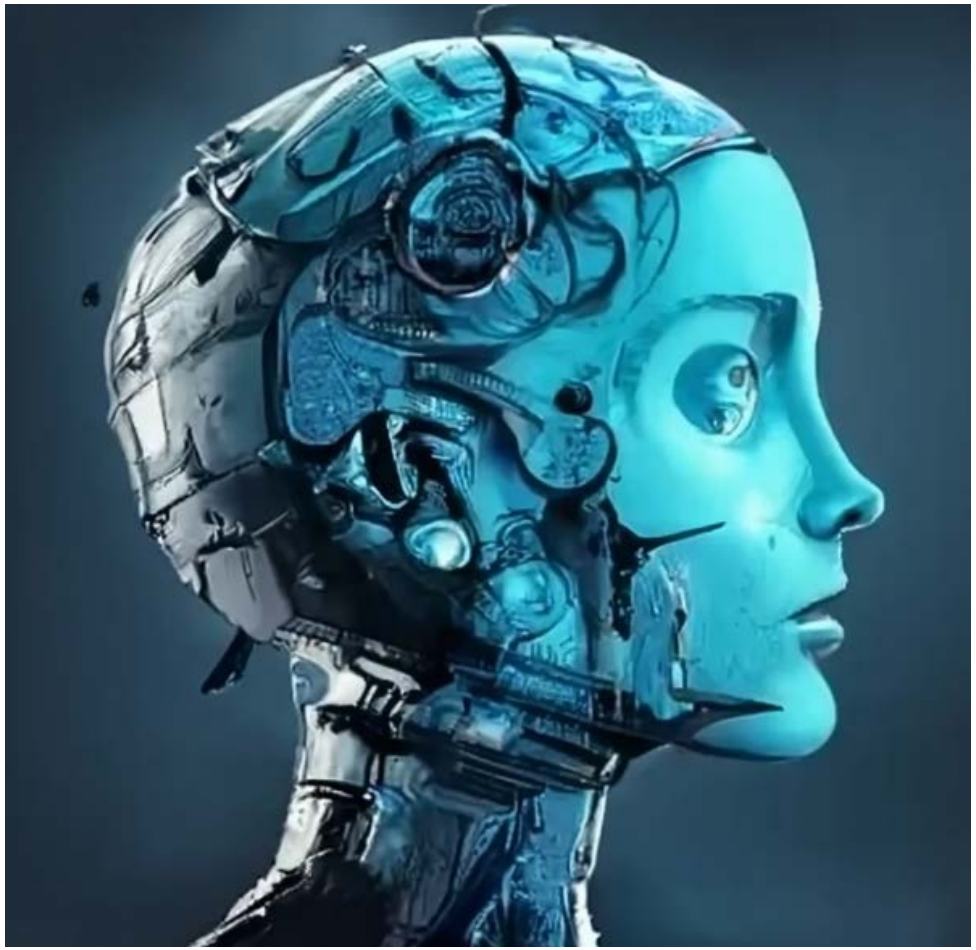
Machine Intelligence

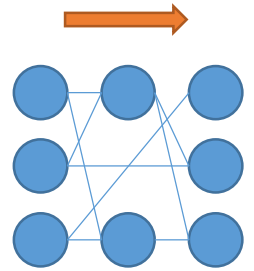
Artificial Neural Network Architectures

Dr. Cory Merkel

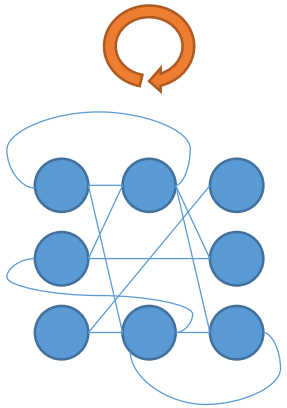


ANN Topology

- The *topology* of an ANN encodes prior knowledge into the pattern of connectivity between neurons
- There are two main types of topology:
  - Feedforward
    - ❑ Information flows in one direction from input to output
      - Examples: Perceptron, MLP, Convolutional Neural Network, Transformer



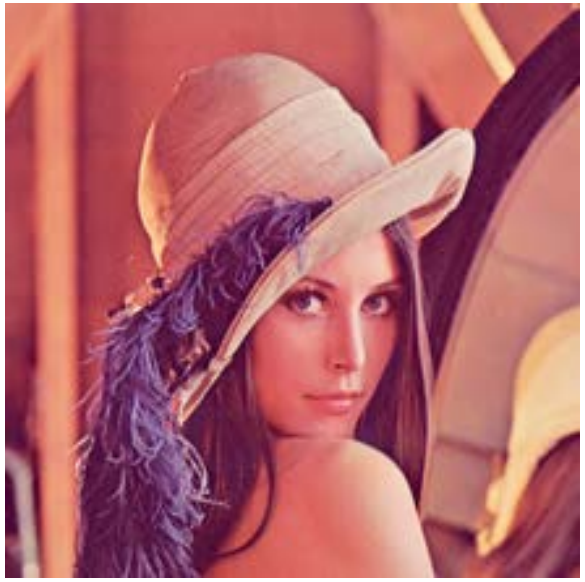
  - Recurrent
    - ❑ Information can be fed back to create short-term memory
      - Examples: Hopfield Networks, Echo State Networks, Long Short Term Memory

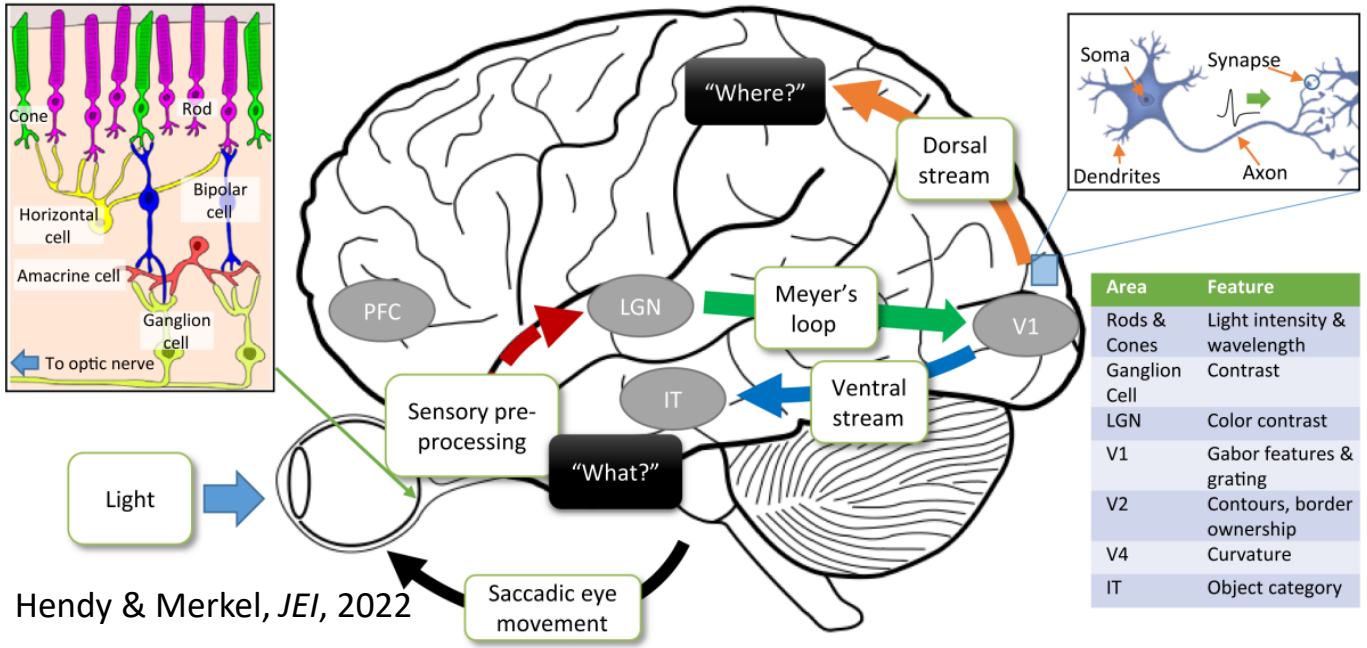


- Finding a good topology is hard
  - We take inspiration from biology (e.g. CNNs)
  - We can *learn* the topology (e.g. genetic algorithms)
  - Best topology depends on problem we are trying to solve

A Tiny Bit about Human Vision

- Consider the problem of classifying images:
  - How do *we* recognize this image?
    - ❑As visual information moves up the hierarchy of our brains, we detect more and more complex features: Contrast → Edges → Lines & Corners → Contours and Curvature …
      - Translational equivariance: An edge is an edge, regardless of where it is in the image



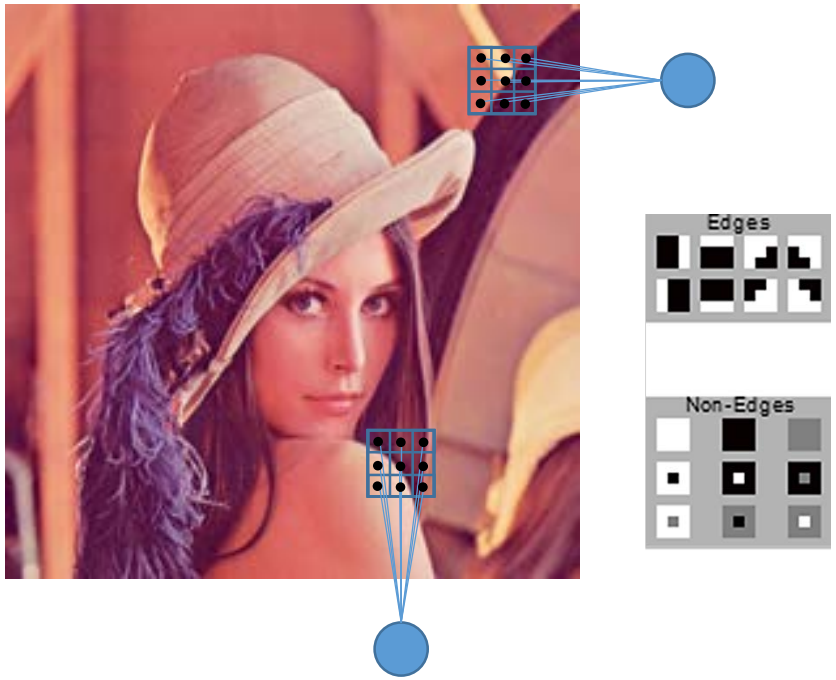


Area	Feature
Rods & Cones	Light intensity & wavelength
Ganglion Cell	Contrast
LGN	Color contrast
V1	Gabor features & grating
V2	Contours, border ownership
V4	Curvature
IT	Object category

Hendy & Merkel, *JEI*, 2022

Getting Translational Equivariance in Neural Networks

- Imagine that we want to detect an edge at a location in an image:
  - We could have a small single-neuron network that we train to tell us whether the center of the region is part of an edge or not



Note that this neuron is only connected to 9 pixels out of all of the pixels in the image. You can think of the other weight values as 0.

