ARTIFICIAL INTELLIGENCE NEEDS REAL INTELLIGENCE

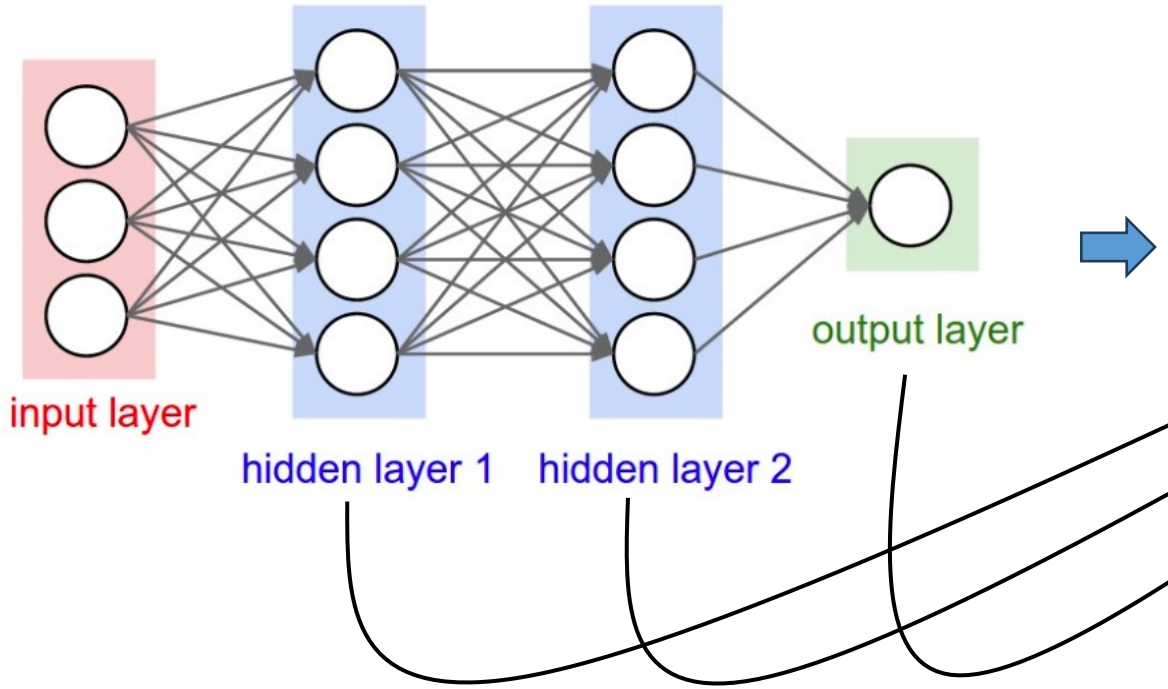
# Convolutional Neural Network I

Professor: Tianyuan ZHANG
tianyuan.zhang@kedgebs.com

KEDGE BUSINESS SCHOOL

# Recap of Previous Session

- **Artificial neural network**




```
# define a custom neural network class
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 4), nn.ReLU(),
            nn.Linear(4, 4), nn.ReLU(),
            nn.Linear(4, 1)
        )

    def forward(self,x):
        return self.net(x)
```

*Non-linear activation function*

*No specified = linear activation function*

# Recap of Previous Session

- **Artificial neural network**

```python
# define a custom neural network class
class Regressor(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 4), nn.ReLU(),
            nn.Linear(4, 4), nn.ReLU(),
            nn.Linear(4, 1)
        )
    def forward(self.x):
        return self.net(x)
```

*For **regression**:*

- ***Linear** activation function for the output layer*
- ***MSE** loss function*

```python
# define a custom neural network class
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 4), nn.ReLU(),
            nn.Linear(4, 4), nn.ReLU(),
            nn.Linear(4, 1), nn.Sigmoid()
        )
    def forward(self.x):
        return self.net(x)
```

*For **binary classification**:*

- ***One** output neuron + **Sigmoid***
- ***Binary cross entropy** loss function*
- *Outputs are **probabilities***

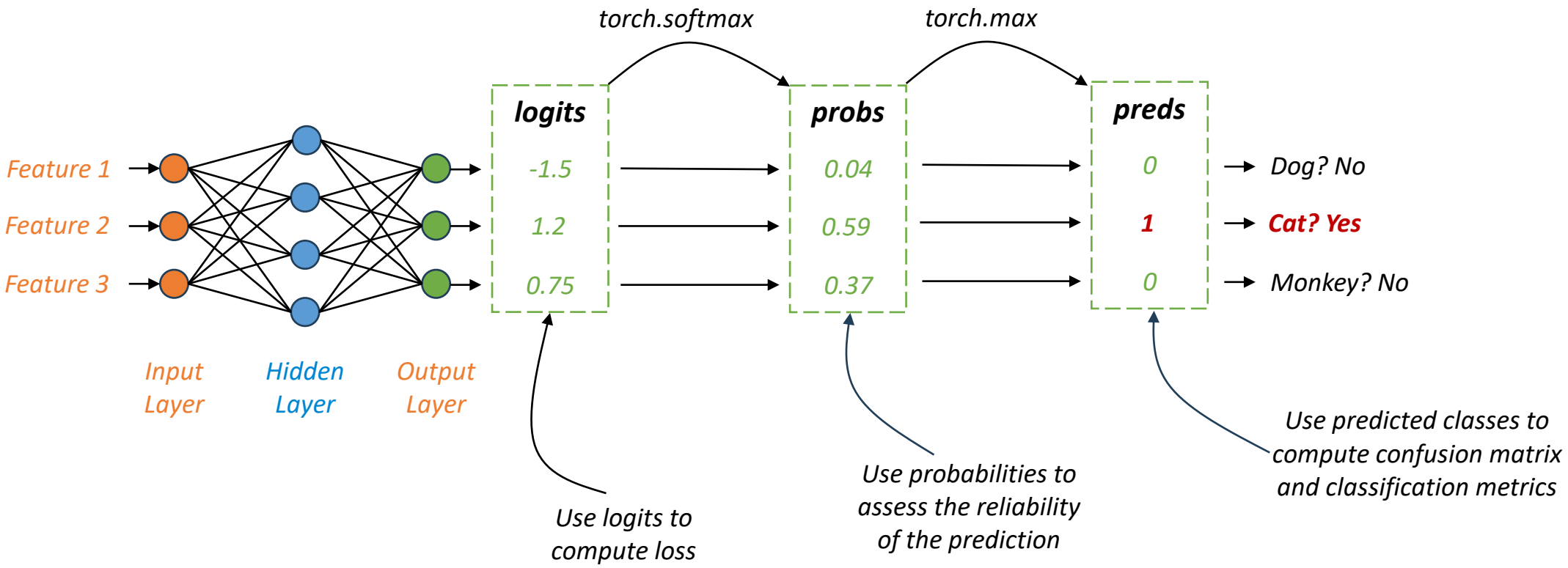
```python
# define a custom neural network class
class MultiClassifier(nn.Module):
    def __init__(self, n_labels):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 4), nn.ReLU(),
            nn.Linear(4, 4), nn.ReLU(),
            nn.Linear(4, n_labels)
        )
    def forward(self.x):
        return self.net(x)
```

*For **multi-class classification**:*

- ***Multiple** output neurons + **Linear** activation function*
- ***Cross entropy** loss function*
- *Outputs are **logits***

# Recap of Previous Session

- **Logits vs. Probabilities vs. Predicted classes**



# Recap of Previous Session

- **Regularization** techniques
  - To prevent the network from **over-fitting** and ensure the generalization
  - **Weight decay**
    - Add a penalty term (L2 norm of weights) to the loss function
    - Penalize the complexity of the network
    - `torch.optim.SGD(weight_decay)`
  - **Learning rate decay**
    - Start training with a relative high learning rate
    - Reduce learning rate when the validation loss stops improving
    - `troch.optim.lr_scheduler.ReduceLROnPlateau()`
  - **Early stopping**
    - Early stop when the validation loss stops improving

# Recap of Previous Session

- **Regularization** techniques
  - To prevent the network from **over-fitting** and ensure the generalization
  - **Dropout**
    - Randomly drop some neurons to reduce the complexity of the network in each training step
    - Add dropout as a layer when specifying the network structure
    - `torch.nn.Dropout(p)`
  - **Batch normalization**
    - Normalize the inputs of a layer by re-centering and re-scaling for each batch
    - Add batch normalization as a layer when specifying the network structure
    - Batch normalization layer has two types of learning parameters
    - `torch.nn.BatchNorm1d()`

# Recap of Previous Session

- **Regularization** techniques

```python
# define a custom neural network class
class MultiClassifier(nn.Module):
    def __init__(self, n_labels):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 4),
            nn.BatchNorm1d(4),
            nn.ReLU(),
            nn.Dropout(p = 0.5),
            nn.Linear(4, n_labels)
        )
    def forward(self.x):
        return self.net(x)
```

*Batch normalization layer is positioned before the activation function layer*

*Dropout layer is positioned after the activation function layer*

# Recap of Previous Session

- **Implement an ANN**
  1. Build the data pipeline
     - torch.utils.data.Dataset
     - torch.utils.data.DataLoader
  2. Create the artificial neural network
     - Define a custom neural network class
  3. Training by gradient descent
     - Define a train( ) function
  4. Save and load a trained model
  5. Make predictions and evaluation
     - Define a test( ) function

- *Network structure & activation functions*
- *Forward propagation method*
- *Batch normalization & Dropout*

