Convolutional Neural Networks:

6.1. Fully Connected Layers to Convolutions
6.2. Convolutions for Images
6.3. Padding and Stride

Annie Moore

Outline

- Introduction to Convolutional Neural Networks
  - What are they?
  - Defining a fully connected layer
  - Methods to calculate weights
- Feature maps and Receptive Fields
- Padding and Stride

Introduction to Convolutional Neural Networks

- Options discussed previously are good for tabular data
- CNNs maintain spatial structure when analyzing image data
  - will leverage knowledge that nearby pixels are related to each other
- Computationally efficient
  - Require fewer parameters than a fully-connected architecture
  - Easy to parallelize
- Are sometimes also used for 1-D or graph structured data
  - audio and text

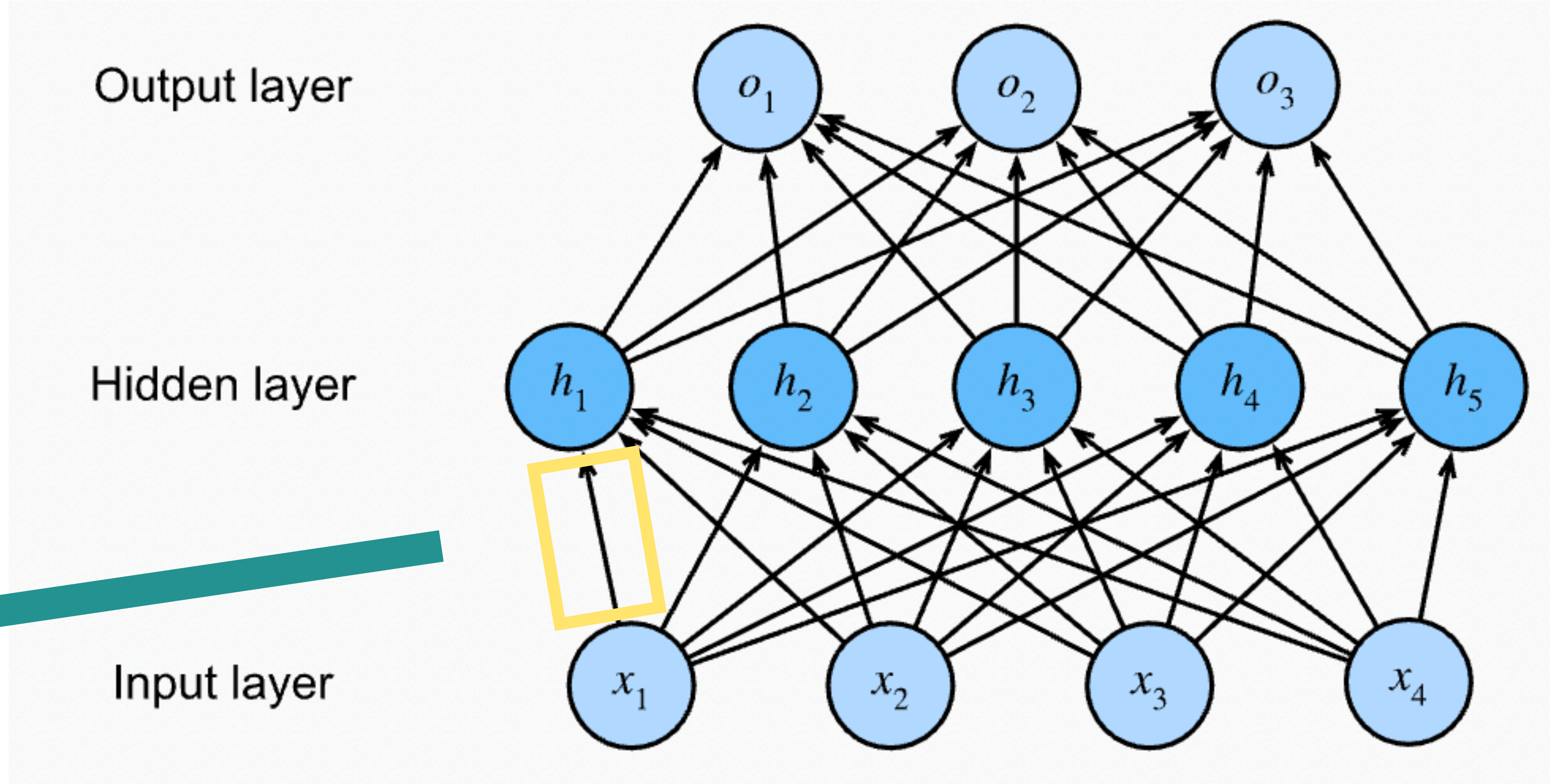
Spatial Invariance

- Should not be concerned with the precise location of the object (spatial invariance)
  - Sweep patches and decide if they contain what we are looking for
  - Hidden layer representations should peak where desired attribute is the highest
  - can be used to learn useful representations with fewer parameters
- Properties of natural signals in images guide the design of the architecture:
  - **In the earliest layers, the network responds similarly to the same patch, regardless of where it appears (translational invariance)**
  - **Earliest layers should focus on correlations within local regions (locality principle)**

Defining the Multi-Layer Perceptron

- Start by thinking of an MLP with 2-D inputs **X** and hidden layers **H**
  - inputs and hidden layers have spatial structure
  - weights are now 4th order tensors