Since we define an edge the same, regardless of what part of the image we're looking at, we should learn identical networks!

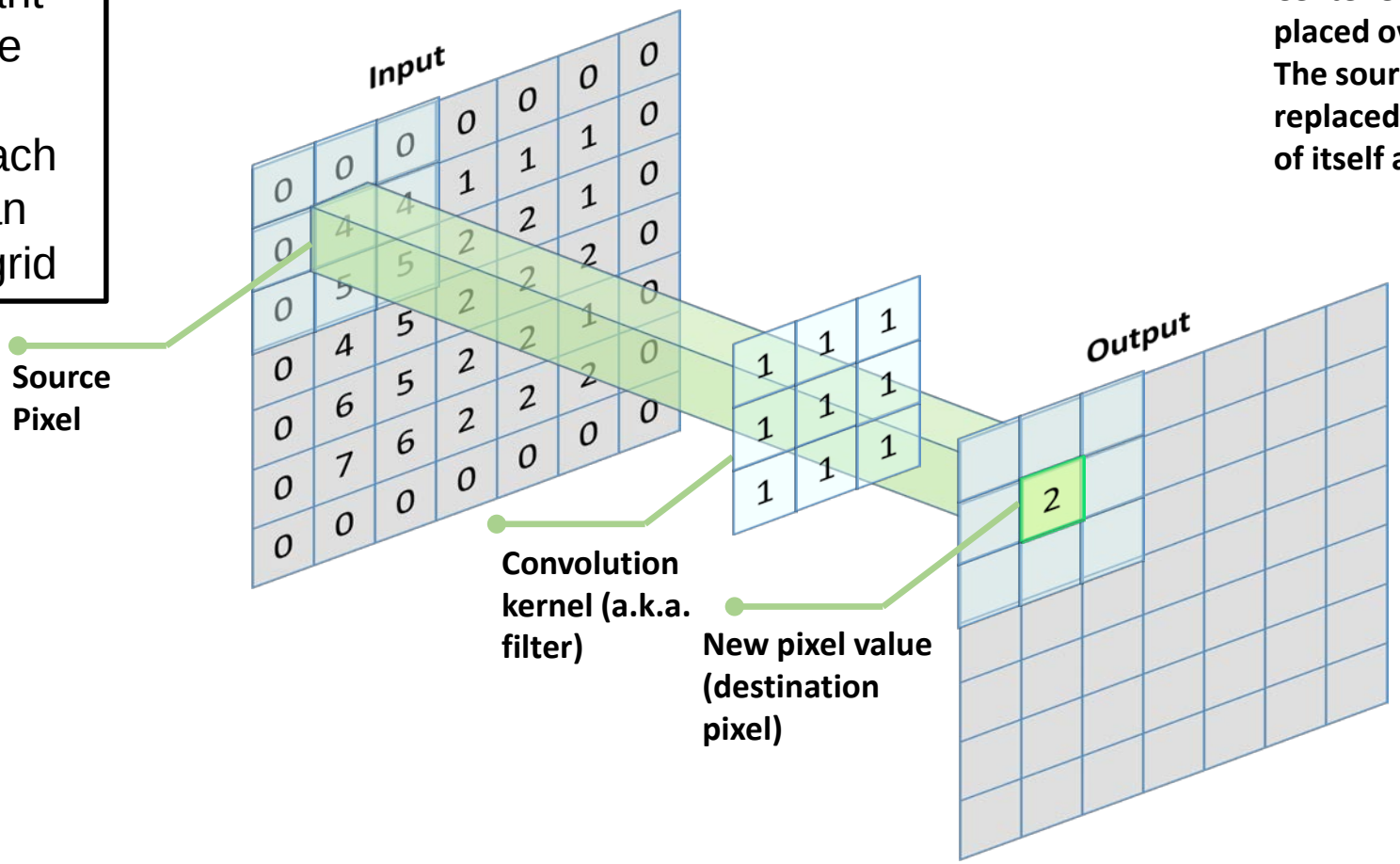
- We can *force* the weights of each network to be the same or, we can build just 1 network and use it at every location of the image!
- Many fewer weights than a fully-connected network!

Getting Translational Equivariance in Neural Networks

- We call our set of weights a *filter* or *kernel*, and we *convolve* the filter over the entire image to see which pixels are part of an edge

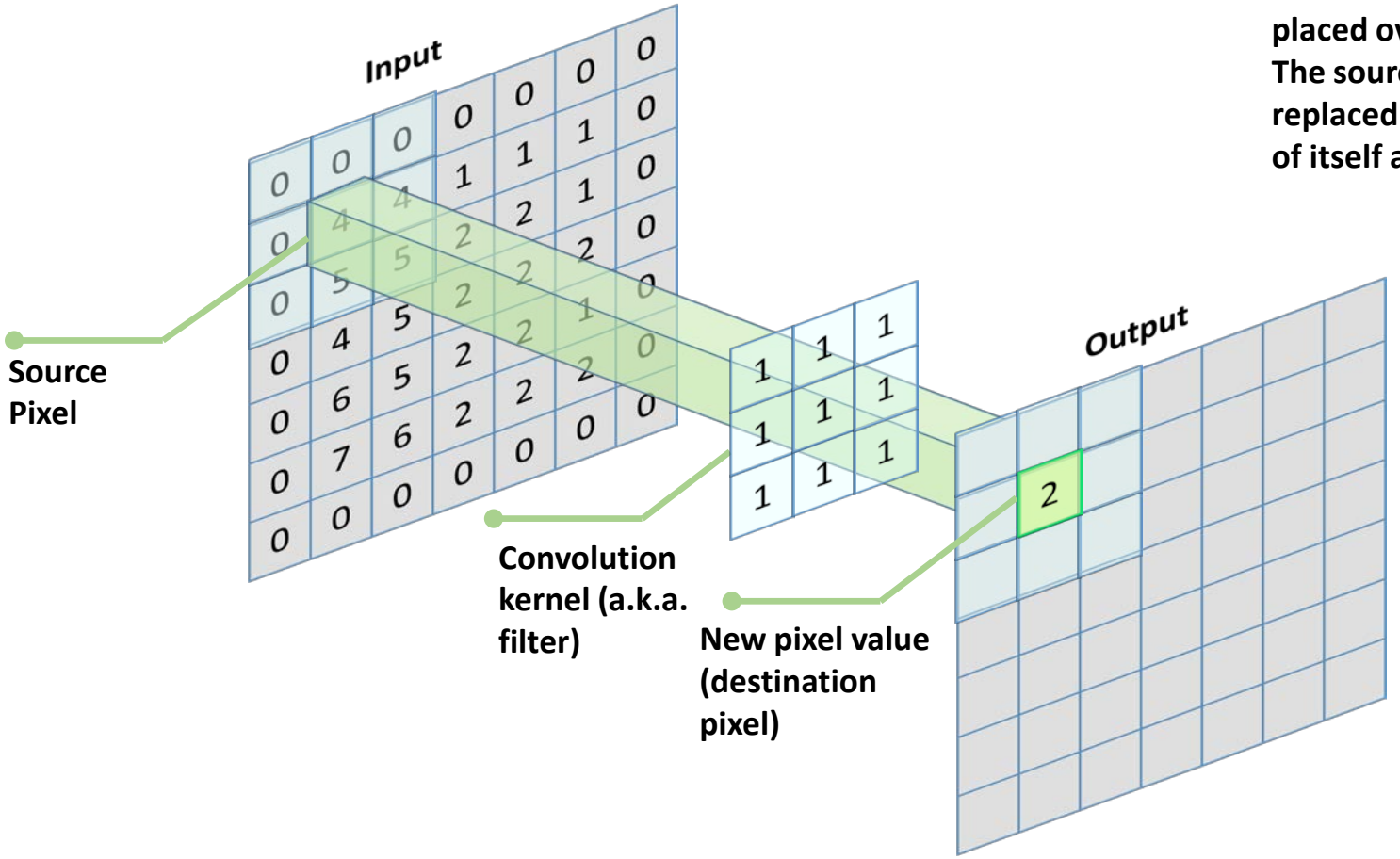
Convolution

Suppose we want to take an image and get a new image where each pixel is the mean value of a 3x3 grid