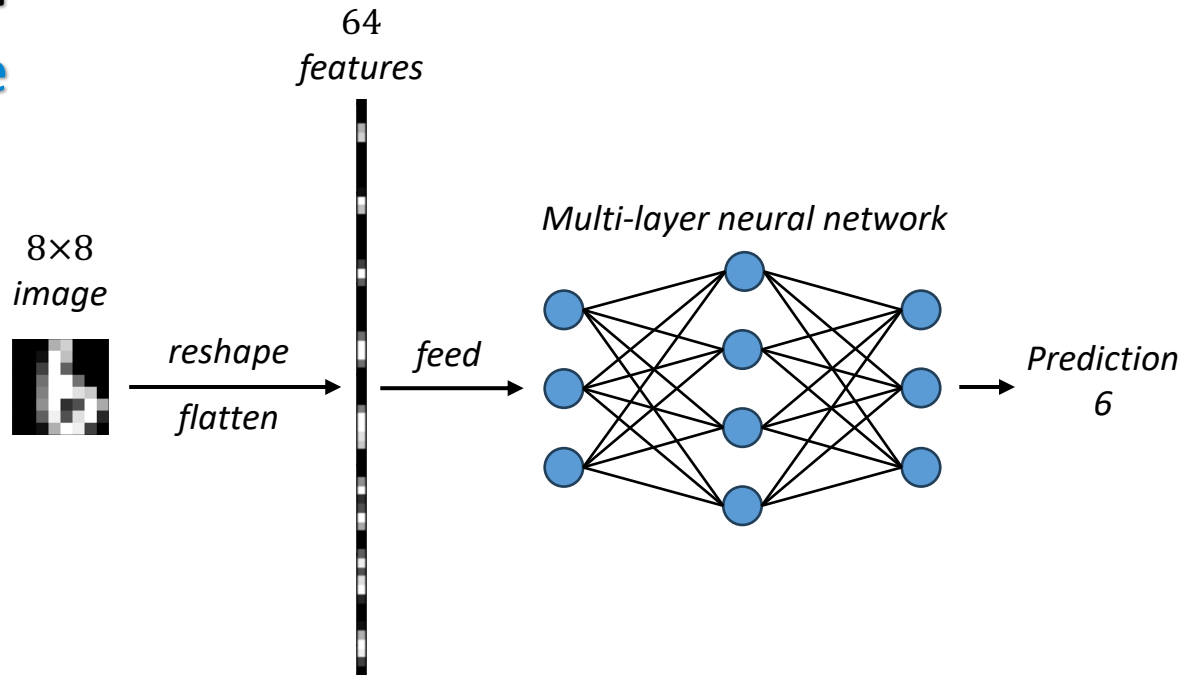
- *Loss function*
- *Optimizer (weight decay)*
- *Learning rate (learning rate decay)*
- *Number of epochs to train (early stopping)*

# Outline

- **Motivation of CNN**
- Convolutional layer
- Pooling layer
- Fully connected layer
- CNN architecture

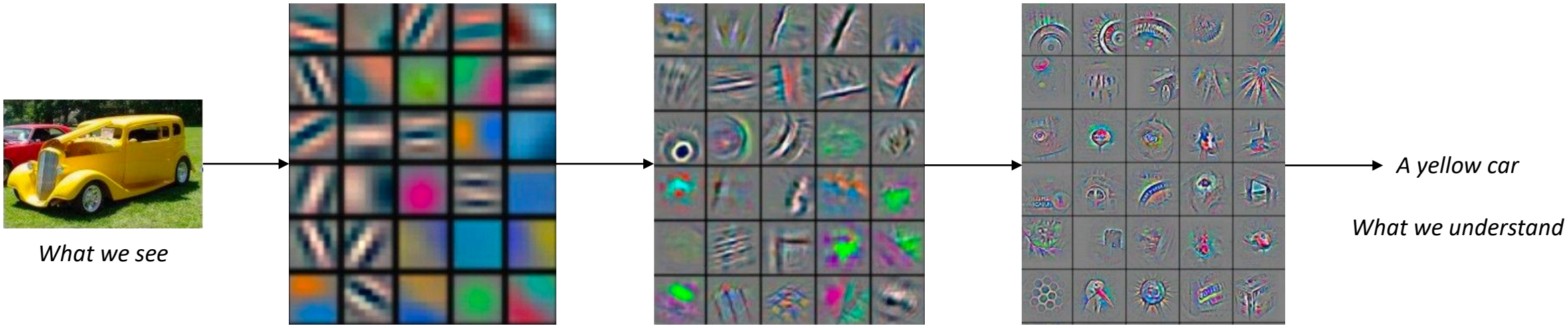
# Motivation of CNN

- Image classification with traditional ANN
  - Each pixel in the image is treated as a single feature
    - Parameter explosion
      - $256 \times 256$ pixels lead to 65,536 features
      - 65,536 connections per neuron in the 1st hidden layer
  - Ignoring the spatial structure / patten
    - A vertical line may indicate 1 or 7
    - A circle may indicate 0 or 6 or 8 or 9
  - **Our vision doesn't treat images as 1D flat vectors**



# Motivation of CNN

- How human vision works? → **Visual cortex**




- Neurons focus on a small region to detect local spatial features
- Visual cortex organized in layers to aggregate features hierarchically

# Motivation of CNN

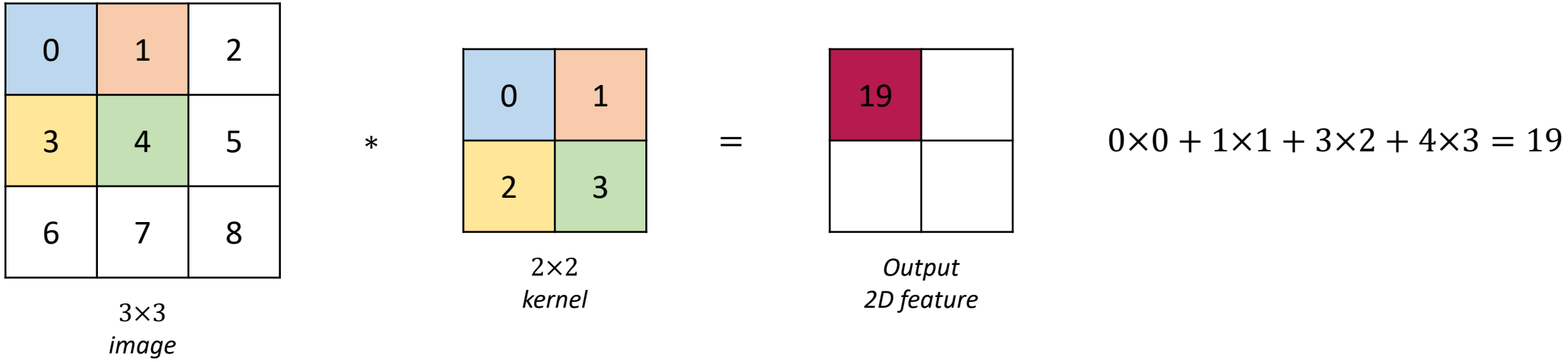
- How can artificial neural networks mimic human vision?
  - Neurons focus on a small region to detect local spatial features
    - Feature extraction with **convolutional kernels / filters** → **Convolutional layer**
  - Visual cortex organized in layers to aggregate features hierarchically
    - Multiple convolutional layers → **Convolutional neural network**

# Outline

- Motivation of CNN
- **Convolutional layer**
- Pooling layer
- Fully connected layer
- CNN architecture

# Convolutional layer

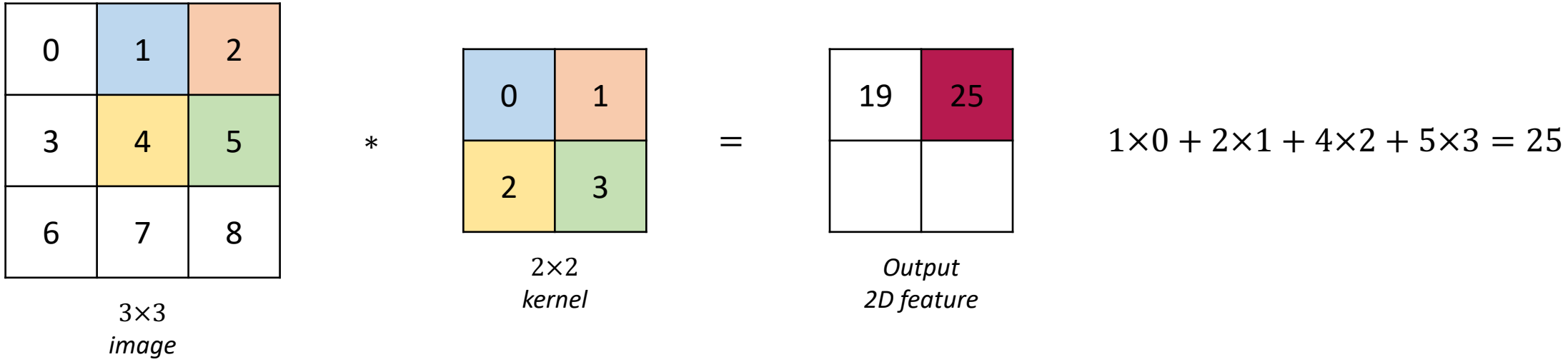
- **2D convolution operation**
  - Applying a kernel or filter to a 2D matrix, denoted by operator $*$




  - Computing the sum of element-wise products in the kernel window

# Convolutional layer

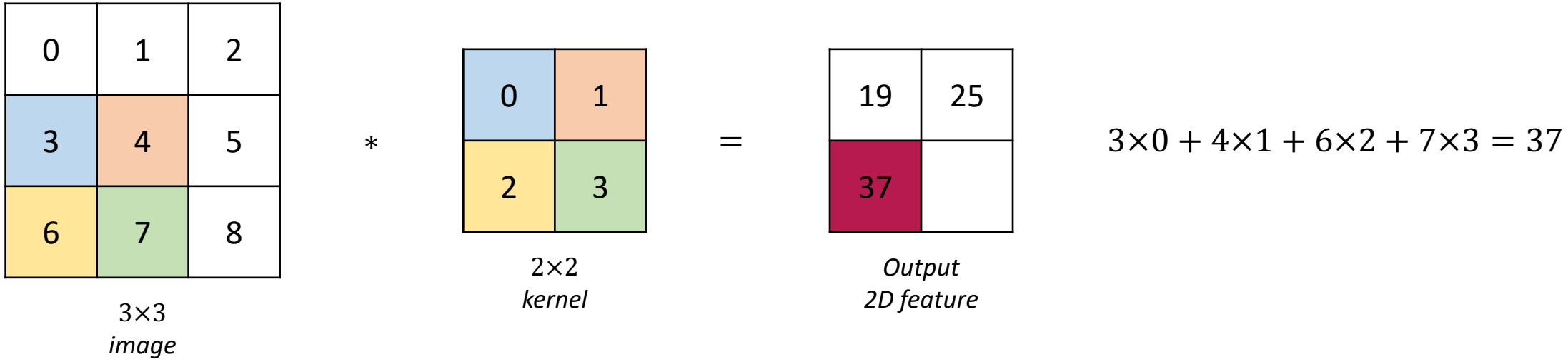
- **2D convolution operation**
  - Applying a kernel or filter to a 2D matrix, denoted by operator $*$




  - Computing the sum of element-wise products in the kernel window
  - Slide the kernel window across the input matrix from left to right