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l}$$

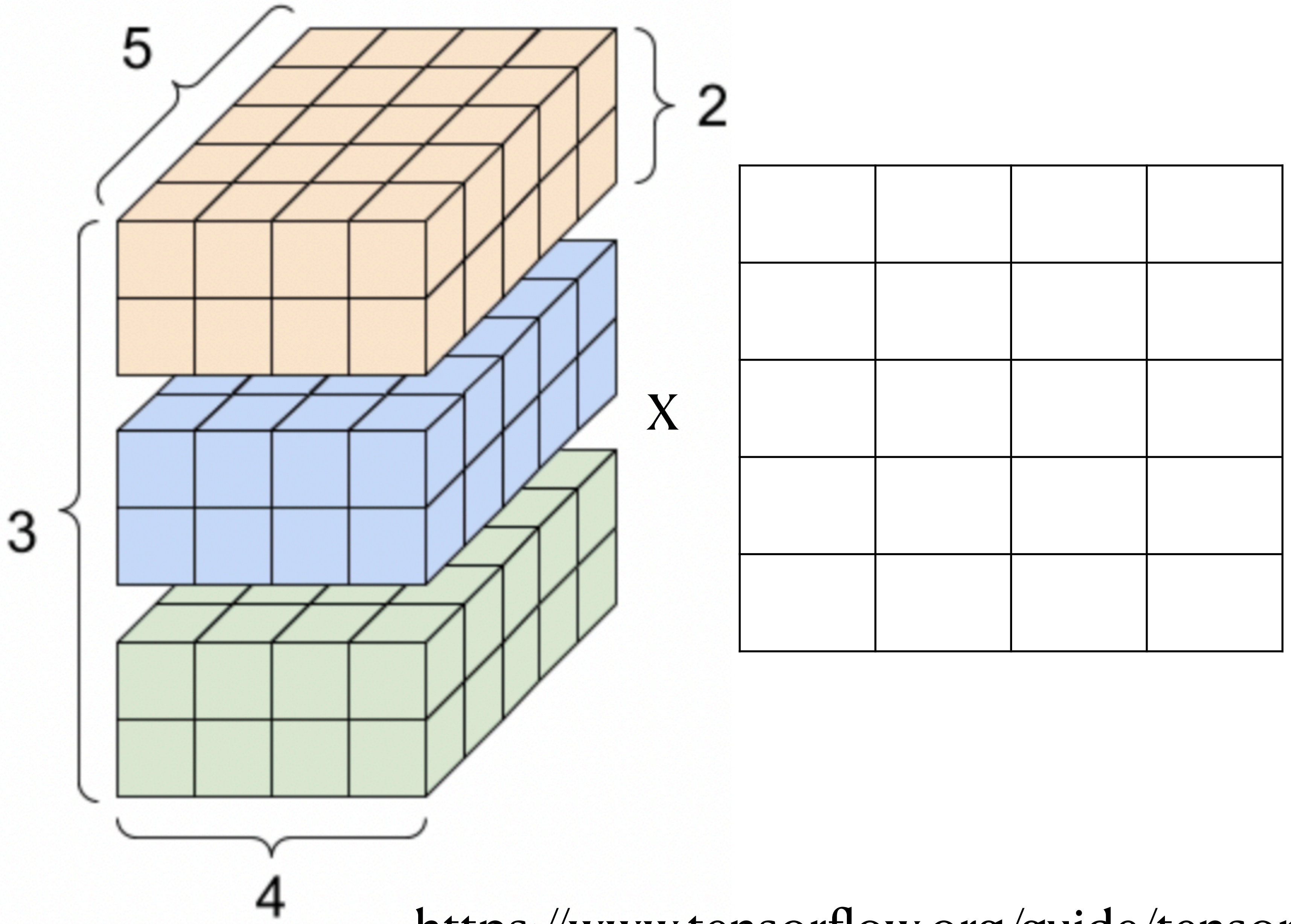


  - Can re-index the weight tensor

$$= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}$$

  - To compute $[\mathbf{H}]_{i,j}$ will sum over pixels in the image entered around location [i,j] and weighted by $[\mathbf{V}]_{i,j,a,b}$
  - Where indices a and b run over positive and negative offsets to cover the entire image