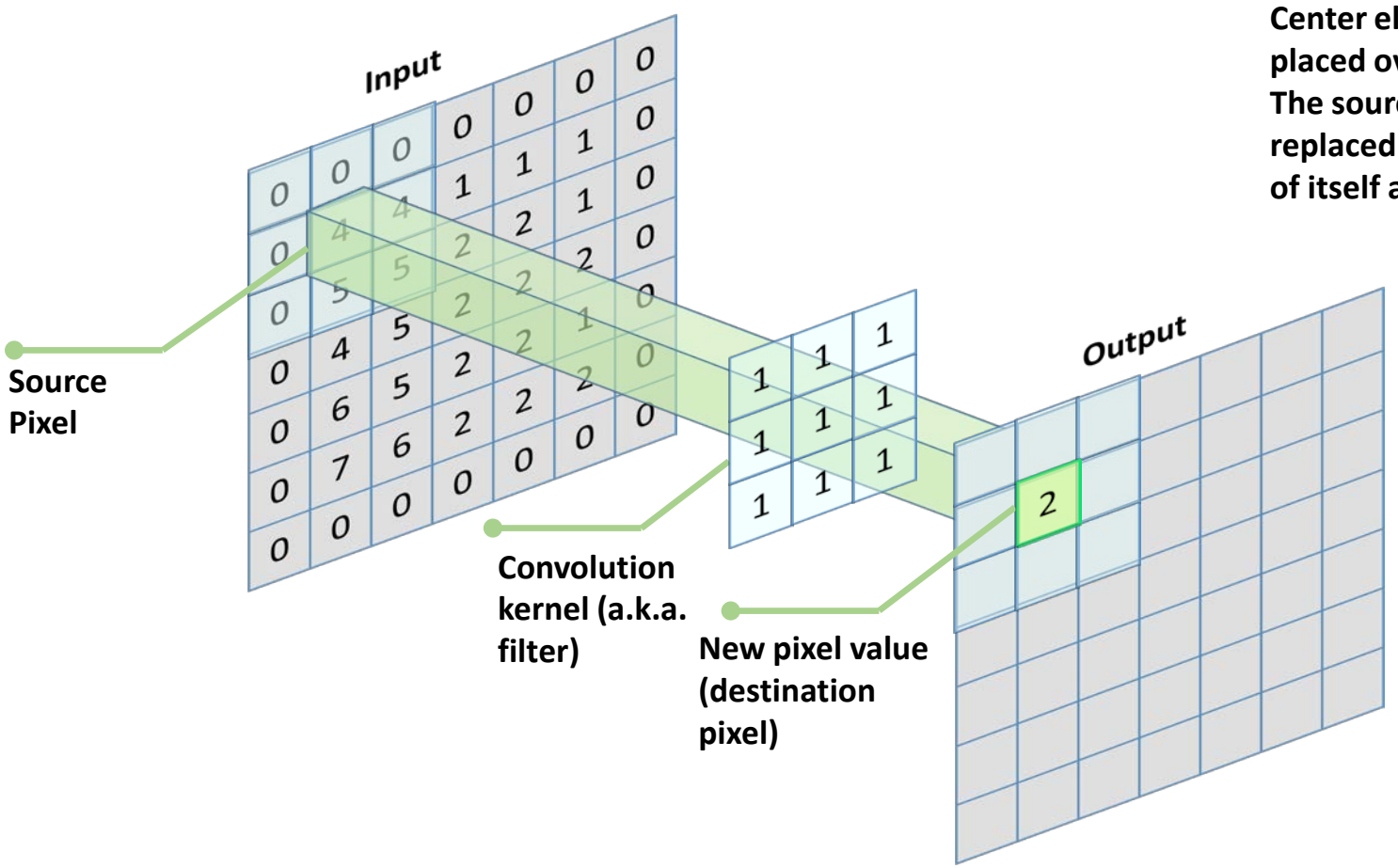
Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

Convolution



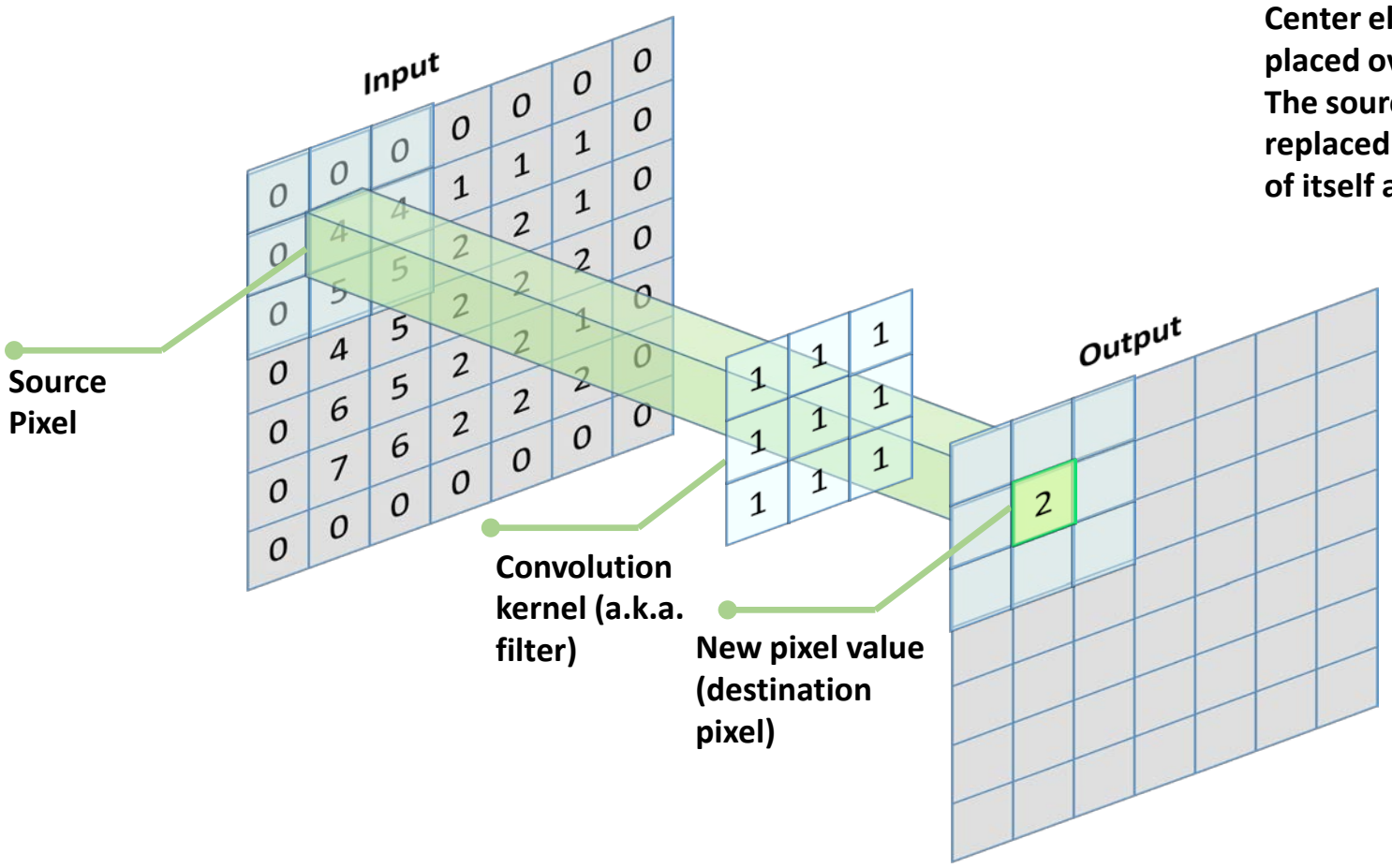
$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+1*0+1*4+1*4+1*0+1*5+1*5)=2$$

Convolution



$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+\underbrace{1*0+1*4+1*4}_{\text{Middle}}+1*0+1*5+1*5)=2$$

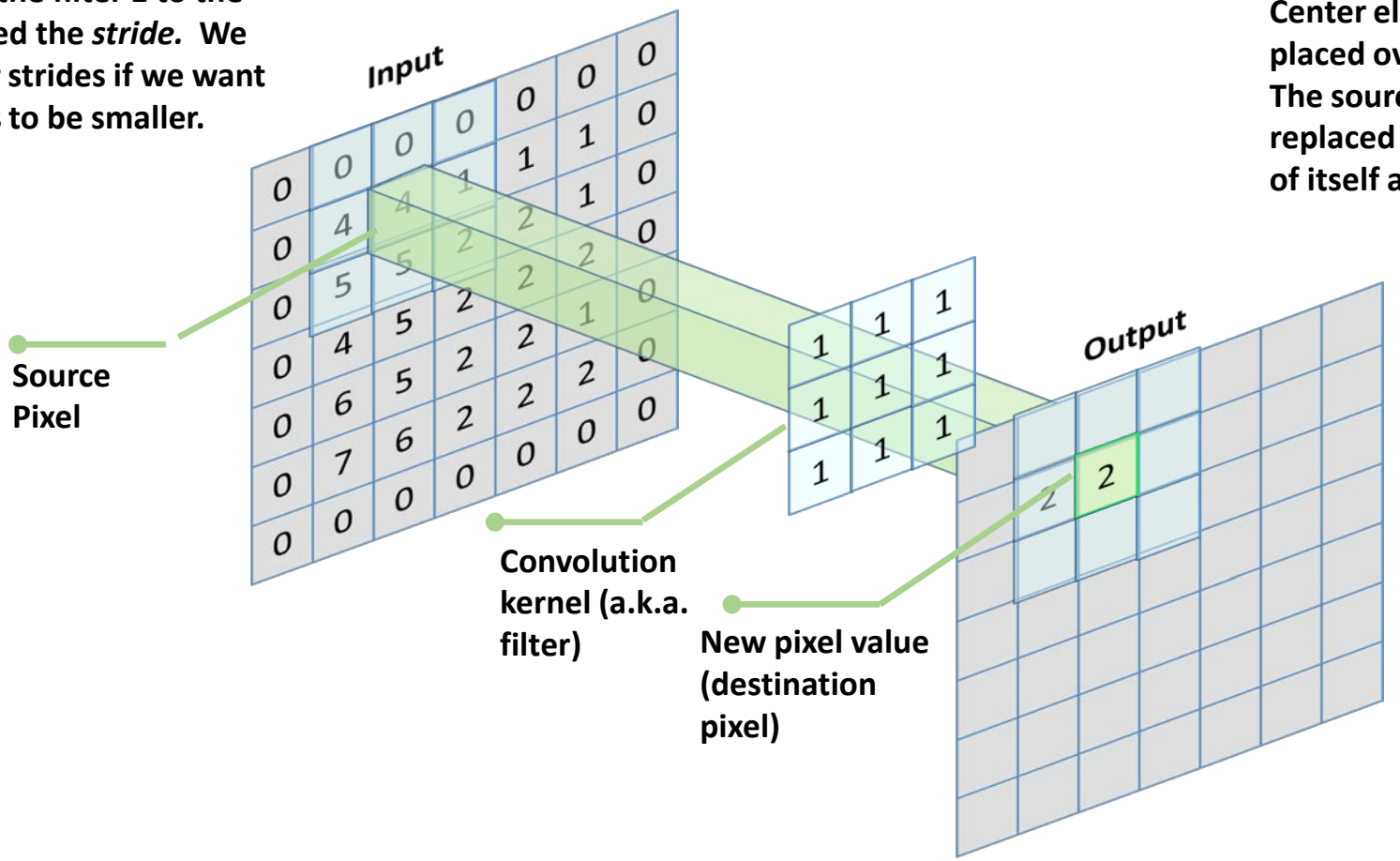
Convolution



$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+\underbrace{1*0+1*4+1*4}_{\text{Middle}}+\underbrace{1*0+1*5+1*5}_{\text{Bottom}})=2$$