# Convolutional layer

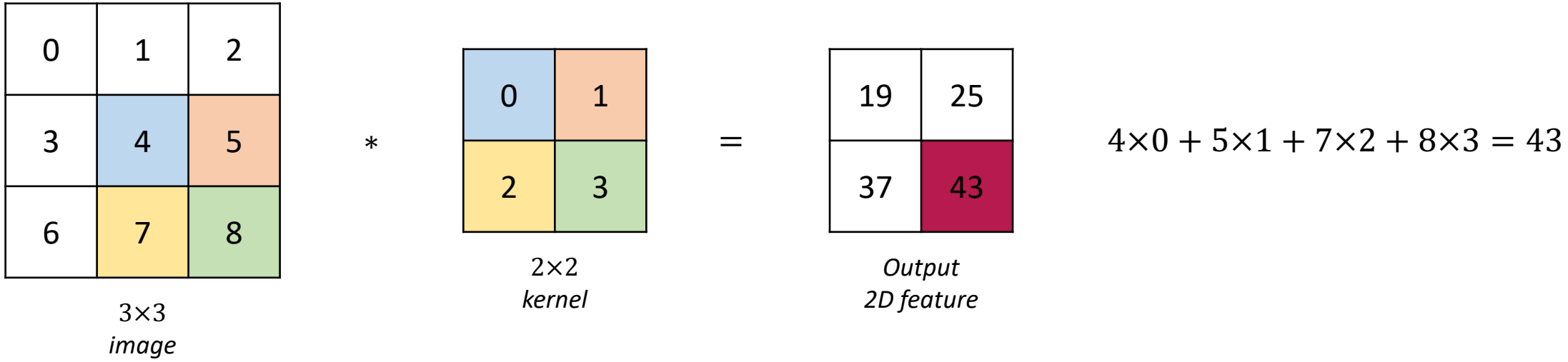
- **2D convolution operation**
  - Applying a kernel or filter to a 2D matrix, denoted by operator $*$




  - Computing the sum of element-wise products in the kernel window
  - Slide the kernel window across the input matrix from left to right, up to down

# Convolutional layer

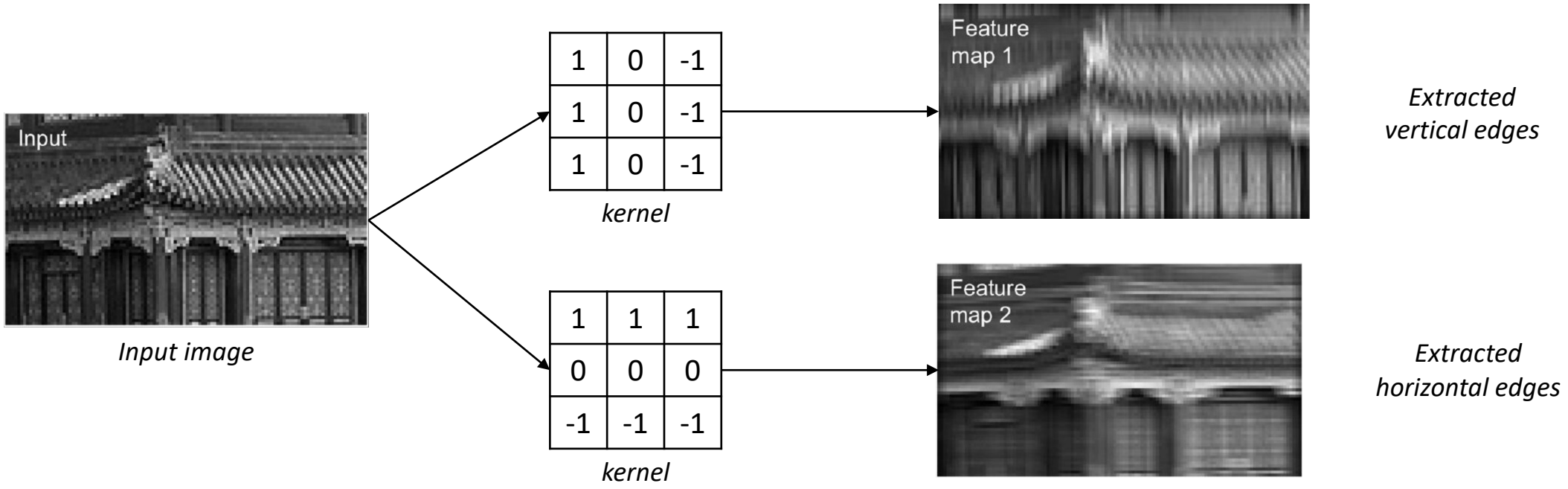
- **2D convolution operation**
  - Applying a kernel or filter to a 2D matrix, denoted by operator $*$




  - Computing the sum of element-wise products in the kernel window
  - Slide the kernel window across the input matrix from left to right, up to down

# Convolutional layer

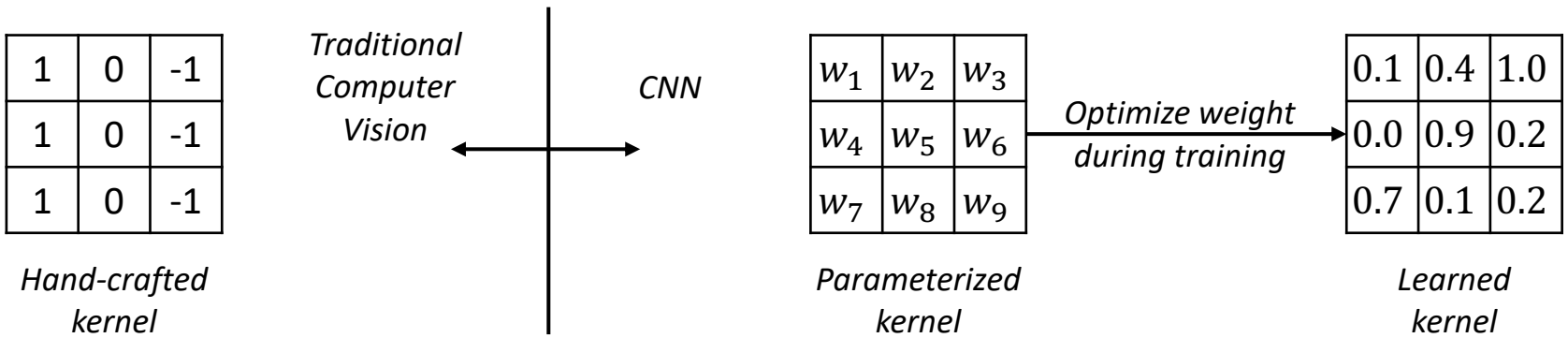
- How convolution operation extracts feature?




- Each convolutional kernel can extract a specific feature as a 2D matrix
  - So called **feature map**

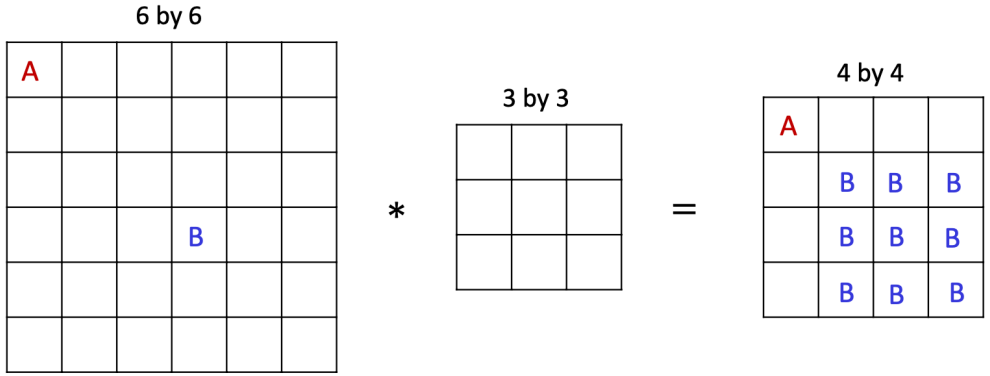
# Convolutional layer

- How to obtain the convolutional kernels?
  - Before CNN
    - Hand-crafted kernels
    - Design the kernels for feature extraction is an important topic in traditional computer vision
  - The CNN era
    - **Kernels are treated as learning parameters**
    - The best kernels are learned automatically in the training process



# Convolutional layer

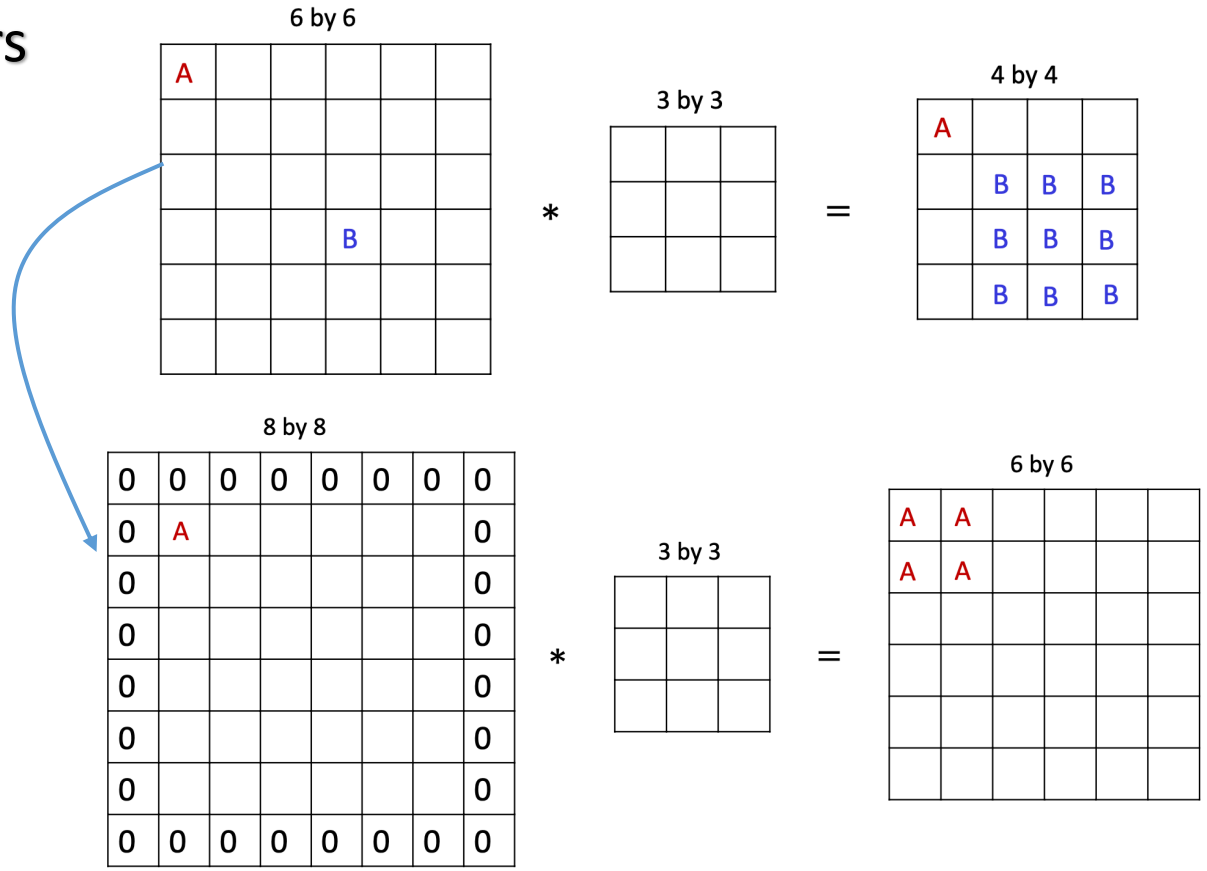
- **2D convolution operation**
  - Given an input image and a kernel, suppose:
    - The input size is $H_{in} \times W_{in}$
    - The kernel size is $K \times K$
    - Then, the output size $H_{out} \times W_{out}$ is:
      - $H_{out} = H_{in} - K + 1$
      - $W_{out} = W_{in} - K + 1$