Tensor with shape: 3,2,4,5



https://www.tensorflow.org/guide/tensor

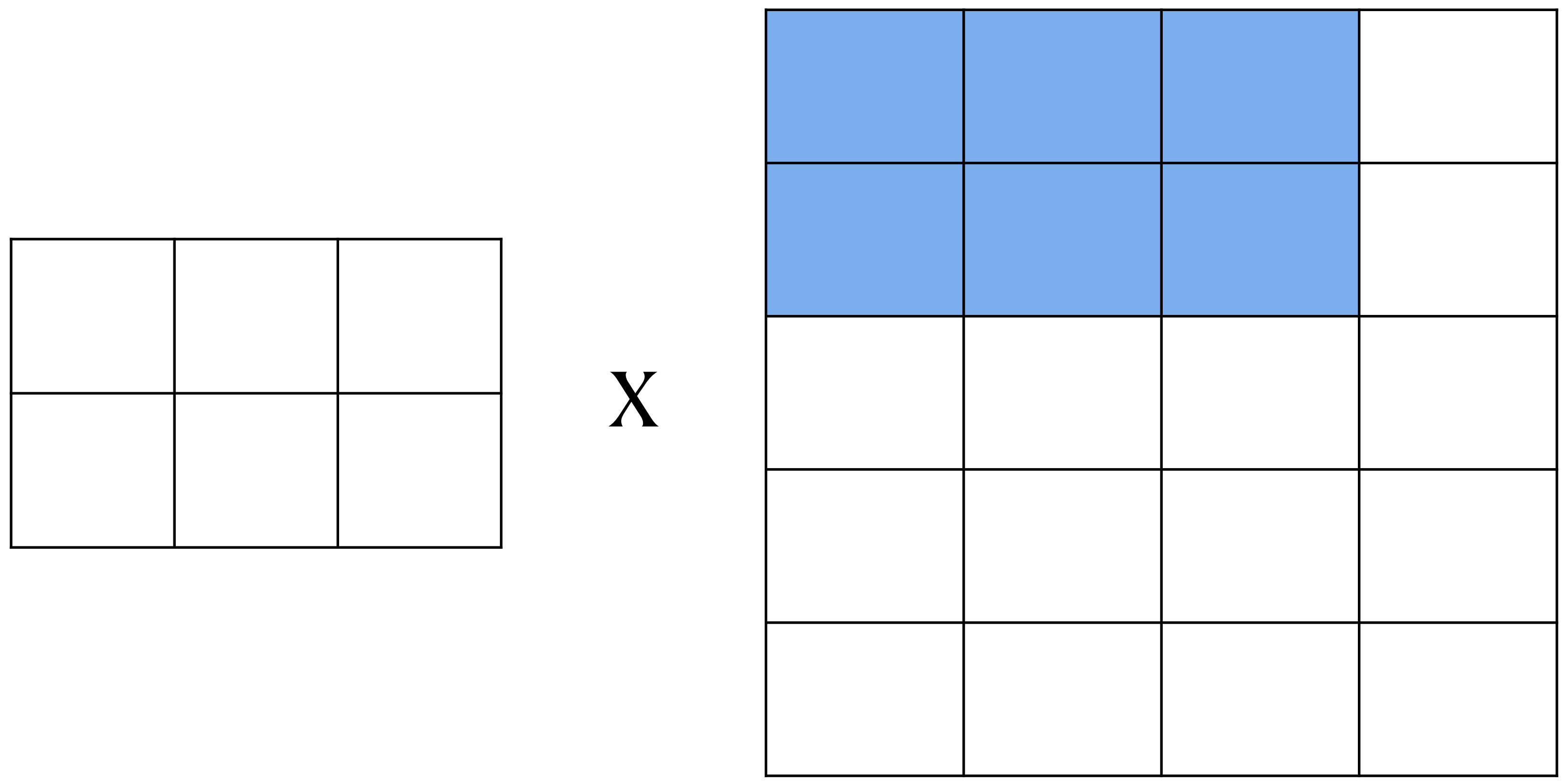
Constraining the Multi-Layer Perceptron

- How does this layer change with constraints?
  - Translational invariance implies that a shift in the input should lead to a shift in the hidden layer
  - bias and weights do not depend on i,j

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}$$

  - Locality: outside some range $[\mathbf{V}]_{a,b}=0$

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}$$



- Above is the definition of a convolutional layer
  - With this we have reduced the number of parameters needed from billions to a few hundred
- Including the possibility of different color channels (like stacked 2-D grids)

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [\mathsf{V}]_{a,b,c,d}[\mathsf{X}]_{i+a,j+b,c}$$

A learning now depends on inductive bias. If biases, don't agree with reality, the models will struggle. Example: correlated objects separated by large distances (described by long wavelength modes)

What is a Convolution?

- A convolution between two functions measures the overlap when one is flipped and shifted by x
  - Integral for continuous objects, sum for discrete objects

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x} - \mathbf{z})d\mathbf{z}.$$

