T \times F},$$

LN computes mean/variance **within each sample**, across the feature dimension:

$$\mu = \frac{1}{F} \sum_{j=1}^F x_j, \quad \sigma^2 = \frac{1}{F} \sum_{j=1}^F (x_j - \mu)^2$$

Normalize:

$$\hat{x}_j = \frac{x_j - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Scale/Shift (one  $\gamma$ ,  $\beta$  per feature):

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

#### 3.3 Key Difference: Axes of Reduction

Operation	BN	LN
Normalize across	Batch (and spatial dims)	Features (per sample)
Depends on batch size	Yes	No

Operation	BN	LN
Good for CNN?	Excellent	OK but less common
Good for RNN/Transformer?	Bad	Excellent
Uses running stats?	Yes	No
Training = Inference?	No	Yes

## 4. Toy Example Showing BN vs LN

Input matrix:

```
ini
```

 Copy code

```
x = [
    [ 1,  2,  3 ],    # sample 0
    [10, 20, 30 ]    # sample 1
]
shape = (2, 3)
```

### 4.1 BatchNorm (normalize across batch – down columns)

Feature 0: [1, 10] → mean = 5.5

Feature 1: [2, 20] → mean = 11

Feature 2: [3, 30] → mean = 16.5

Variance yields z-scores:

BN(x) becomes:

 Copy code

```
[
    [-1, -1, -1],
    [+1, +1, +1]
]
```

**Interpretation:**

- BN makes *each feature dimension* (column) have zero mean and unit variance.

- Samples are compared **to each other**.
- 

## 4.2 LayerNorm (normalize within each sample – across row)

Sample 0 mean = 2 → normalized: [-1, 0, +1]

Sample 1 mean = 20 → normalized: [-1, 0, +1]

LN(x) becomes:

 Copy code

```
[  
  [-1, 0, +1],  
  [-1, 0, +1]  
]
```

### Interpretation:

- LN makes *each sample* internally normalized.
  - Samples remain independent.
- 

## 5. Summary

### Residual Blocks

- Enable deep networks to learn identity mappings easily.
- Provide stable gradients via skip connections.
- Typically use **two convolution layers** for optimal expressivity.

### BatchNorm

- Normalizes per feature across the mini-batch.
- Important for CNNs.
- Uses running statistics in inference.
- Sensitive to batch size.

### LayerNorm

- Normalizes per sample across features.
  - Ideal for RNNs, LSTMs, Transformers.
  - No running statistics; training and inference identical.
-

If you want, I can turn this into a **PDF**, **LaTeX version**, or **Sphinx-ready note** for your course website.



PDF

Your PDF is ready:

 [Download residual\\_norm\\_notes.pdf](#)

If you'd like a more polished version (LaTeX typesetting, equations, diagrams, section numbering, table of contents), just tell me — I can generate a full textbook-grade PDF.

⟨⟩



yes, full textbook grade PDF

Your full textbook-grade PDF is ready:

 [Download residual\\_norm\\_textbook.pdf](#)

If you'd like:

- LaTeX-style math typesetting (via images or embedded formulas)
- More diagrams (skip connections, BN vs LN axes)
- A chapter-style layout (table of contents, section numbers)
- More examples and exercises
- Integration into your Sphinx course site

...I can generate an even more polished version. ⟨⟩



What is (N C H W ) again?



In deep learning (especially PyTorch and CNNs), a **4-D tensor** that represents an image batch is written as:

 **(N, C, H, W)**

Here is what each letter means:

---

## **N — Batch size**

Number of images (or samples) in the batch.

Examples:

- $N = 32 \rightarrow$  training with 32 images at a time
  - $N = 1 \rightarrow$  a single image
  - $N = 128 \rightarrow$  larger mini-batch
- 

## **C — Channels**

Number of **feature channels** (or color channels):

Examples:

- $C = 1 \rightarrow$  grayscale image
- $C = 3 \rightarrow$  RGB image
- $C = 16 / 32 / 64 / \text{etc.} \rightarrow$  output of convolution layers

After several convolution layers, C grows because the network extracts more feature maps.

---

## **H — Height**

Vertical pixel dimension of the image / feature map.

Examples:

- CIFAR-10  $\rightarrow H = 32$
  - MNIST  $\rightarrow H = 28$
  - After pooling or stride, H becomes smaller.
-

## W — Width

Horizontal pixel dimension.

Examples:

- CIFAR-10  $\rightarrow W = 32$
- MNIST  $\rightarrow W = 28$

Like H, this shrinks after pooling/stride.

---

## Putting it together

**Example 1 — Batch of 32 RGB CIFAR-10 images:**

SCSS

 Copy code

(32, 3, 32, 32)

**Example 2 — Output of first conv layer (16 filters):**

SCSS

 Copy code

(32, 16, 32, 32)

**Example 3 — After max-pool (height/width reduced by 2):**

SCSS

 Copy code

(32, 16, 16, 16)

---

## Summary

Symbol	Meaning	Example
N	Batch size	32 images
C	Channels	RGB=3, conv features=64
H	Height	32 pixels
W	Width	32 pixels

If you'd like, I can also draw a diagram showing how Conv2D transforms (N, C, H, W) step by step.



And (B T F ) for RNN?

Yes — for RNNs, LSTMs, GRUs, and Transformers (sequence models), we use a **3-dimensional tensor**, commonly written as:

 **(B, T, F)**

Here's what each dimension means:

---



## **B — Batch size**

Number of independent sequences processed in parallel.

Examples:

- $B = 32 \rightarrow$  training with 32 sentences/sequences at once
  - $B = 1 \rightarrow$  one sequence (e.g., inference)
- 



## **T — Sequence length (Time steps)**

How many tokens / timesteps / elements in each sequence.