  - If the kernel size $K > 1$, **the input image shrinks after convolution**
    - If there are many successive convolution operations, we end up with a very small output
  - **Pixels on the corners or edges are used much less** than pixels in the middle
    - The information from the edges of the image are throwed away.

# Convolutional layer

- **Padding**
  - Padding the image with additional borders
  - **Zero padding**
    - Set pixel values to 0 on the border
  - After padding
    - Pixels on the corners or edges are used more frequently than before padding



# Convolutional layer

- **Padding**
  - We can decide how many pixels used as the padding borders
    - Usually set $P$ to 1, if $K = 3$, then
    - $H_{out} = H_{in} - K + 2P + 1 = H_{in}$
    - $W_{out} = W_{in} - K + 2P + 1 = W_{in}$
    - The output feature map has the same size as the input image

*Original input*

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

*Zero padding, P = 1*

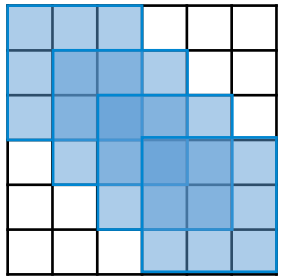
| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

*Zero padding, P = 2*

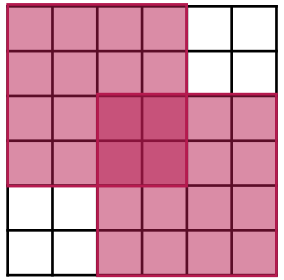
| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Convolutional layer

- **Stride**
  - For 2D convolution operation, the kernel window slides across the input matrix
  - Stride controls how far the kernel window slides at each step
  - We refer to the number of rows and columns traversed per slide as stride
  - By default, $S = 1$
  - When the kernel size is large, and we want less overlaps, we can set a larger stride



*K = 3*
*S = 1*



*K = 4*
*S = 2*

# Convolutional layer

- **2D convolution operation summary**
  - Input size: $H_{in} \times W_{in}$
  - Convolutional kernel:
    - Kernel size: $K \times K$
    - Zero padding: $P$
    - Stride: $S$
  - Output size: $H_{out} \times W_{out}$
    - $H_{out} = \frac{H_{in} - K + 2P}{S} + 1$
    - $W_{out} = \frac{W_{in} - K + 2P}{S} + 1$

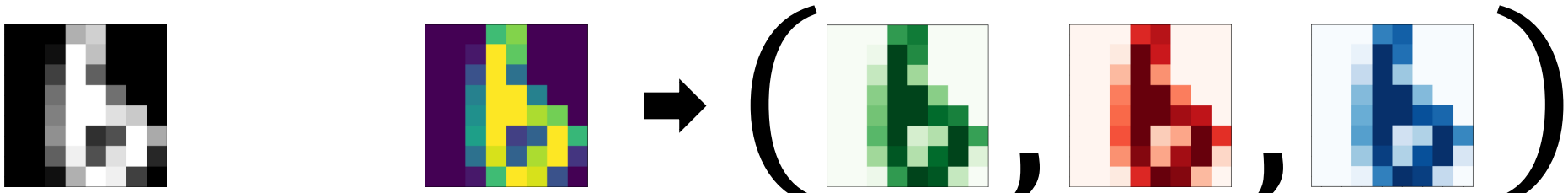
*Common practice:*

- *K = 3*
- *P = 1*
- *S = 1*

*The output size is the same as input*

# Convolutional layer

- **Convolution over channels**
  - For a grayscale image, there is only one channel
    - Size: $1 \times Height \times Width$
  - For a color image, there are three channels: green, red, blue
    - Size: $3 \times Height \times Width$



$1 \times 8 \times 8$
*Grayscale*
*N_channels = 1*

$3 \times 8 \times 8$
*RGB*
*N_channels = 3*

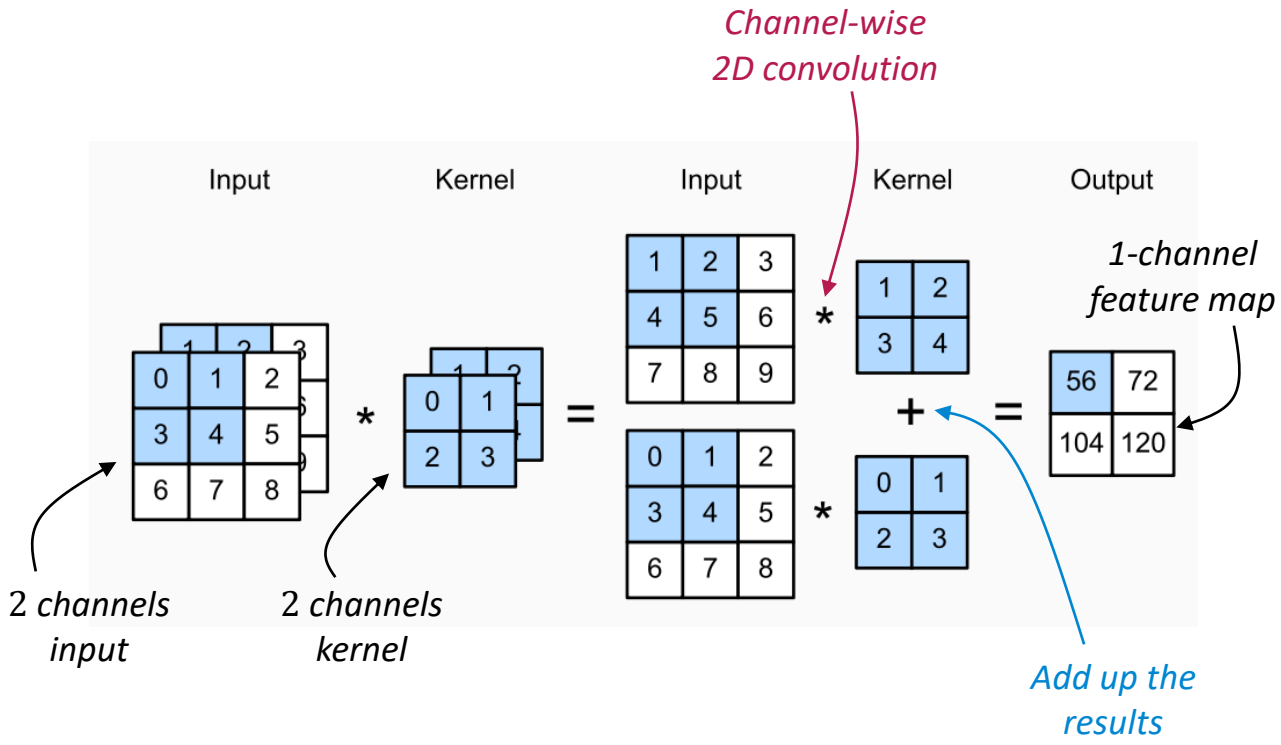
$8 \times 8$
*Green channel*

$8 \times 8$
*Red channel*

$8 \times 8$
*Blue channel*

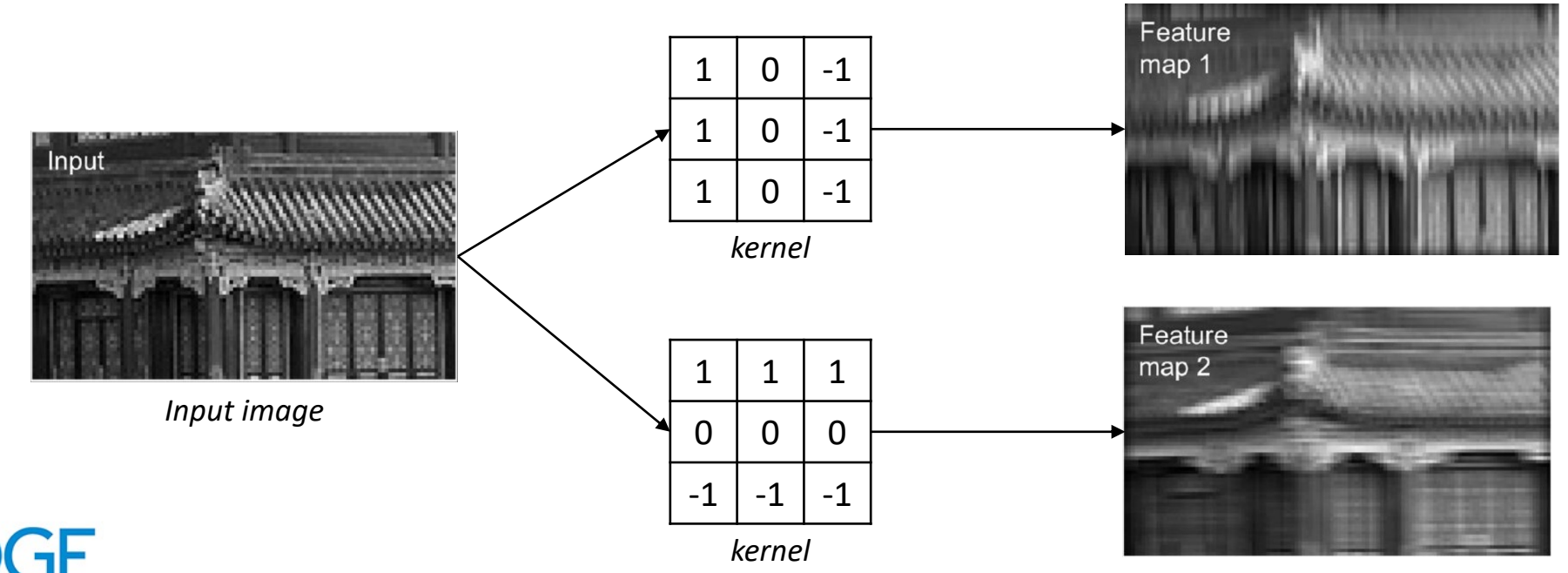
# Convolutional layer

- **Convolution over channels**
  - For an input with multiple channels
    - Size: $Channel_{in} \times Height_{in} \times Weight_{in}$
  - We also need a multi-channel convolutional kernel
    - Size: $Channel_{in} \times K \times K$
  - The convolutional operation:
    1. 2D convolution per input channel
    2. Add up the results as the output
  - The output feature map only has one channel
    - Size: $1 \times Height_{out} \times Weight_{out}$