Convolution

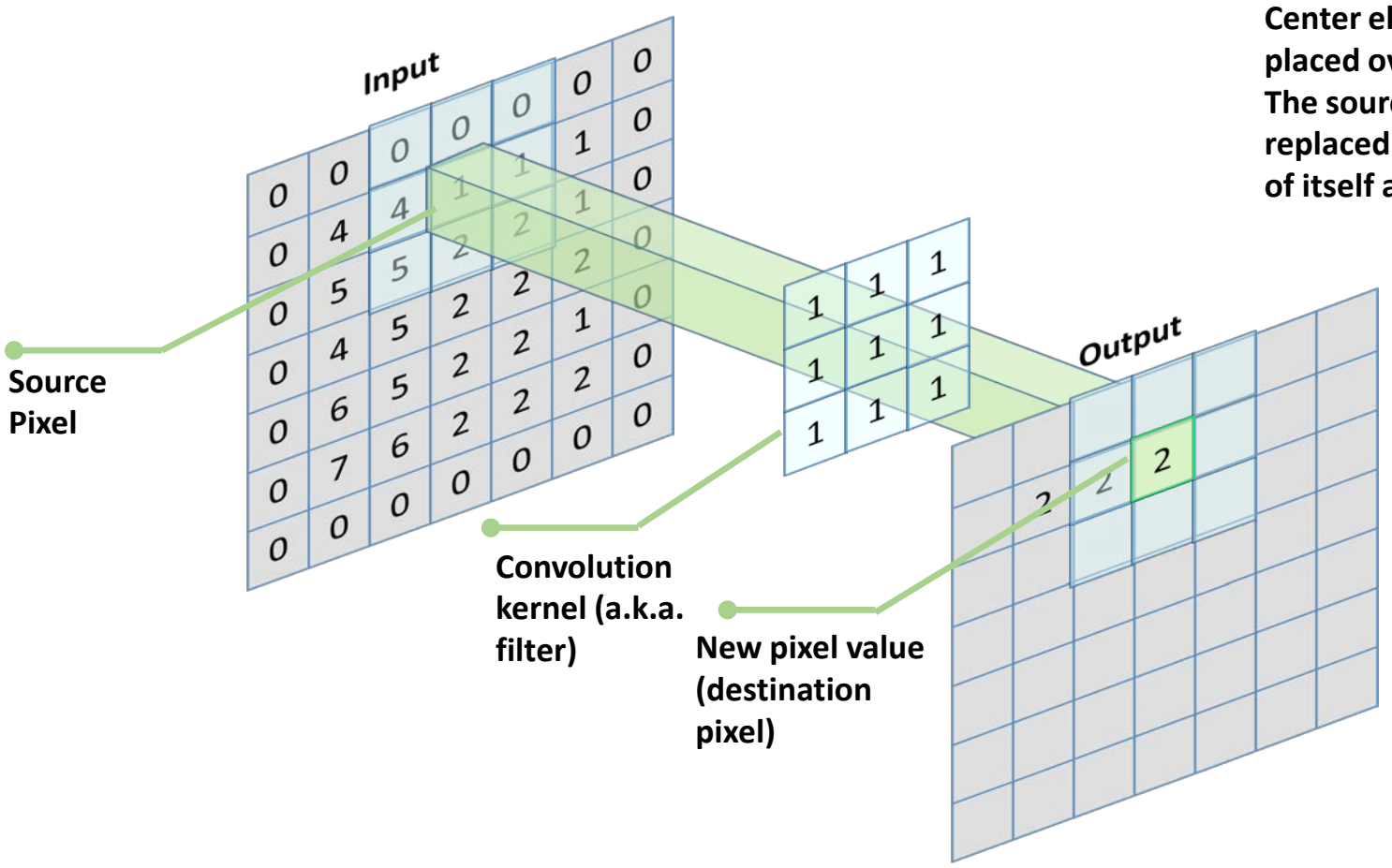
Here, we moved the filter 1 to the right. This is called the *stride.* We could have larger strides if we want our feature maps to be smaller.



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+\underbrace{1*4+1*4+1*1}_{\text{Middle}}+\underbrace{1*5+1*5+1*2}_{\text{Bottom}})=2.3\approx 2$$

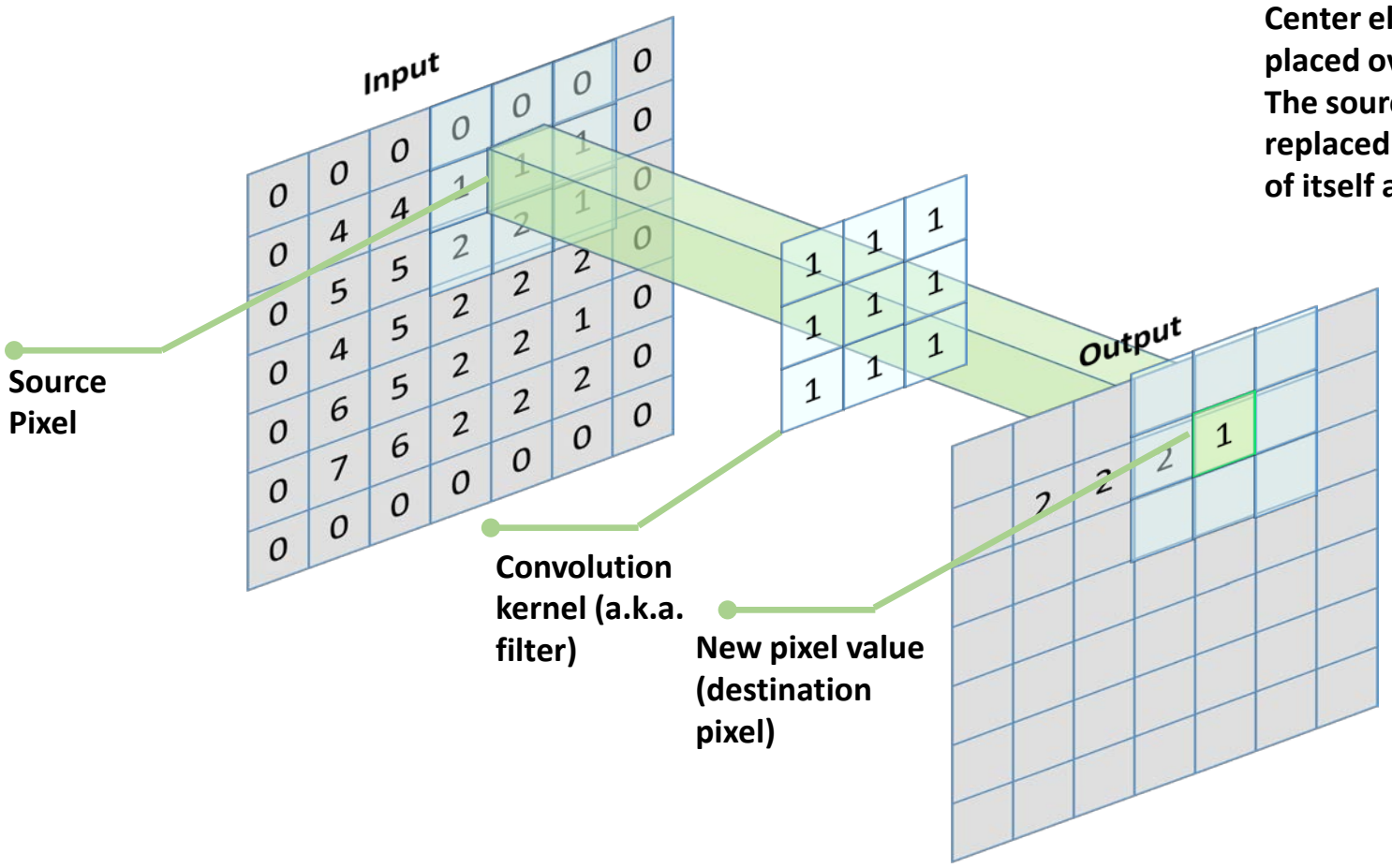
Convolution



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

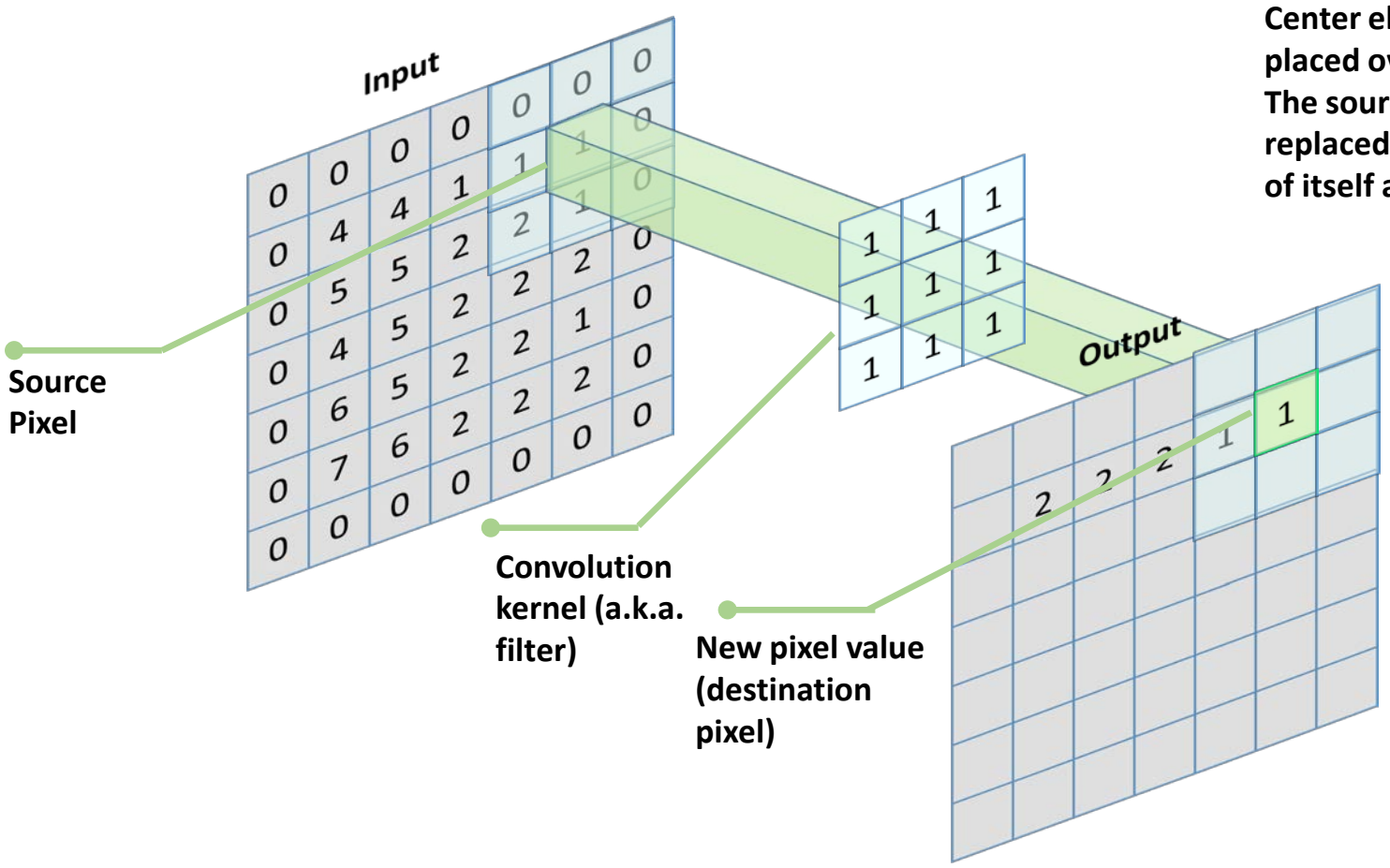
$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+\underbrace{1*4+1*1+1*1}_{\text{Middle}}+\underbrace{1*5+1*2+1*2}_{\text{Bottom}})=1.7\approx 2$$