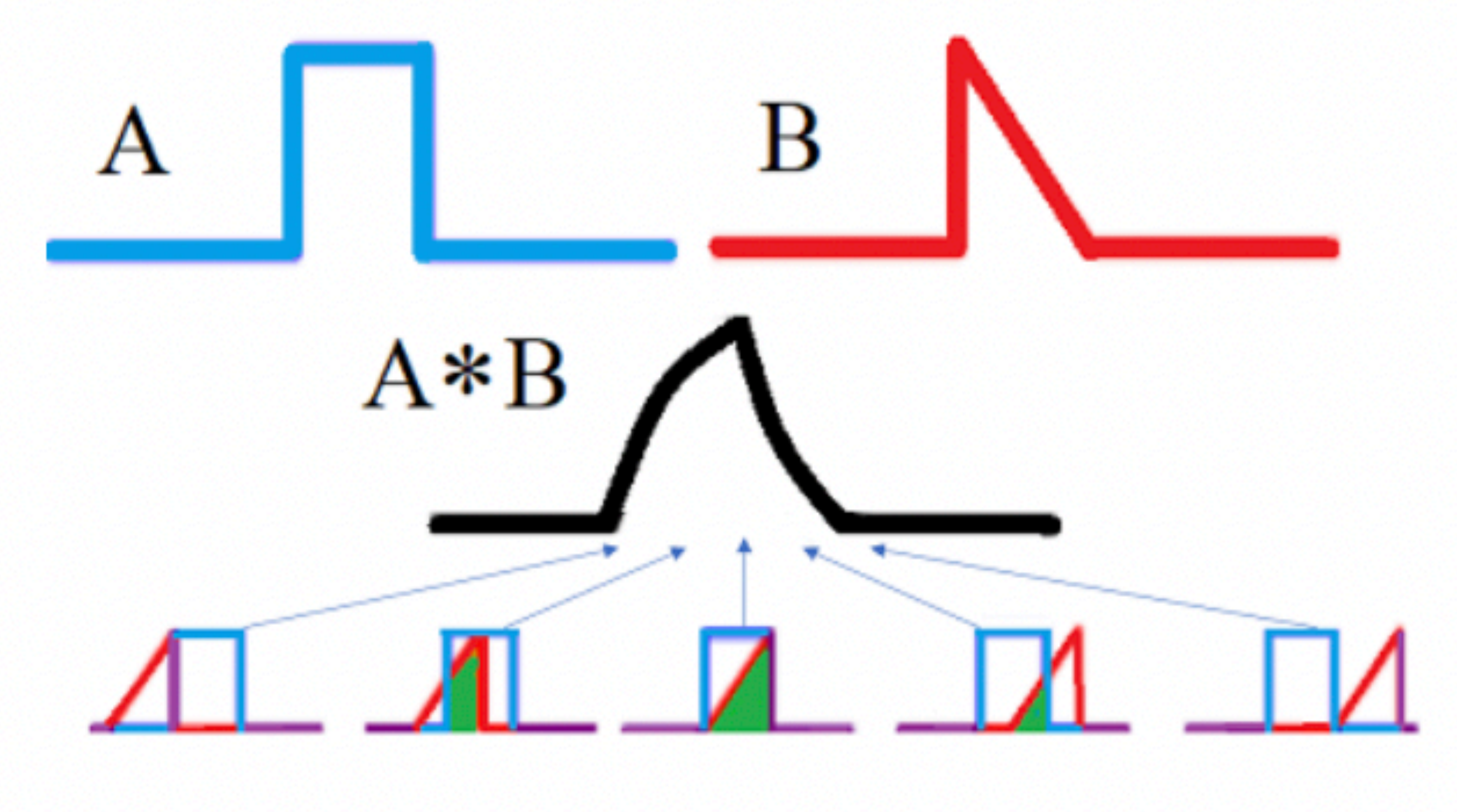
$$(f * g)(i) = \sum_a f(a)g(i - a).$$

  - For 2-D tensors it takes the form

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b)$$

  - Sum instead of difference is a cross correlation

We can consider our process a convolution because the resulting tensor will change shape



https://www.calculushowto.com/convolution-integral-simple-definition/

Cross Correlation vs. Convolution

- Can compare the outputs of the two operations by flipping one horizontally and vertically
  - should get the same answer

```
#creates a tensor where diagonal elements are 0
test=torch.ones((4,4))
for i in range(4):
    test[i,i]=0.

#Define a kernel
Ktest=torch.tensor([[1.0,1.0],[-1.0,-1.0]])

#Perfom convolution and get output
Ytest = corr2d(test, Ktest)
Ytest
```

```
tensor([[ 0.,  1.,  0.],
        [-1.,  0.,  1.],
        [ 0., -1.,  0.]])
```

```
#flip kernel horizontally and vertically to get convolution
KtestCon=torch.tensor([[-1.0,-1.0],[1.0,1.0]])

#new output
YtestCon = corr2d(test, KtestCon)
YtestCon
```

```
tensor([[ 0., -1.,  0.],
        [ 1.,  0., -1.],
        [ 0.,  1.,  0.]])
```