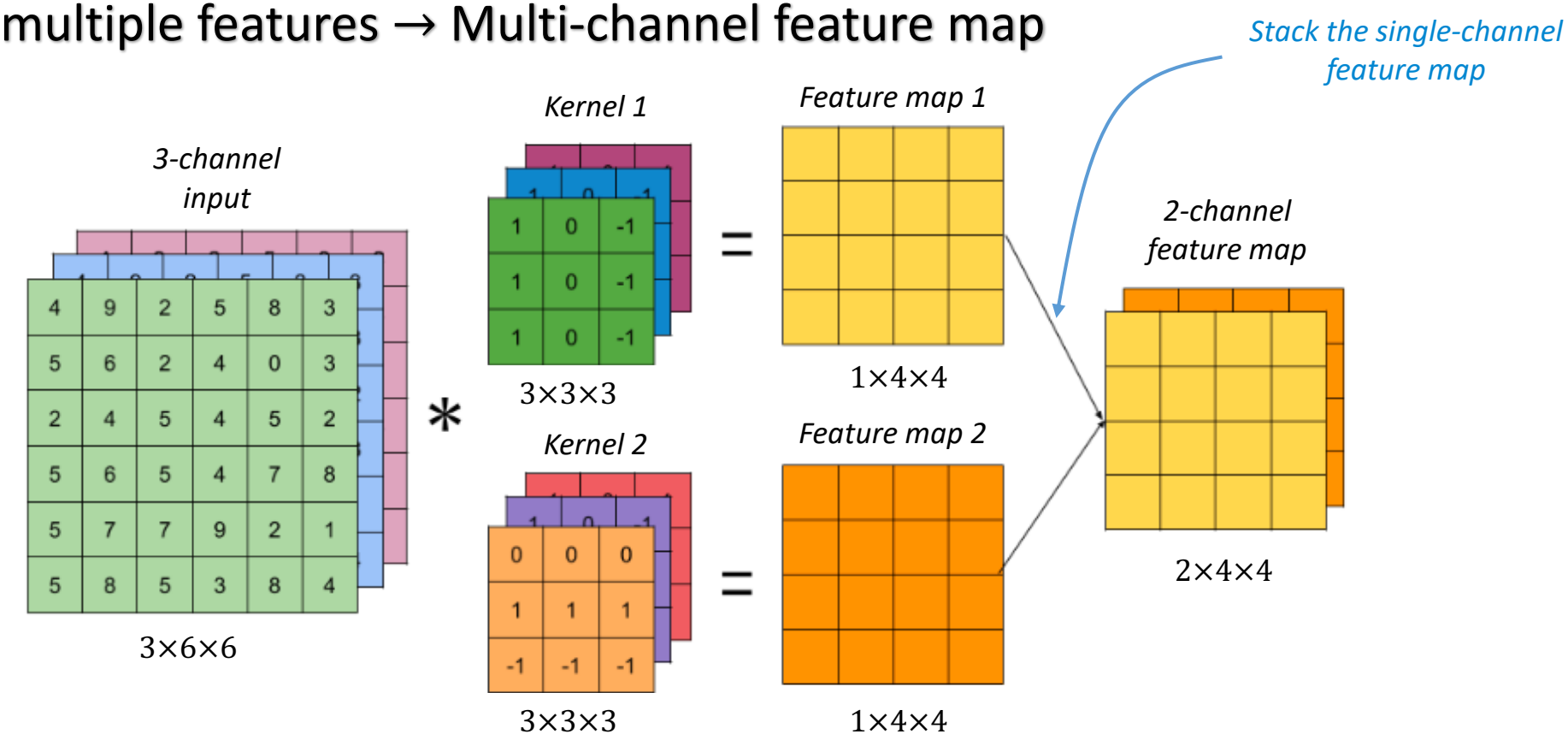
# Convolutional layer

- **Convolution over channels**
  - The output feature map only has one channel
    - Size: $1 \times Height_{out} \times Weight_{out}$
    - One output channel can be regarded as one extracted feature



# Convolutional layer

- **Convolution over channels**
  - We need multiple features → Multi-channel feature map



# Convolutional layer

- **Convolution over channels**
  - We need multiple features → Multi-channel feature map
    - For an input with multiple channels
      - Size: $Channel_{in} \times Height_{in} \times Weight_{in}$
    - We need more than one multi-channel kernel
      - Size: $Channel_{out} \times Channel_{in} \times Height_{in} \times Weight_{in}$
    - Then the output feature map has multiple channel
      - Size: $Channel_{out} \times Height_{out} \times Weight_{out}$
  - Input channels
    - The number of channels of the input image (or the output feature map of previous layer)
  - Output channels
    - The number of convolutional kernels, the number of channels of the output feature map

# Convolutional layer

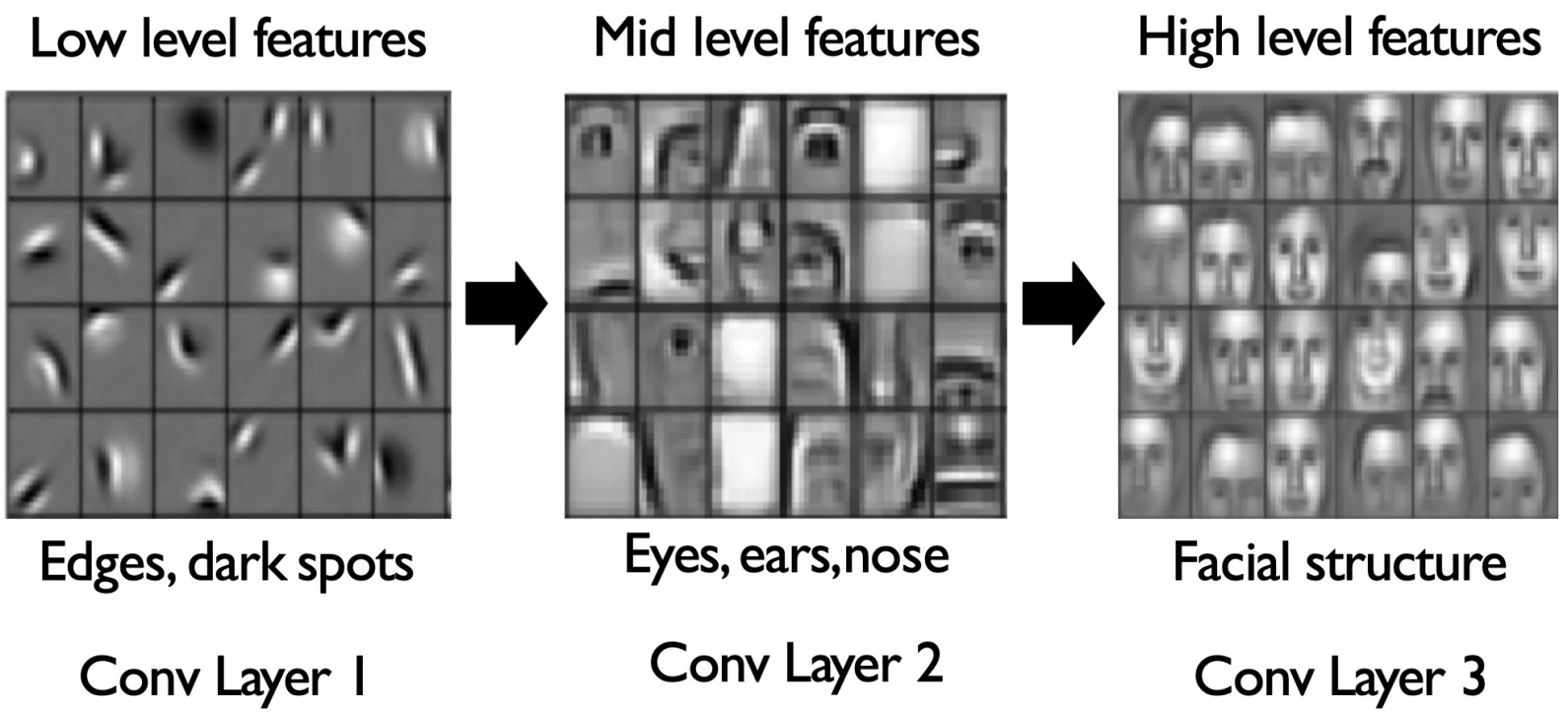
- **A convolutional layer**
  - in_channels
  - out_channels
  - kernel_size
  - stride, default: 1
  - padding, default: 0
  - bias, default: True
    - If true, add a bias to the output feature map per output channel