Examples:

- Words in a sentence
- Characters in a string
- Time steps in a timeseries
- Frames in audio/video processing

Examples:

- $T = 10 \rightarrow$  10-word sentence
- $T = 50 \rightarrow$  50-timestep sequence
- $T = \text{varies} \rightarrow$  dynamic-length sequences (often padded)



## F — Feature dimension (Embedding or hidden size)

The vector dimension of **each timestep**.

Examples:

- Word embedding size  $\rightarrow F = 128, 256, 768$ , etc.
- LSTM hidden size  $\rightarrow F = 256, 512$

For a Transformer:

- $F$  = model dimension (e.g., 512 for GPT-2 small, 4096 for GPT-3-style models)



## Put it together

**Example 1 — Batch of 32 sentences, each 20 tokens, each token is a 128-dim vector:**

SCSS

Copy code

(32, 20, 128)

**Example 2 — Output of LSTM with hidden size 256, batch size 16, sequence length 50:**

SCSS

Copy code

(16, 50, 256)

**Example 3 — Transformer token embeddings (batch 8, seq len 128, dimension 768):**

SCSS

Copy code

(8, 128, 768)



## Summary Table

Symbol	Meaning	Example
B	Batch size	32 sequences

Symbol	Meaning	Example
T	Sequence length	20 tokens
F	Feature dimension	embedding/hidden size

## Visual intuition

Imagine a batch of sentences:

```
arduino
```

 Copy code

```
[  
  [ token1 → vector of size F ],  
  [ token2 → vector of size F ],  
  ...  
]
```

Stack **B** of those sentences, each with **T** tokens → you get a 3-D tensor.

If you want, I can also show how LSTM weights interact with (B, T, F), or how Transformers apply attention over  $T \times T$  matrices.

     ...

Can you produce a better toy example for BN vs LN?  
Something like 3 batches , 5 features

Absolutely — here is a **much clearer, larger, more realistic toy example** for comparing **BatchNorm (BN)** versus **LayerNorm (LN)** using:

- **Batch size = 3**
- **Features = 5**

This example is designed to be *lecture-quality*, easy to follow, and clearly illustrates the difference in normalization axes.

## Toy Example

We take a batch of 3 samples, each with 5 features:

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 10 & 20 & 30 & 40 & 50 \\ 7 & 14 & 21 & 28 & 35 \end{bmatrix}$$

Shape: (B=3, F=5)

Let's compute **BatchNorm** and **LayerNorm** step-by-step.

---

### Part 1 — BatchNorm (BN)

BN normalizes *each feature column* across the batch.

So we compute mean and variance per feature:

#### 1. Means (column-wise)

Feature	Values in batch	Mean
0	1, 10, 7	6
1	2, 20, 14	12
2	3, 30, 21	18
3	4, 40, 28	24
4	5, 50, 35	30

#### 2. Variances (column-wise)

$$\sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + (x_3 - \mu)^2}{3}$$

Every feature column has values spaced proportionally, giving:

Feature	Variance	Std ( $\sigma$ )
0	14	$\sqrt{14} \approx 3.742$
1	56	$\sqrt{56} \approx 7.483$
2	126	$\sqrt{126} \approx 11.225$

Feature	Variance	Std ( $\sigma$ )
3	224	$\sqrt{224} \approx 14.966$
4	350	$\sqrt{350} \approx 18.708$

### 3. BN normalized output

$$\hat{x}_{i,f} = \frac{x_{i,f} - \mu_f}{\sigma_f}$$

Computing each row:

**Row 0 (1, 2, 3, 4, 5):**

CSS

 Copy code

$[(1-6)/3.742, (2-12)/7.483, (3-18)/11.225, (4-24)/14.966, (5-30)/18.708]$   
 $\approx [-1.336, -1.337, -1.336, -1.338, -1.336]$

**Row 1 (10,20,30,40,50):**

CSS

 Copy code

$[(10-6)/3.742, (20-12)/7.483, (30-18)/11.225, (40-24)/14.966, (50-30)/18.708]$   
 $\approx [1.069, 1.069, 1.069, 1.069, 1.069]$

**Row 2 (7,14,21,28,35):**

CSS

 Copy code

$[(7-6)/3.742, (14-12)/7.483, (21-18)/11.225, (28-24)/14.966, (35-30)/18.708]$   
 $\approx [0.267, 0.267, 0.267, 0.267, 0.267]$

**BatchNorm Output (rounded)**

$$\text{BN}(x) \approx \begin{bmatrix} -1.34 & -1.34 & -1.34 & -1.34 & -1.34 \\ 1.07 & 1.07 & 1.07 & 1.07 & 1.07 \\ 0.27 & 0.27 & 0.27 & 0.27 & 0.27 \end{bmatrix}$$

👉 Every feature (column) has mean 0, variance 1 across the batch.

👉 Each row is shifted based on its magnitude relative to *other samples*.

LN normalizes *each sample row* independently across features.

Compute mean and variance per row.

---

Row 0: [1, 2, 3, 4, 5]

$$\mu = 3, \quad \sigma = \sqrt{2}$$

LN output:

swift

 Copy code

```
[(1-3)/√2, (2-3)/√2, (3-3)/√2, (4-3)/√2, (5-3)/√2]  
= [-1.414, -0.707, 0, 0.707, 1.414]
```

---

Row 1: [10, 20, 30, 40, 50]

$$\mu = 30, \quad \sigma = \sqrt{200}$$

LN output:

swift

 Copy code

```
[(10-30)/√200, ..., (50-30)/√200]  
= [-1.414, -0.707, 0, 0.707, 1.414]
```

---

Row 2: [7, 14, 21, 28, 35]

$$\mu = 21, \quad \sigma = \sqrt{98}$$