Flipped Horizontally

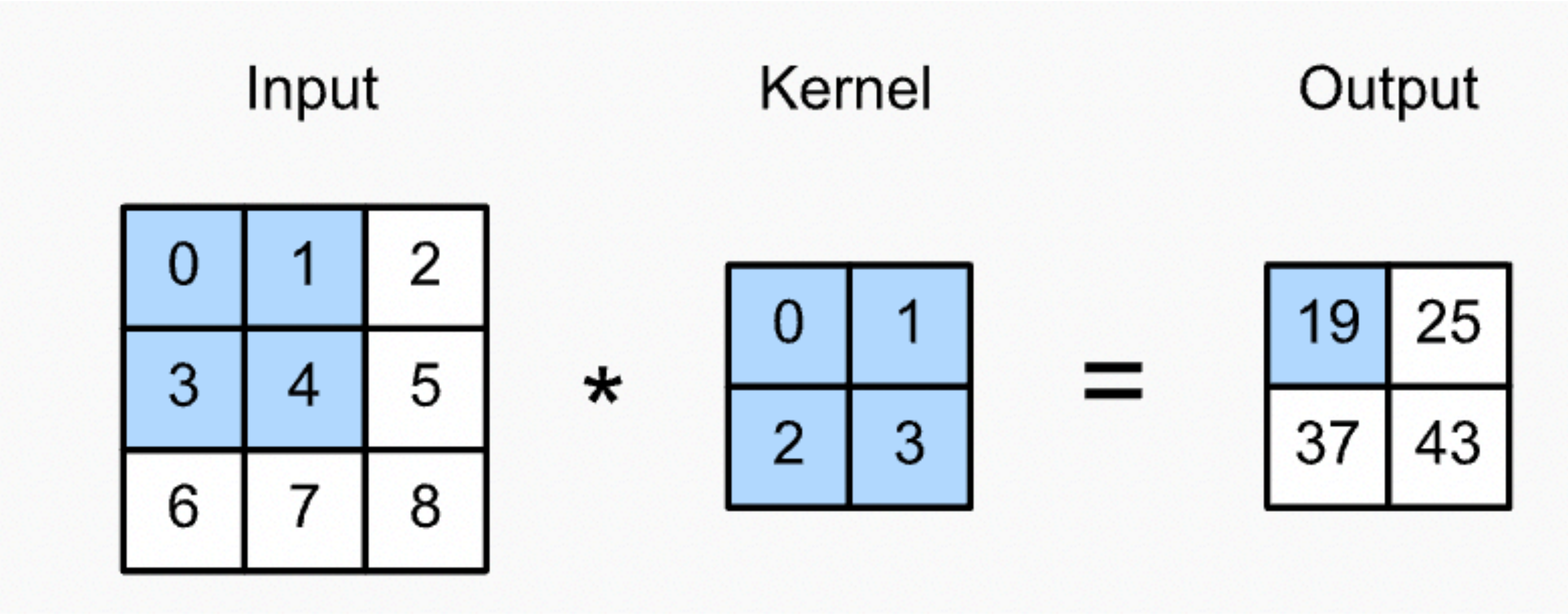
```
tensor([[ 0., -1.,  0.],
        [-1.,  0.,  1.],
        [ 0.,  1.,  0.]])
```

Flipped Vertically

```
tensor([[ 0.,  1.,  0.],
        [-1.,  0.,  1.],
        [ 0., -1.,  0.]])
```

Cross Correlation Operation

- Consider only 2-D data



$$
\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19,\\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25,\\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37,\\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43.
\end{aligned}
$$

  - Can only compute the cross correlation for locations where kernel fits wholly in the image
  - Output size depends on size of input and kernel

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

Cross Correlation Operation

- Code for above

```
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    #in example kernel is 2x2
    h, w = K.shape
    #gives shape of output using equation in book
    #will return 2x2
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    #does matrix multiplication
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

Convolutional Layers

- To be a full convolutional layer, we need to include the bias

```
#define a standard block
#calls Module from pytorch
class Conv2D(nn.Module):
    #calls nn.Module to perfrom necessary initializations
    #will also specify parameters
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    #defines forward propagation function, how block will return output
    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

- Typically initialize the kernels randomly
  - Can now implement a full layer

Learning a Kernel

- Can learn kernel from input and desired output
  - From edge detection example

```
[[1., 1., 0., 0., 0., 0., 1., 1.],
 [1., 1., 0., 0., 0., 0., 1., 1.],
 [1., 1., 0., 0., 0., 0., 1., 1.],
 [1., 1., 0., 0., 0., 0., 1., 1.],     *  [[ 1., -1.]]
 [1., 1., 0., 0., 0., 0., 1., 1.],
 [1., 1., 0., 0., 0., 0., 1., 1.]]
```

```
[[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
 [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
 [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
 [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
 [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
 [ 0.,  1.,  0.,  0.,  0., -1.,  0.]]
```

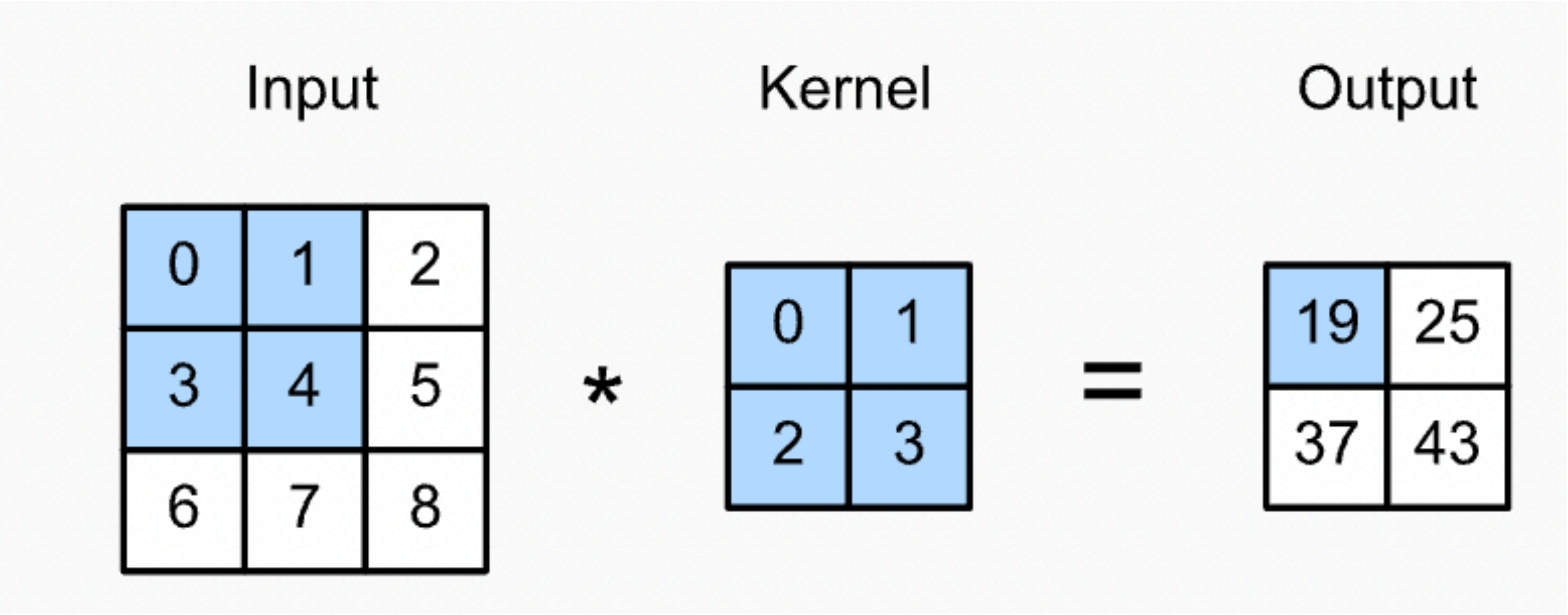
```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.Conv2d(1,1, kernel_size=(1, 2),bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both
X1 = X1.reshape((1, 1,6, 8))
Y1 = Y1.reshape((1, 1, 6,7))
lr = 3e-2  # Learning rate

for i in range(14):
  #some output
    Y_hat = conv2d(X1)
    #use mean square error to find loss function
    l = (Y_hat - Y1) ** 2
    #find gradient with computation graph
    conv2d.zero_grad()
    l.sum().backward()
    # Update the kernel
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    #will print loss for even epochs
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
conv2d.weight.data.reshape((1, 2))
```