Convolution



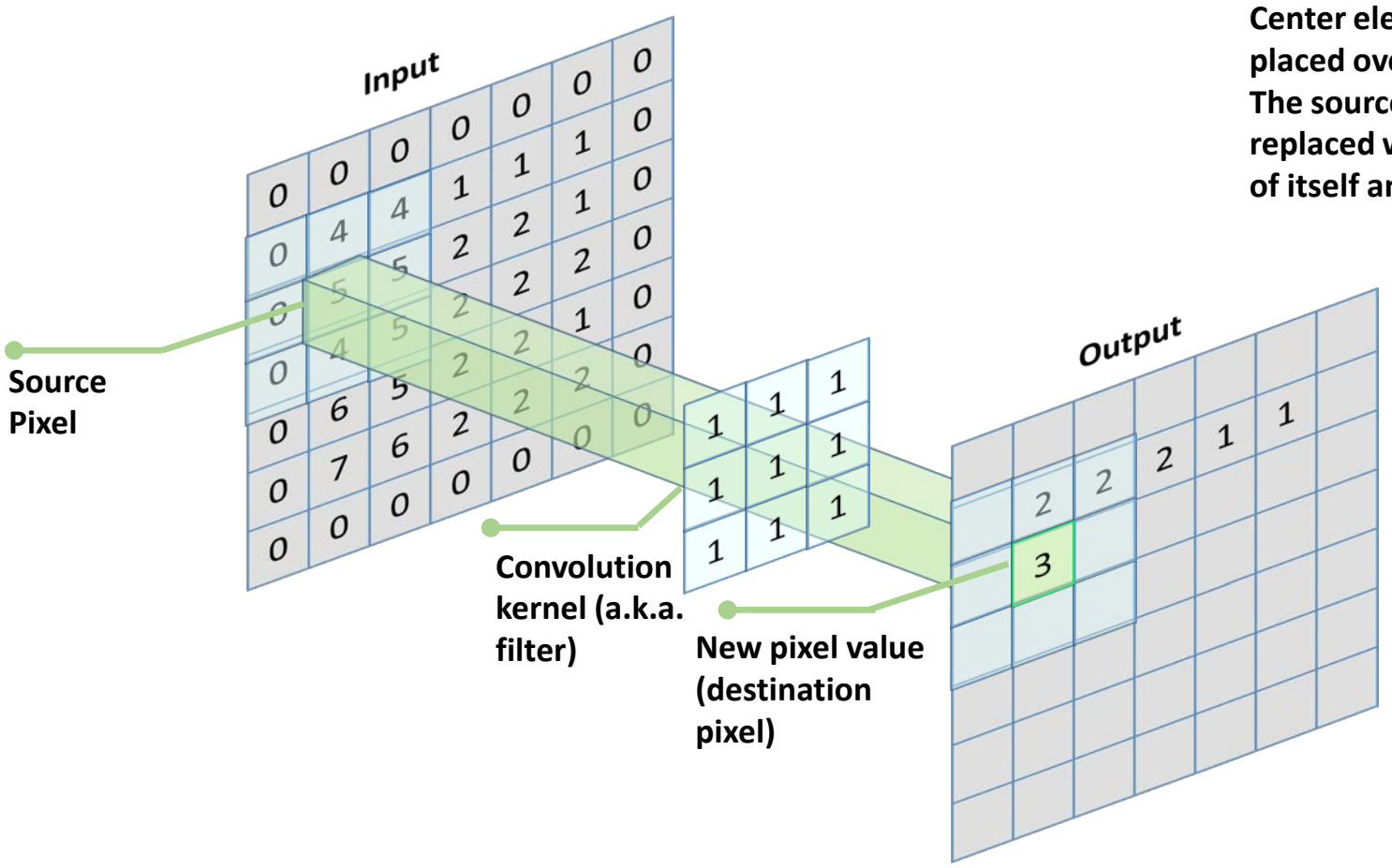
$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+\underbrace{1*1+1*1+1*1}_{\text{Middle}}+\underbrace{1*2+1*2+1*1}_{\text{Bottom}})=0.9\approx 1$$

Convolution



$$\frac{1}{9}(\underbrace{1*0+1*0+1*0}_{\text{Top}}+\underbrace{1*1+1*1+1*0}_{\text{Middle}}+\underbrace{1*2+1*1+1*0}_{\text{Bottom}})=0.6\approx 1$$

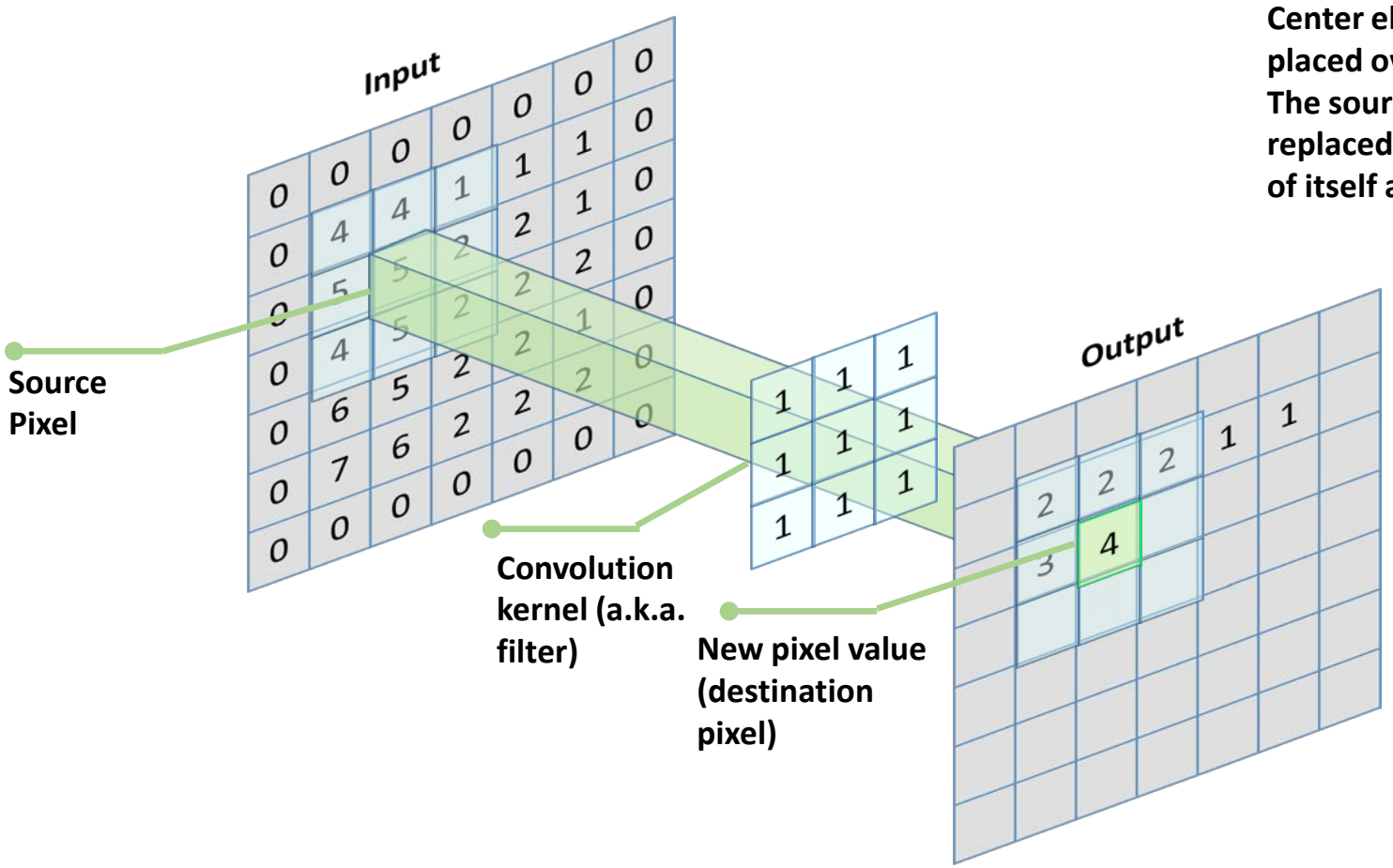
Convolution



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

$$\frac{1}{9}(\underbrace{1*0+1*4+1*4}_{\text{Top}}+\underbrace{1*0+1*5+1*5}_{\text{Middle}}+\underbrace{1*0+1*4+1*5}_{\text{Bottom}})=3$$