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
      dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)
```

*No padding by default*

# Convolutional layer

- **Convolutional neural network**
  - A stack of multiple convolutional layers to extract and aggregate feature hierarchically

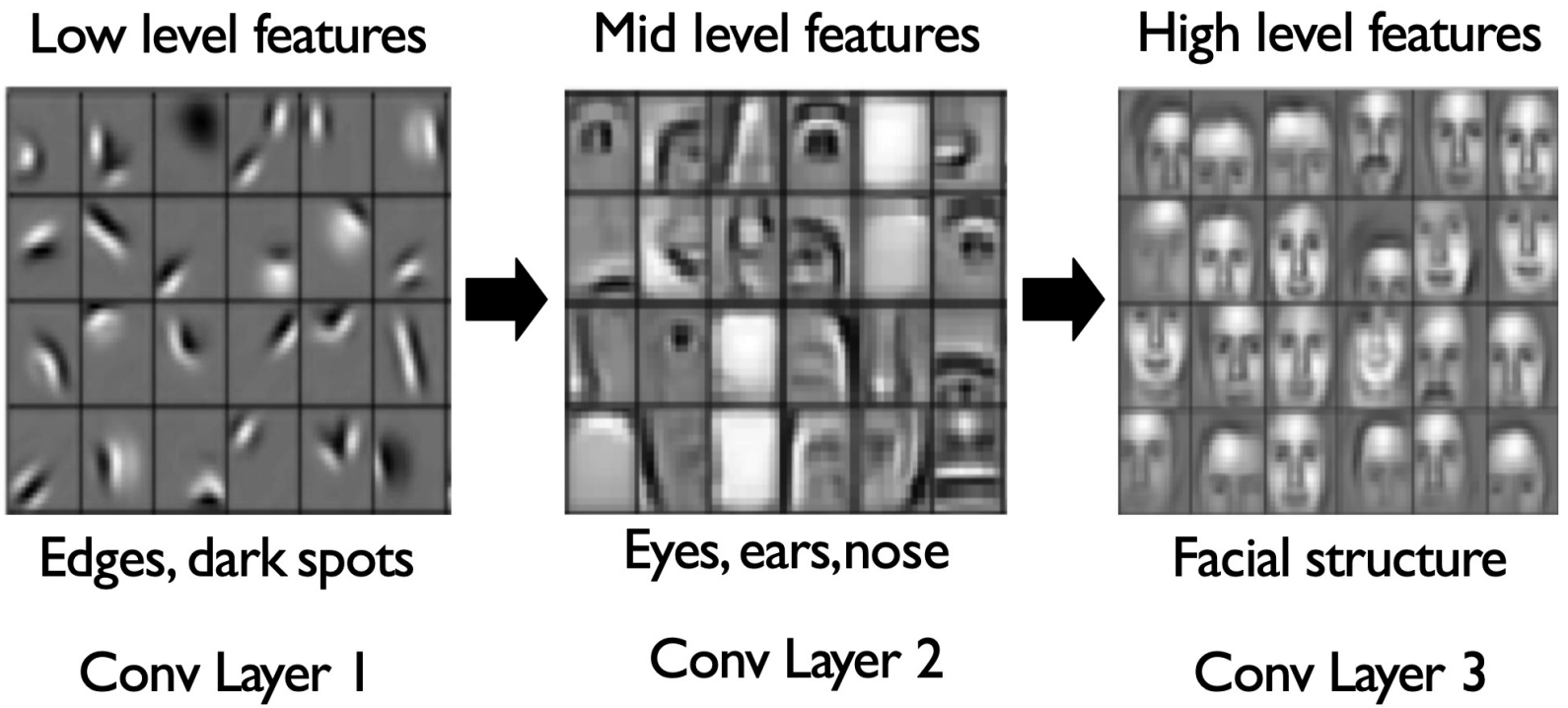


# Outline

- Motivation of CNN
- Convolutional layer
- **Pooling layer**
- Fully connected layer
- CNN architecture

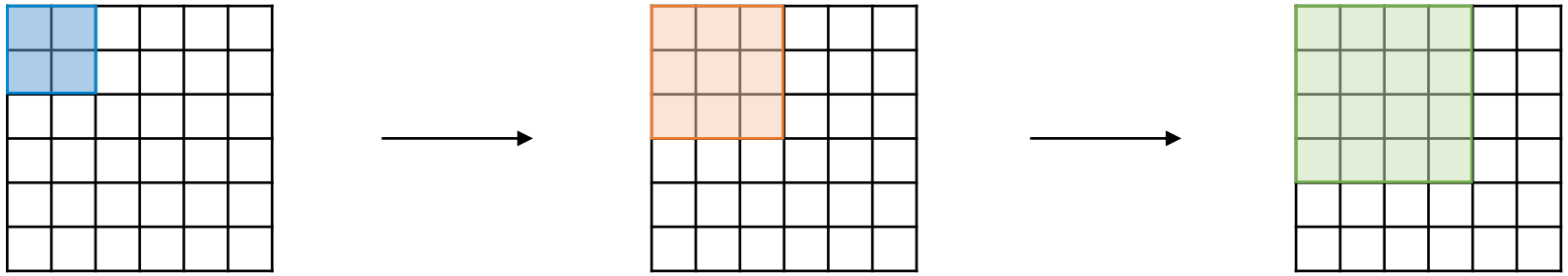
# Pooling layer

- **Motivation of pooling**
  - Low → High level features represent Local → Global spatial pattern / structure



# Pooling layer

- **Motivation of pooling**
  - Low → High level features represent Local → Global spatial pattern / structure
  - To extract high-level features when the network go deeper:
    - We can increase the kernel size gradually to consider more global spatial pattern



*K = 2*
*Low-level*

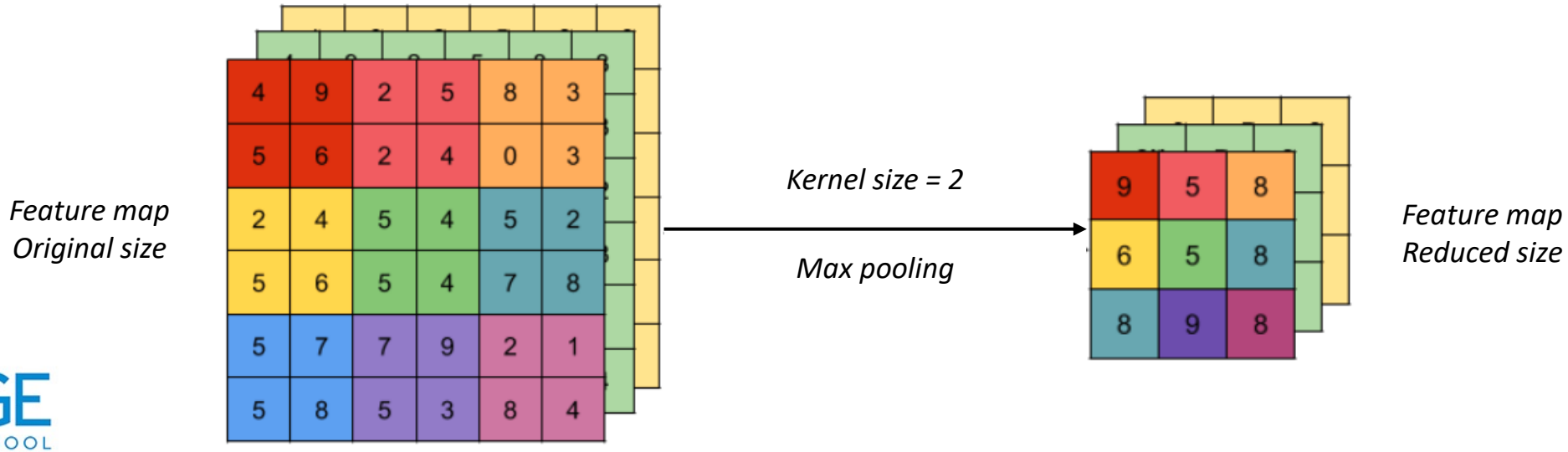
*K = 3*
*Mid-level*

*K = 4*
*High-level*

  - **Or we can reduce the size of the feature map**
    - The subsequent convolutional layer can see a larger area with the same kernel window size

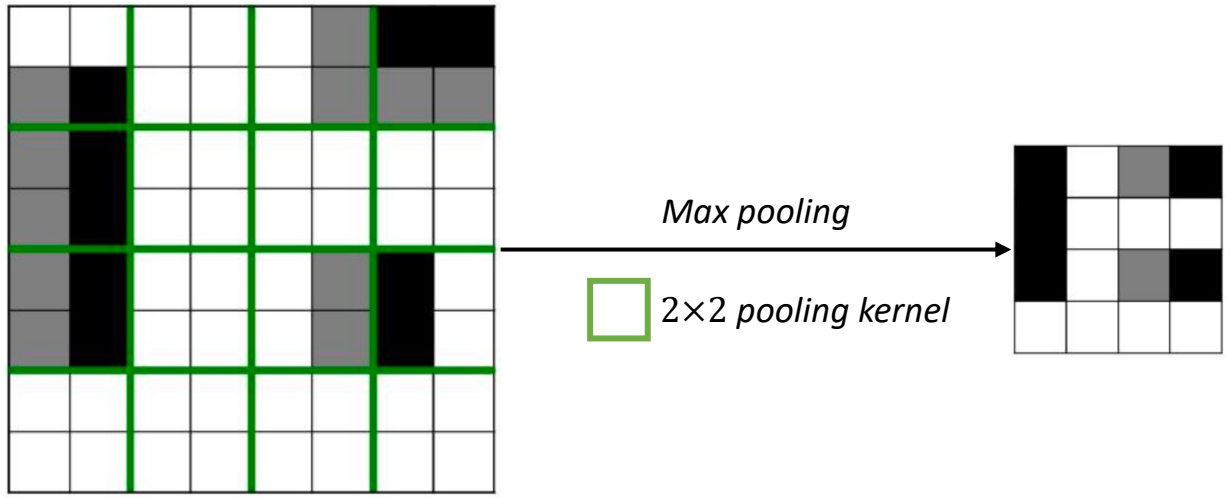
# Pooling layer

- **Max pooling**
  - Like convolution, pooling has a fixed-size window sliding across the input
    - For convenience, we also refer to the window size as the kernel size
  - Unlike convolution that computes the sum of element-wise product
    - Max pooling directly output the max value in the window
  - Pooling doesn't change the number of channels of the feature map



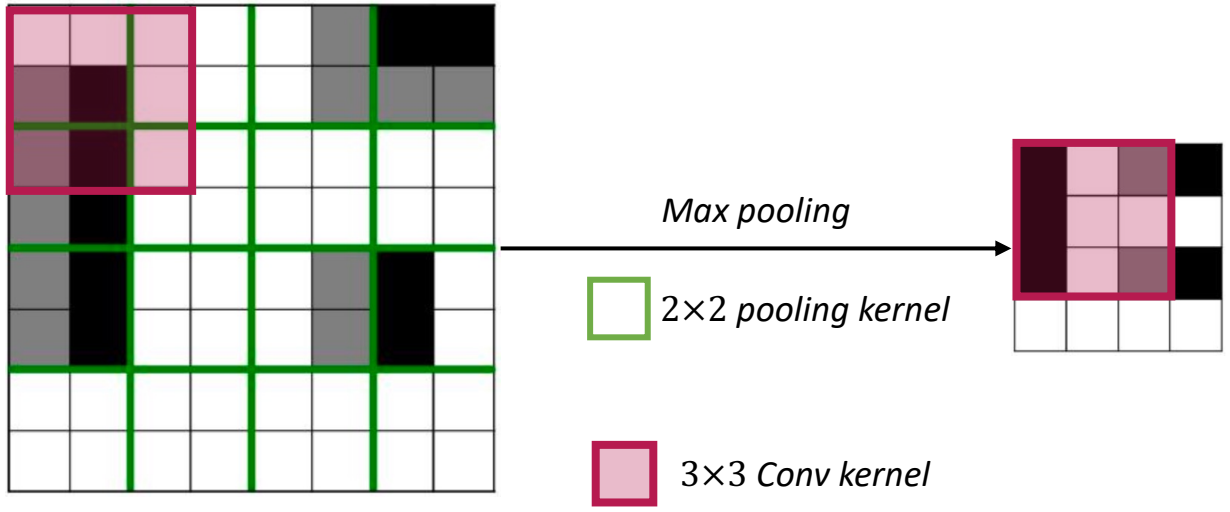
# Pooling layer

- **Max pooling**