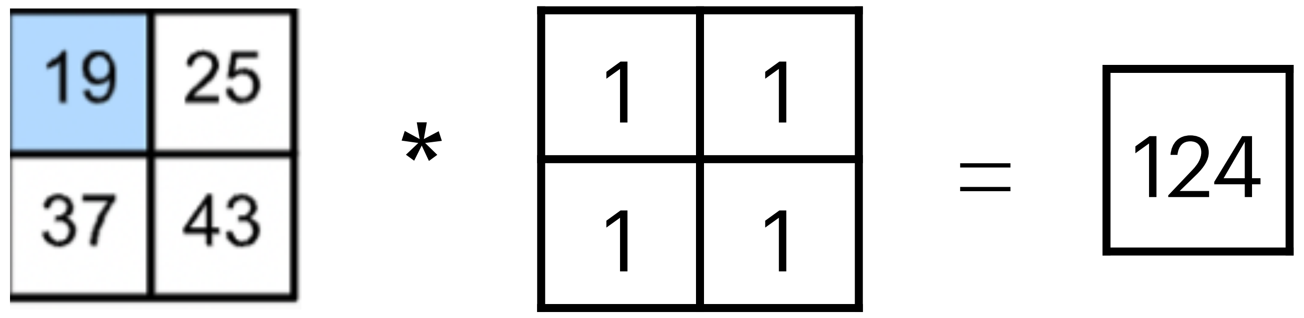
```
tensor([[-0.9982,  1.0006]])
```

Feature Map and Receptive Field

- Output of a convolutional layer is sometimes called a feature map
- For some element x, there is a receptive field that refers to all elements from previous layers that may affect the calculation of x during forward propagation
- When any element in a feature map needs a larger receptive field to detect input features over a broader area, we can build a deeper network
  - May be larger than size of input