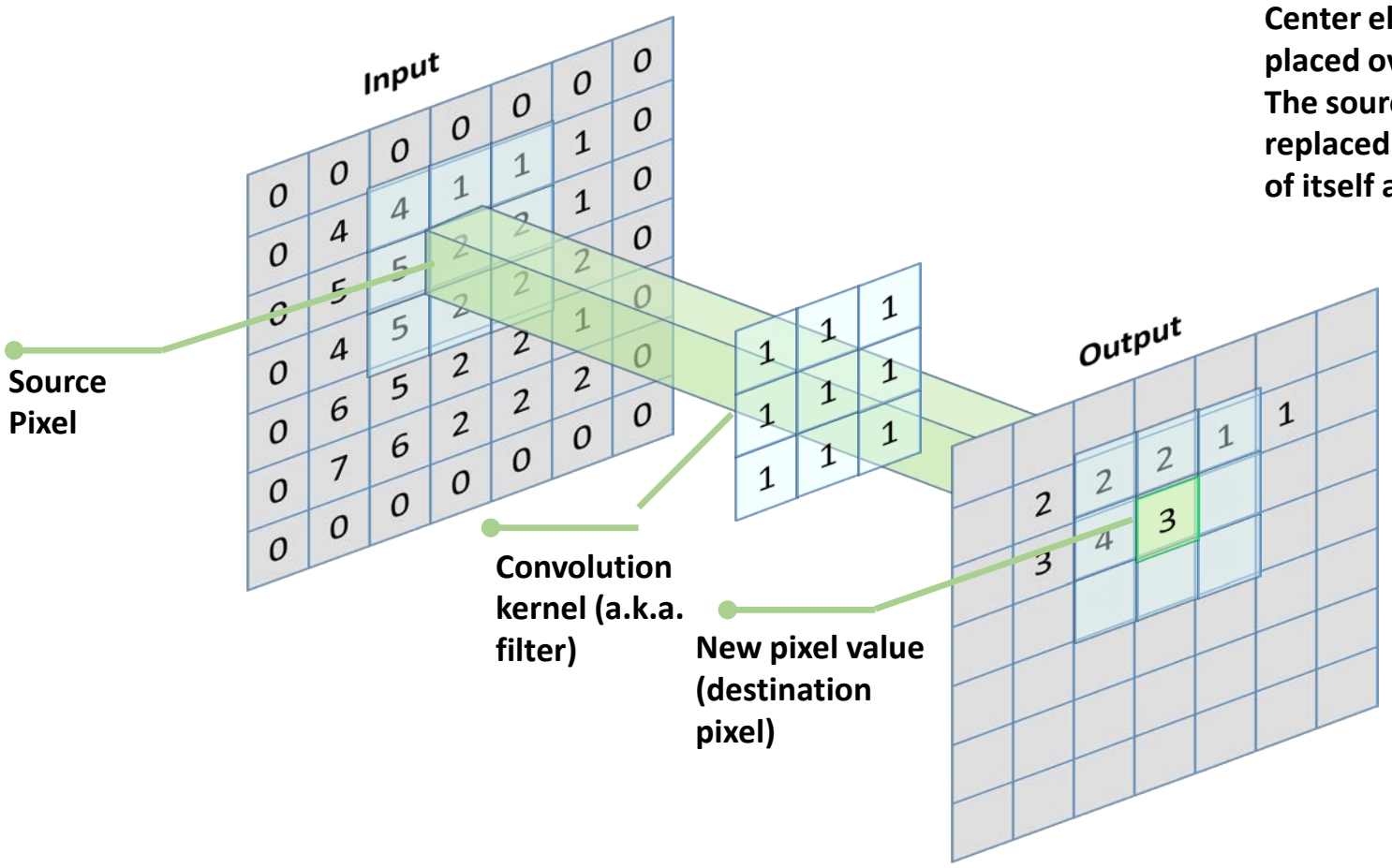
Convolution



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

$$\frac{1}{9}(\underbrace{1*4+1*4+1*1}_{\text{Top}}+\underbrace{1*5+1*5+1*2}_{\text{Middle}}+\underbrace{1*4+1*5+1*2}_{\text{Bottom}})=3.6\approx 4$$

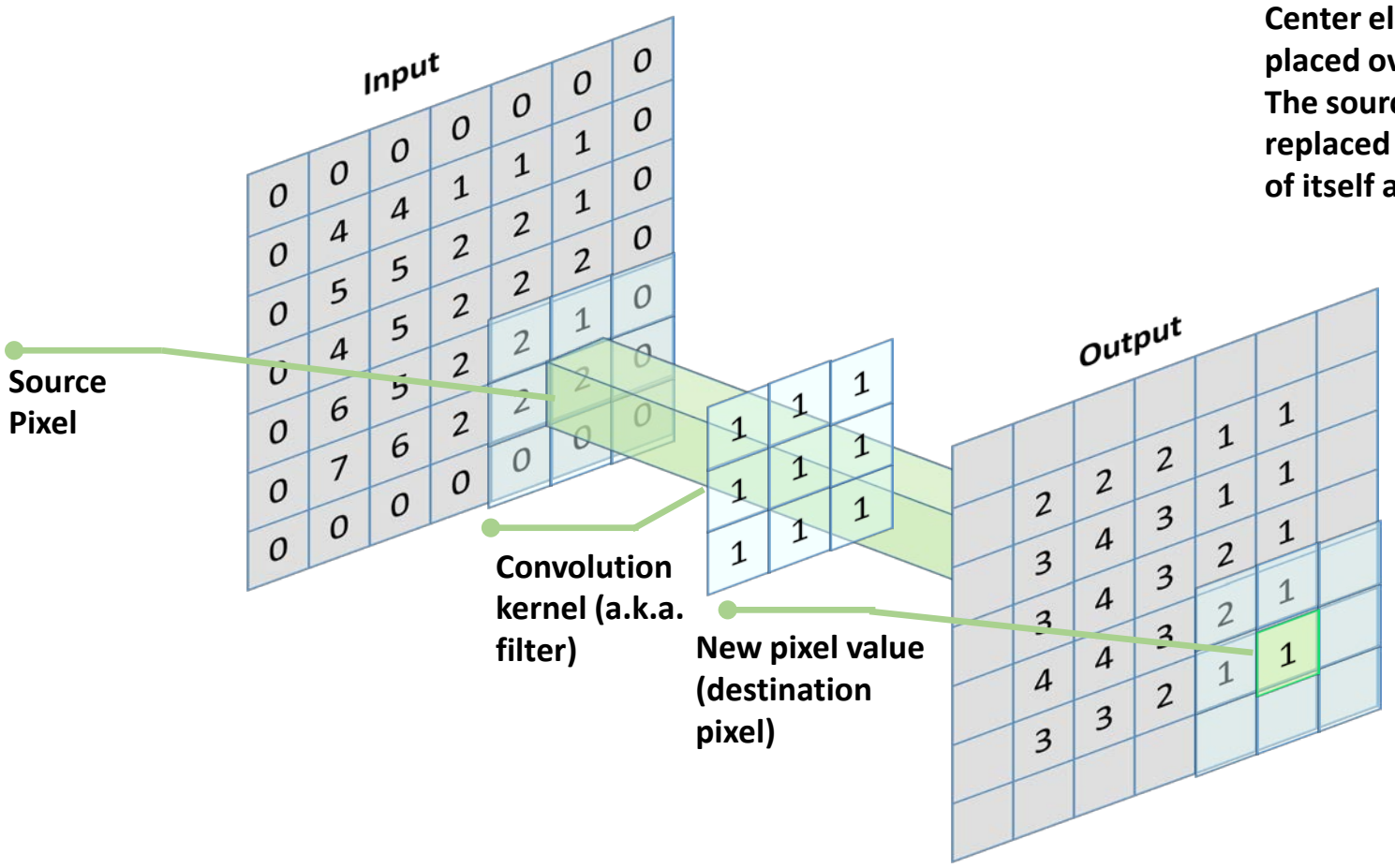
Convolution



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

$$\frac{1}{9}(\underbrace{1*4+1*1+1*1}_{\text{Top}}+\underbrace{1*5+1*2+1*2}_{\text{Middle}}+\underbrace{1*5+1*2+1*2}_{\text{Bottom}})=2.7\approx 3$$

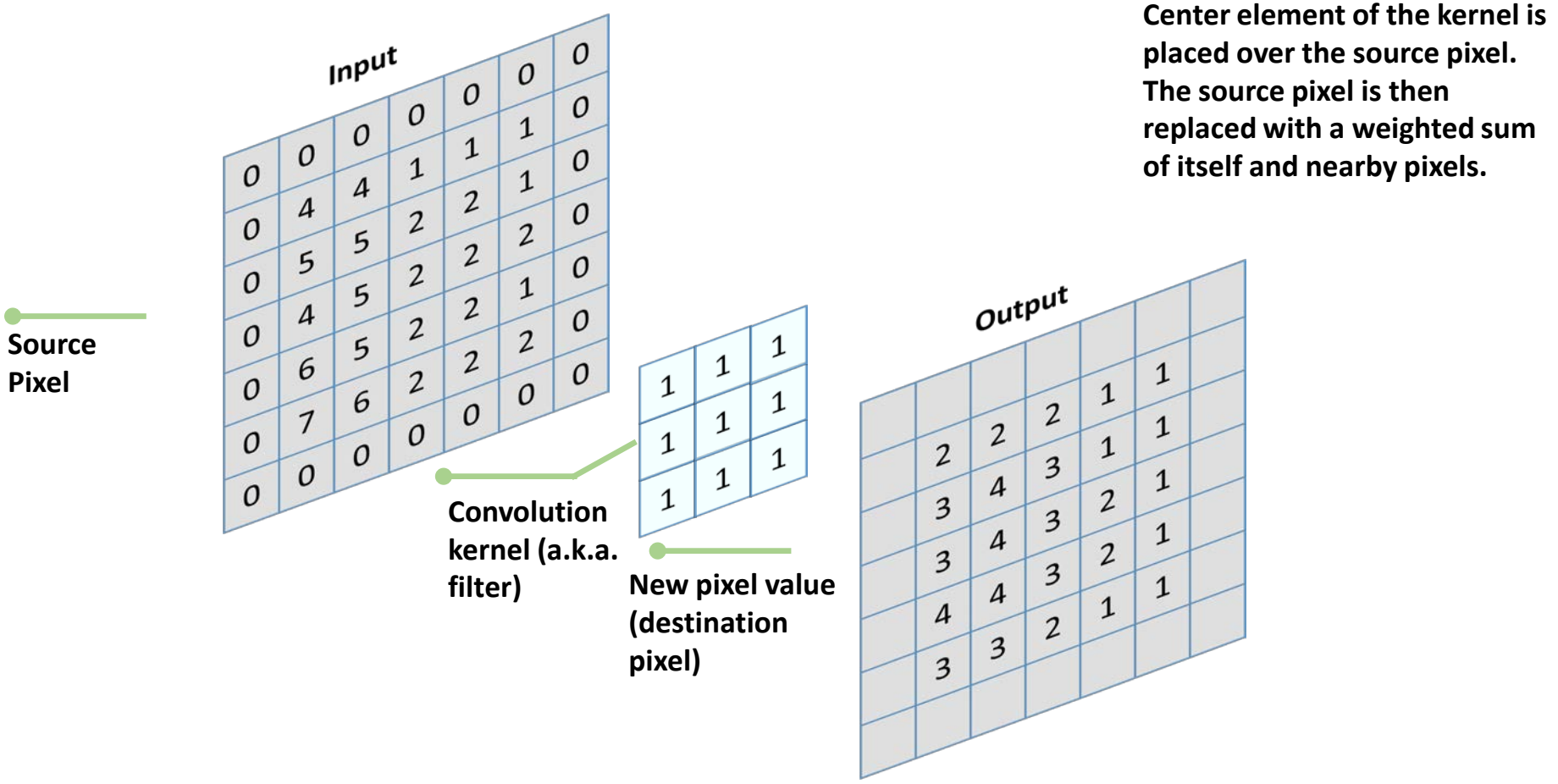
Convolution



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.