  - Max pooling can preserve most of the global spatial information while ignore some of the local spatial pattern



# Pooling layer

- **Max pooling**
  - Max pooling can preserve most of the global spatial information while ignore some of the local spatial pattern
  - After pooling, the convolutional kernel can see a larger region to extract more global features



# Pooling layer

- **Max pooling**
  - Stride:
    - By default, stride is equal to the kernel size
    - No overlap when sliding across the feature map
  - Padding
    - By default, there is no padding

MAXPOOL2D

CLASS torch.nn.MaxPool2d(*kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False*) [SOURCE]

# Outline

- Motivation of CNN
- Convolutional layer
- Pooling layer
- **Fully connected layer**
- CNN architecture

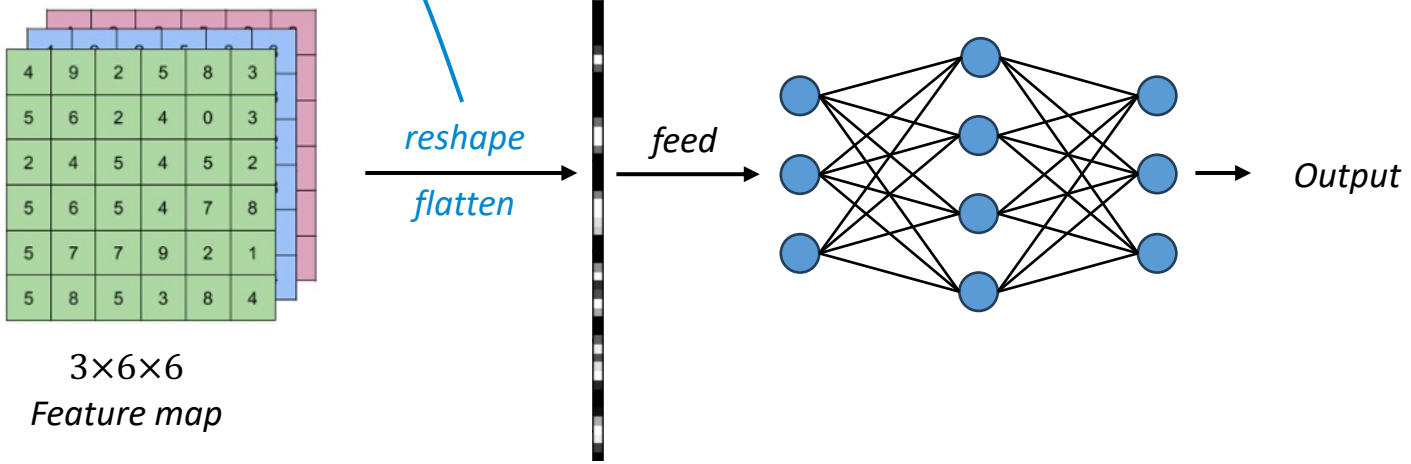
# Fully connected layer

- Conv layer (+ Pooling layer) can output multi-channel feature map
- To use this feature map, we need to reshape it into 1D feature vector

TORCH.FLATTEN

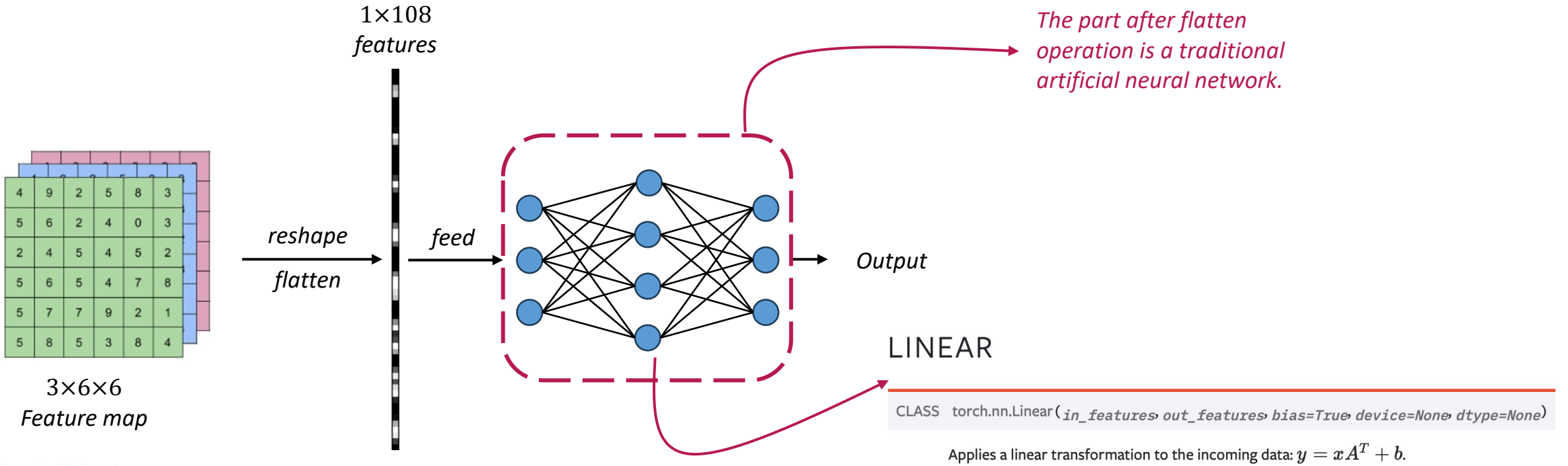
```
torch.flatten(input, start_dim=0, end_dim=-1) → Tensor
```

*Flatten input feature map by reshaping it into a one-dimensional tensor.*



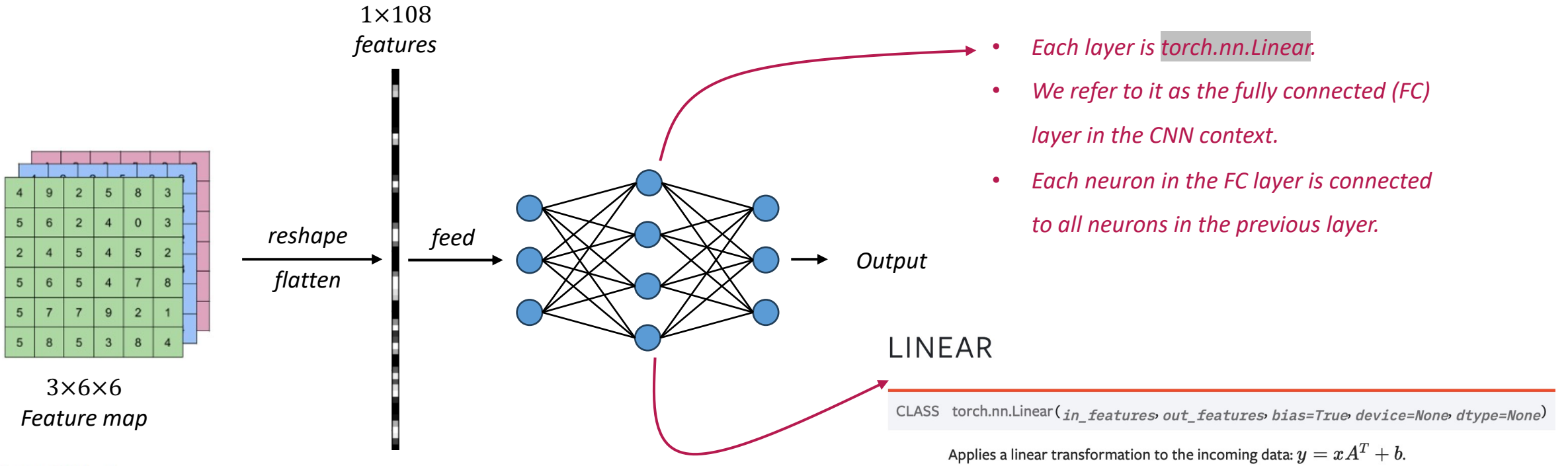
# Fully connected layer

- Conv layer (+ Pooling layer) can output multi-channel feature map
- To use this feature map, we need to reshape it into 1D feature vector



# Fully connected layer

- Conv layer (+ Pooling layer) can output multi-channel feature map
- To use this feature map, we need to reshape it into 1D feature vector

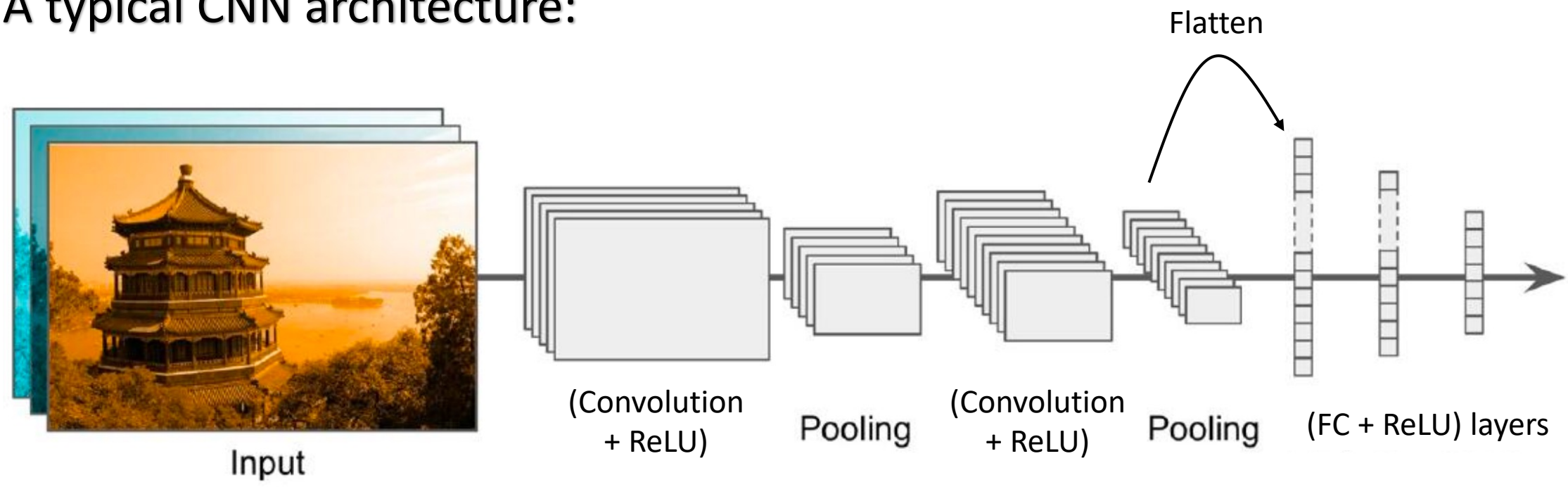


# Outline

- Motivation of CNN
- Convolutional layer