Can build a
deeper network



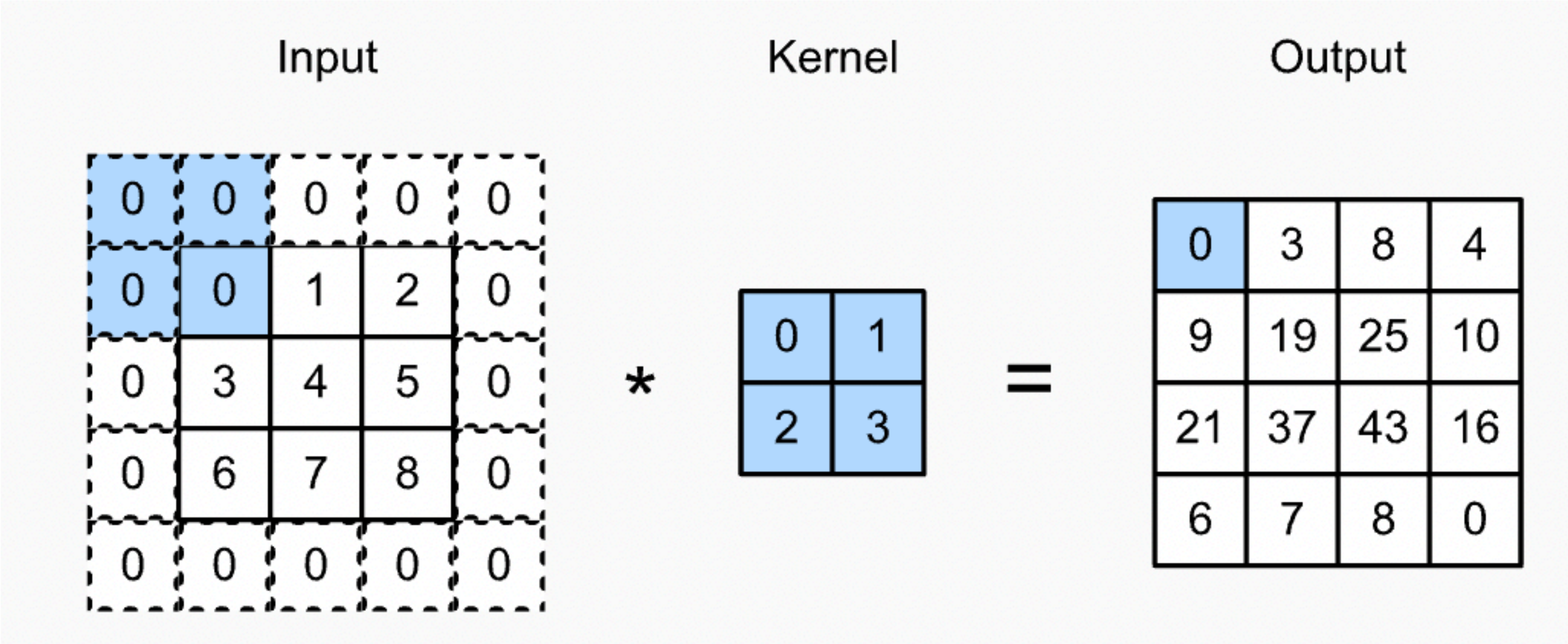
Padding and Stride

- Before output shape is determined by the shape of the input and convolution kernel
  - Many layers can really decrease dimensionality
- Can make use of two techniques:
  - Padding: keep more of the information at the border of the image
  - Stride: will drastically reduce the dimensionality

Padding

- Add extra pixels around the boundary of the input image
  - Typically set the values to zero
  - In general, the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$



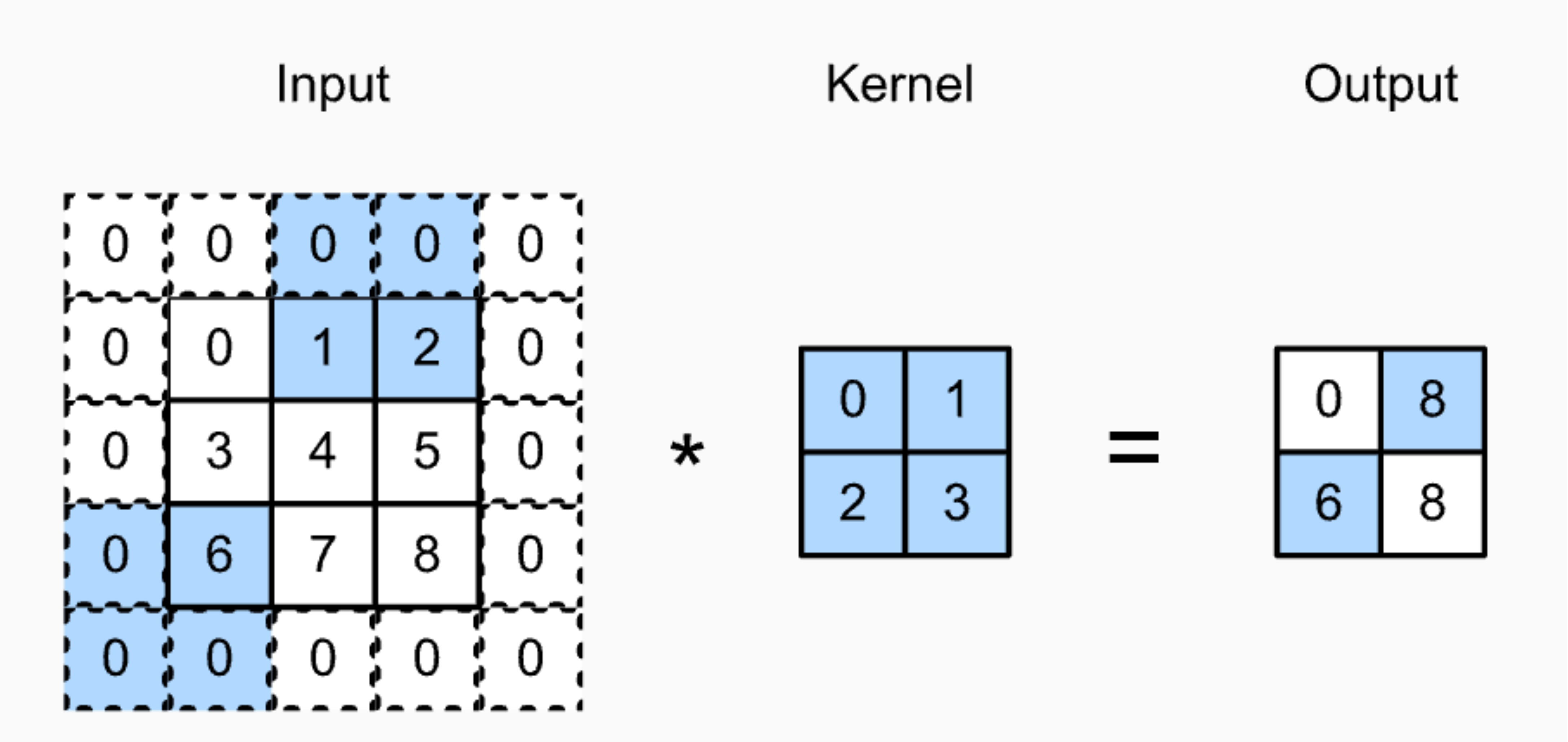
- Want to give input and output the same height and width
  - Set $p_h = k_h - 1$ and $p_w = k_w - 1$
  - Typically choose k to be odd to add same number of rows on each side
    - Will preserve spatial dimensionality
    - Information on the edges of image is weighted more evenly
- When height and width of kernel are different, we can use different amounts of padding (uncommon)