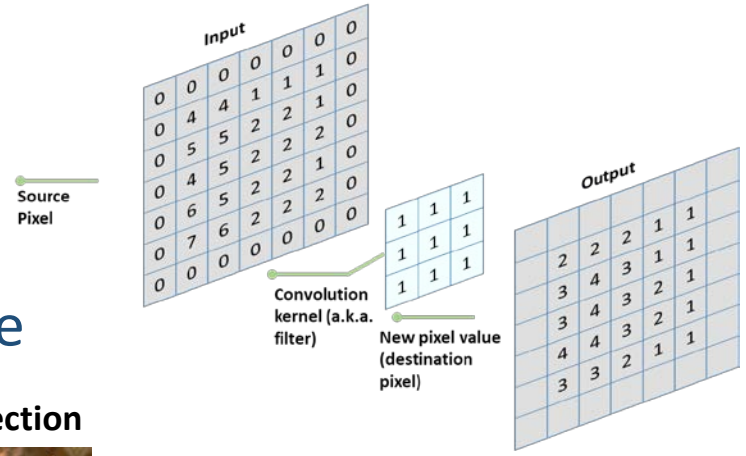
$$\frac{1}{9}(\underbrace{1*2+1*1+1*0}_{\text{Top}}+\underbrace{1*2+1*2+1*0}_{\text{Middle}}+\underbrace{1*0+1*0+1*0}_{\text{Bottom}})=0.8\approx 1$$

Convolution



Some Details about Convolution

- Notice that our result is just another image, but we lost the border
  - We could fix this by *padding* our original image with more pixels:



Zero Padding

Boundary Replication

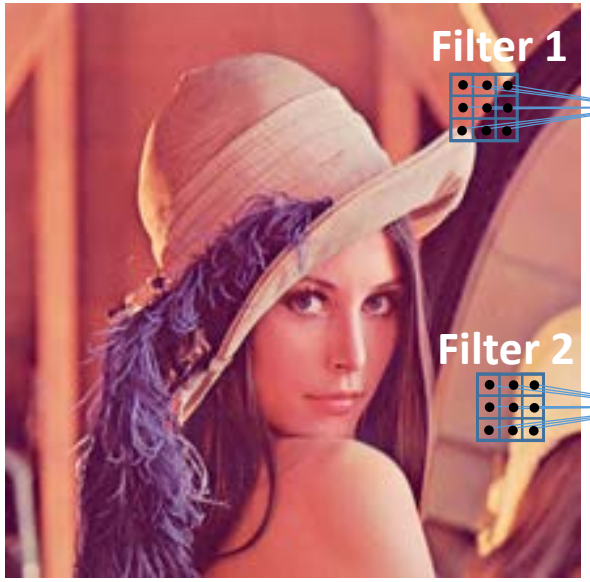
Boundary Reflection

Source: S. Marschner

- If we wanted to get an even smaller output image, we could opt to move our filter to only every other pixel, every third pixel, etc. We call this the *stride*

Multiple Filters

- We will usually want to detect several different types of features at a particular layer of a CNN
  - We use different filters for different features
- Each filter produces a different feature map



Detects horizontal edges

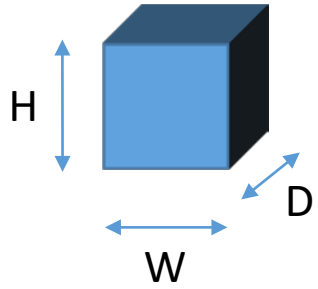
Detects vertical edges

Multiple Filters

- We can think of the output of a CNN layer as 3D, where the 3rd dimension is the number of feature maps or *channels*:
  - x and y dimensions depend on the filter sizes and padding

CNN Layer Output:

Note that we will need 3D CNN filters at the next layer!

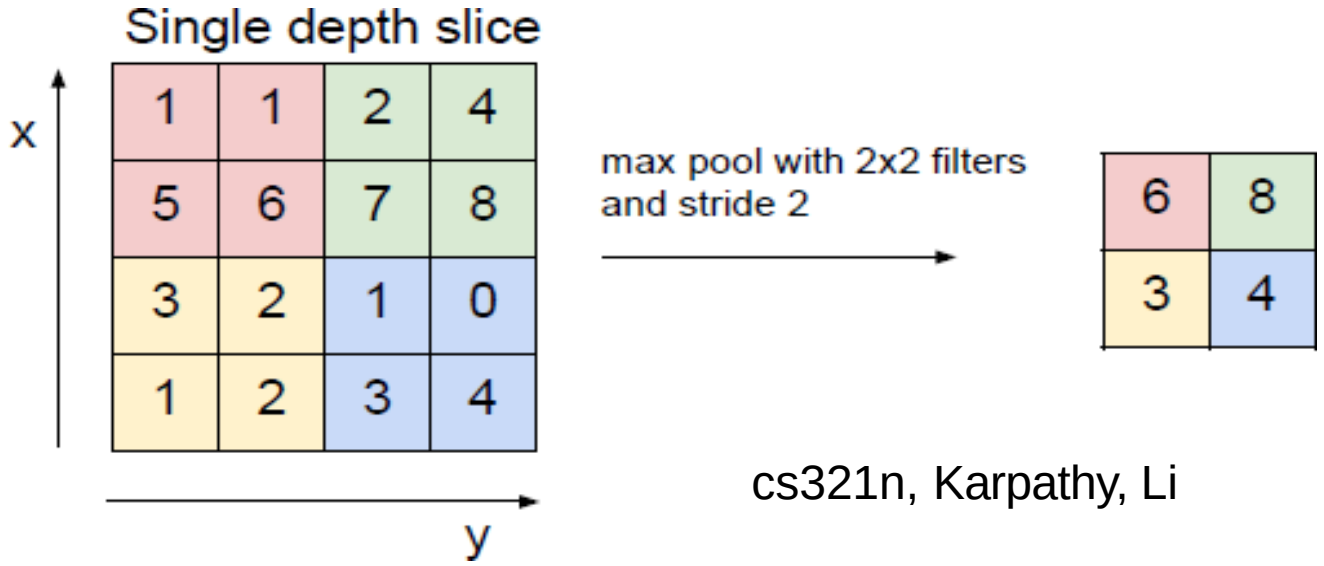


Basheer et al., “Digital Image Sobel Edge Detection Using FPGA”

Spatial Pooling

- We've seen that shared weights in CNN filters give us equivariance of features
- We can also achieve translational *invariance* by *pooling* the outputs of several neighboring filter outputs:
  - Example, max pooling:

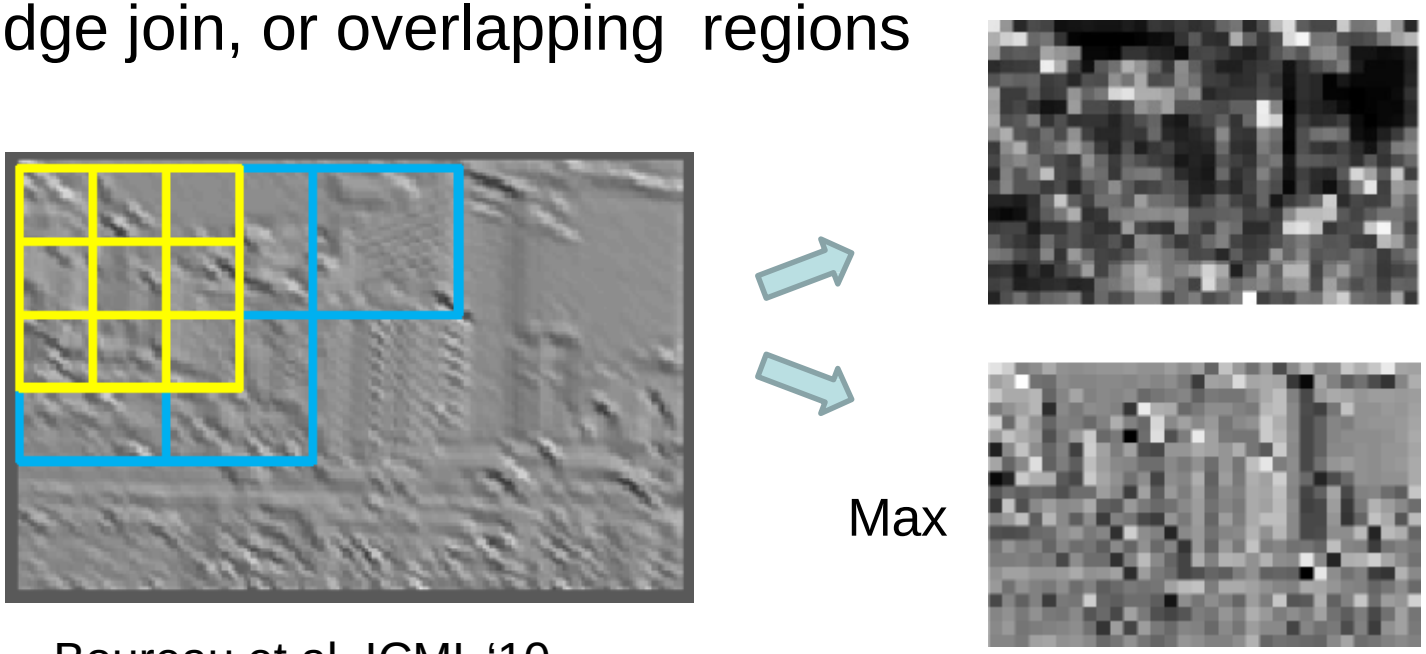
- The next CNN layer loses information about the *exact* location of a feature, but instead knows about the approximate location of the feature
  - Why might this be useful?