- Pooling layer
- Fully connected layer
- **CNN architecture**

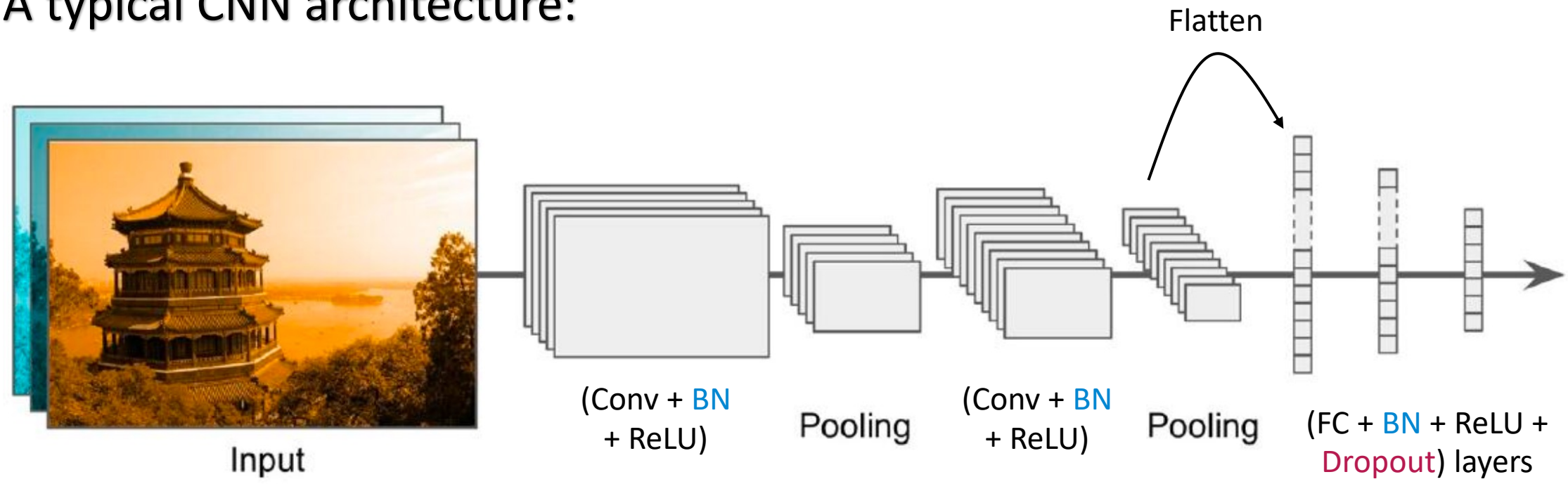
# CNN architecture

- A typical CNN architecture:



# CNN architecture

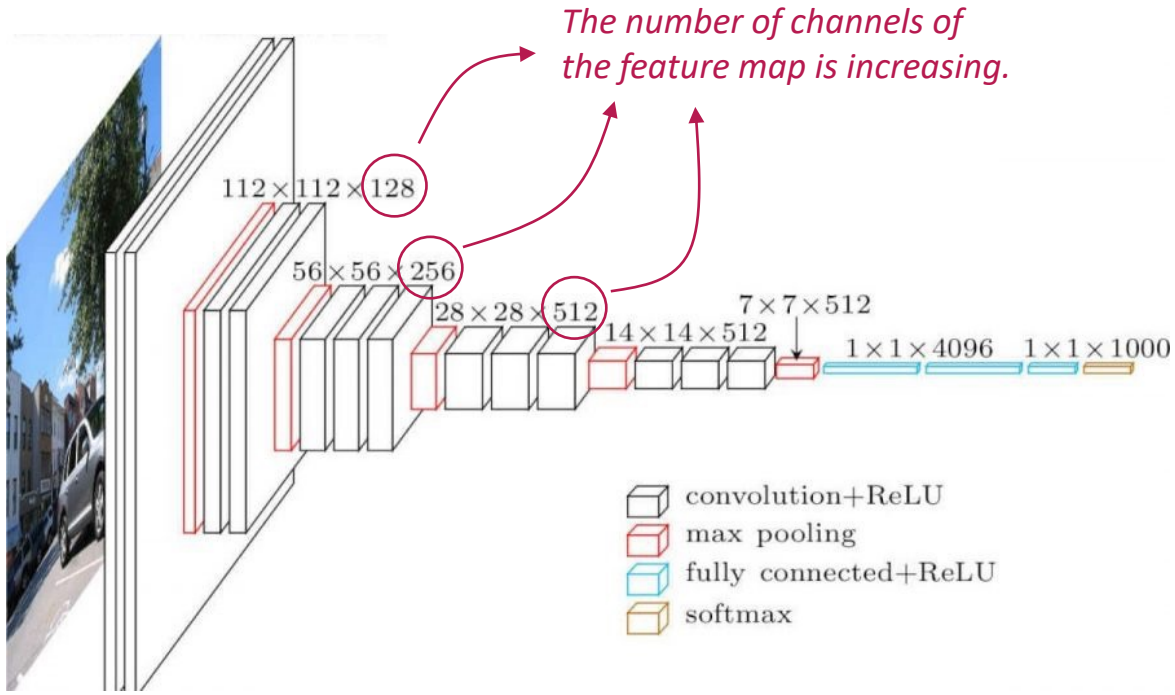
- A typical CNN architecture:




- We can include the Batch Normalization (BN) and Dropout layers in CNN

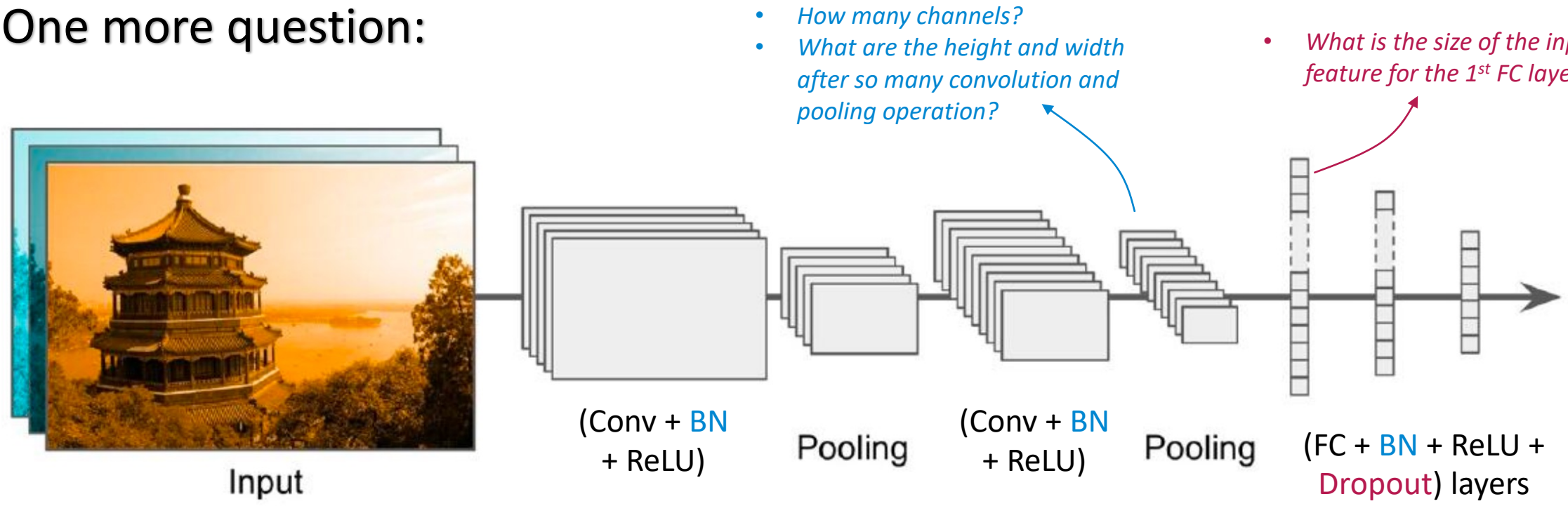
# CNN architecture

- Most of the CNNs follow a so-called **pyramid architecture**:
  - As we go deeper in the CNN
    - **The size of feature map is shrinking** by convolution and pooling layers
      - Low-level feature → High-level feature
      - Local spatial pattern → Global spatial pattern
    - The number of channels of feature map is increasing
      - **Increase the number of convolutional kernels**
      - Corresponds to more possible combinations of simple features



# CNN architecture

- One more question:




- We need to **calculate the size of the last feature map** before flatten

# CNN architecture

- One more question:
  - We need to **calculate the size of the last feature map** before flatten
  - Let PyTorch handles this issue:
    - There is a type of layers called the **Lazy layers**
      - `torch.nn.LazyLinear`
      - `torch.nn.LazyConv2d`
      - `torch.nn.LazyBatchNorm1d` & `torch.nn.LazyBatchNorm2d`
    - **We don't need to specify the size of input for Lazy layers**
      - The lazy layer will infer the input size automatically when we first feed data to the network.
      - Thanks to the lazy layers, we can be lazy.

# Hands-on Exercise

- **Exercise 06 CNN for Image Classification - Instruction**
- Exercise 06 CNN for Image Classification - Assignment