Stride

- Move kernel more than one element at a time
  - Number of rows and columns traversed per slide as stride
- In general, the output shape is

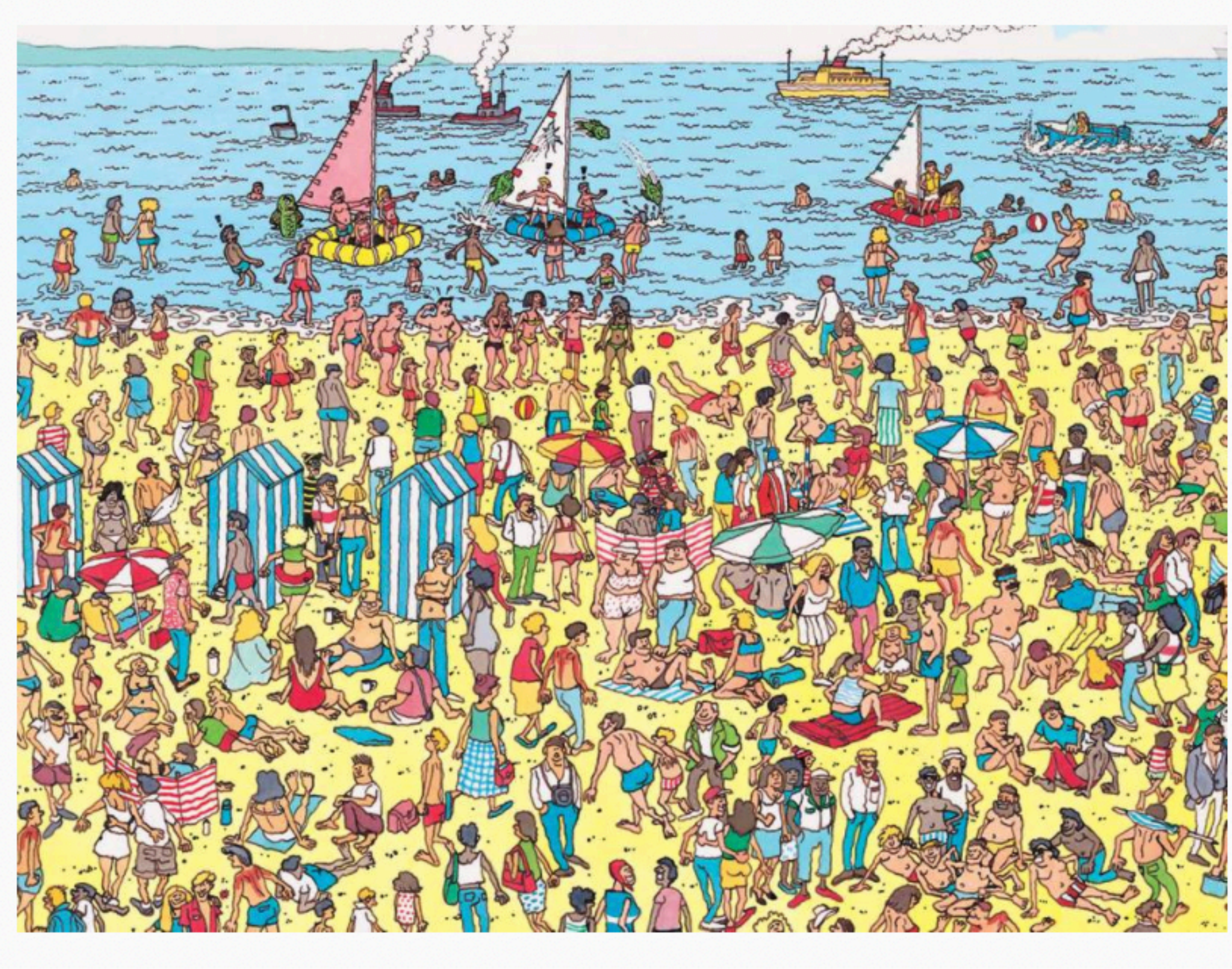
$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

- Rarely use inhomogeneous stride
- Using stride in later layers can remove important information from image



Application - Where's Waldo Example

- How will these things help Where's Waldo example?
  - Convolution operation - designed so that it will peak at areas of 'high waldoness'
  - Padding - will create a more even weighting of pixels at the edges. Will be easier to identify waldos that are at the edges the images
  - Stride - will quickly decrease number of pixels so waldo peak can be found faster



Summary

- Convolutional neural networks are a computationally efficient way to analyze image data
- Number of parameters needed is decreased with translational invariance and locality
- A kernel can be learned to reproduce a desired output
- When an element in a feature map needs a larger receptive field to detect broader features on the input, you can build a deeper network
- Padding can be used to preserve edge effects
- Stride can be used to increase efficiency or down sample