cs321n, Karpathy, Li

Spatial Pooling

- Several neighboring features combined into a single feature: Max Pooling or Average Pooling
- Improves invariance to transformation and noise, yields more compact representations
- Sparse, edge join, or overlapping regions

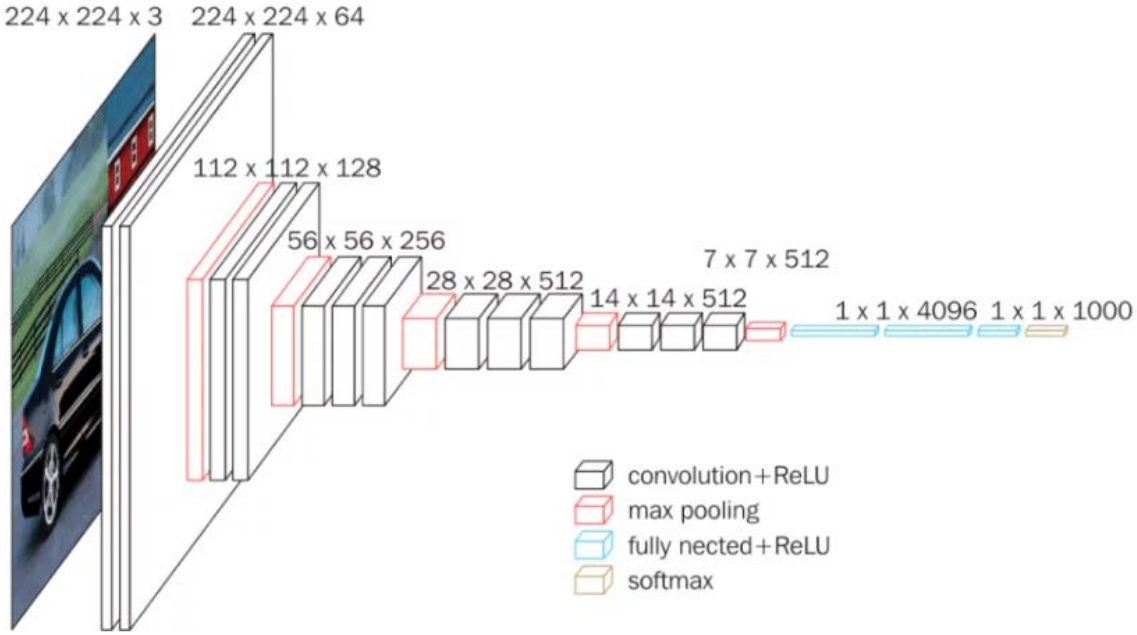


Boureau et al. ICML '10

Putting it All Together

- VGG-16 Network
  - 3x3 filters w/ stride=1 and *same* padding
  - 2x2 max pooling w/ stride=2

- Lots of parameters, but far fewer than a fully-connected network!

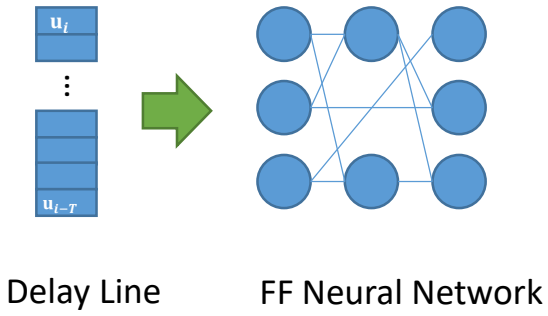


https://neurohive.io/en/popular-networks/vgg16/

VGG16 - Structural Details													
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													138,423,208

Recurrent Neural Networks

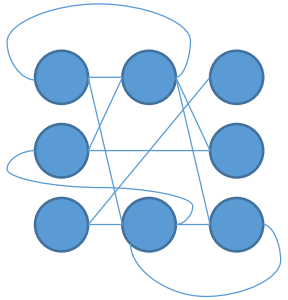
- What if we want to learn a function $\hat{\mathbf{y}} = h(\mathbf{u}, t)$?
  - So far, we have focused on learning functions where inputs are purely spatial
  - Now, we move to the more general case of *spatiotemporal* functions
- One approach is to use *tapped delay lines*



  - We may not know how large of a time window is needed

Recurrent Neural Networks

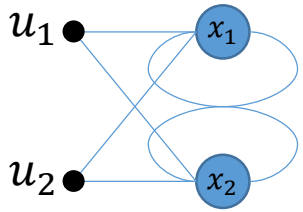
- Recurrent neural networks allow us to use feedback connections to maintain a short-term memory so that inputs at some time t can be compared to inputs from previous timesteps



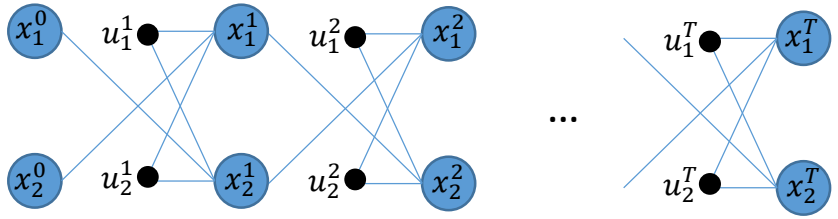
- But how can we train this?
  - Updating a particular weight with backpropagation depends on the current activity of the network, which also depends on its past activity!

Backpropagation Through Time

- Consider the following recurrent neural network



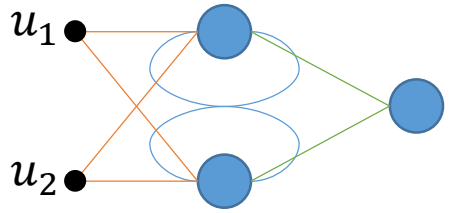
- We can transform this to a feedforward neural network by "unrolling" it in time:



  - Now, we force the weights in each layer to be the same
  - Note that we may have a target output at each time step or only at the end of a period T
  - Still need to define T, which may be very large:
    - Could have vanishing/exploding gradients

Another Way…

- The difficulties in training recurrent neural networks led to the development of “reservoir computing,” where we leave the recurrent connections (reservoir layer) of the network untrained (with random initialization)



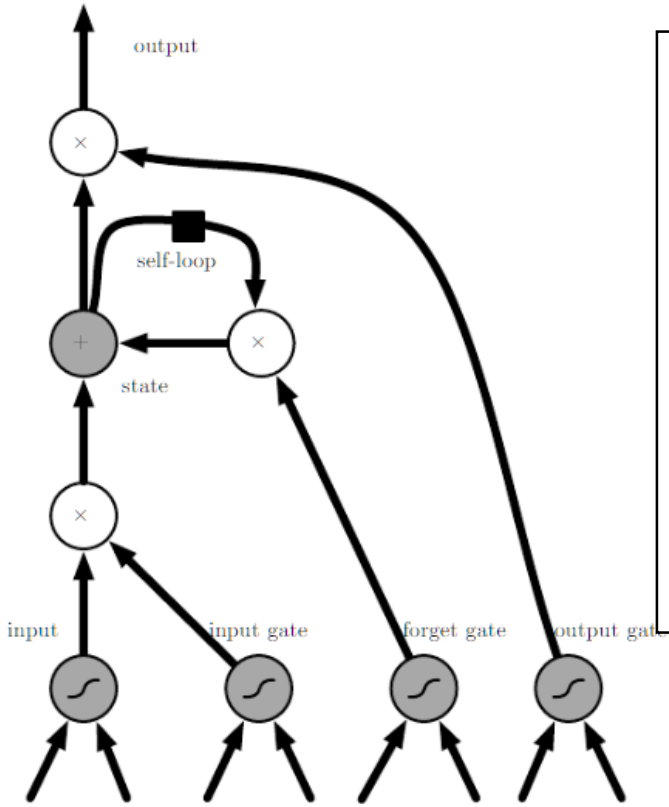
Input “Reservoir” Output

- The idea is that the random recurrent layer will extract random spatiotemporal features from the input
  - If the recurrent layer is large enough, there’s a high probability that the random features will be linearly separable
    - ❑ Can use linear regression at the output layer
    - ❑ Input and reservoir weights are left untrained → Only output layer is trained!

Handling Long-Range Dependencies

- Handling long-time dependencies and precise control of temporal memory are challenging in RNNs
  - One approach is to add explicit (gated) memory units

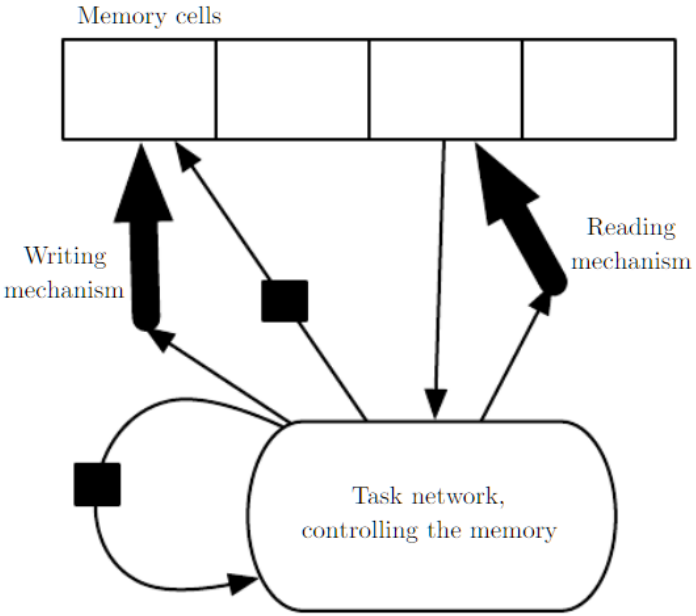
Long Short Term Memory Cell



Replace each neuron in the network with an LSTM cell. Cell has "gates" that control what gets stored or sent to other cells. Distributed memory.

Memory Augmented Neural Networks

Add an explicit memory unit and train the network how to read and write from/to it. These types of networks (e.g. Neural Turing Machines) can learn algorithms!



Goodfellow, et al., *Deep Learning*