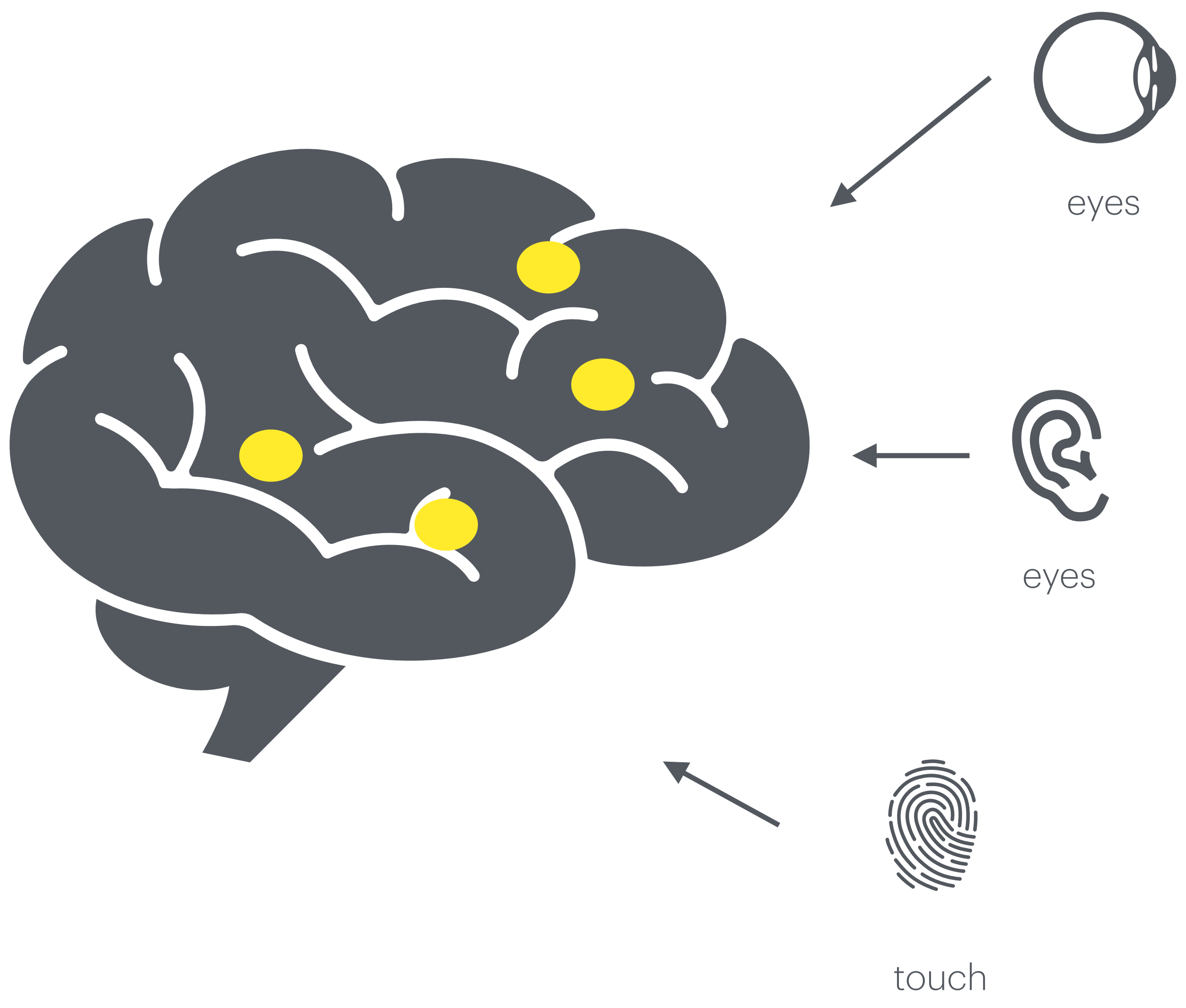
Deep Computer Vision
Convolutional Neural Networks

Perception



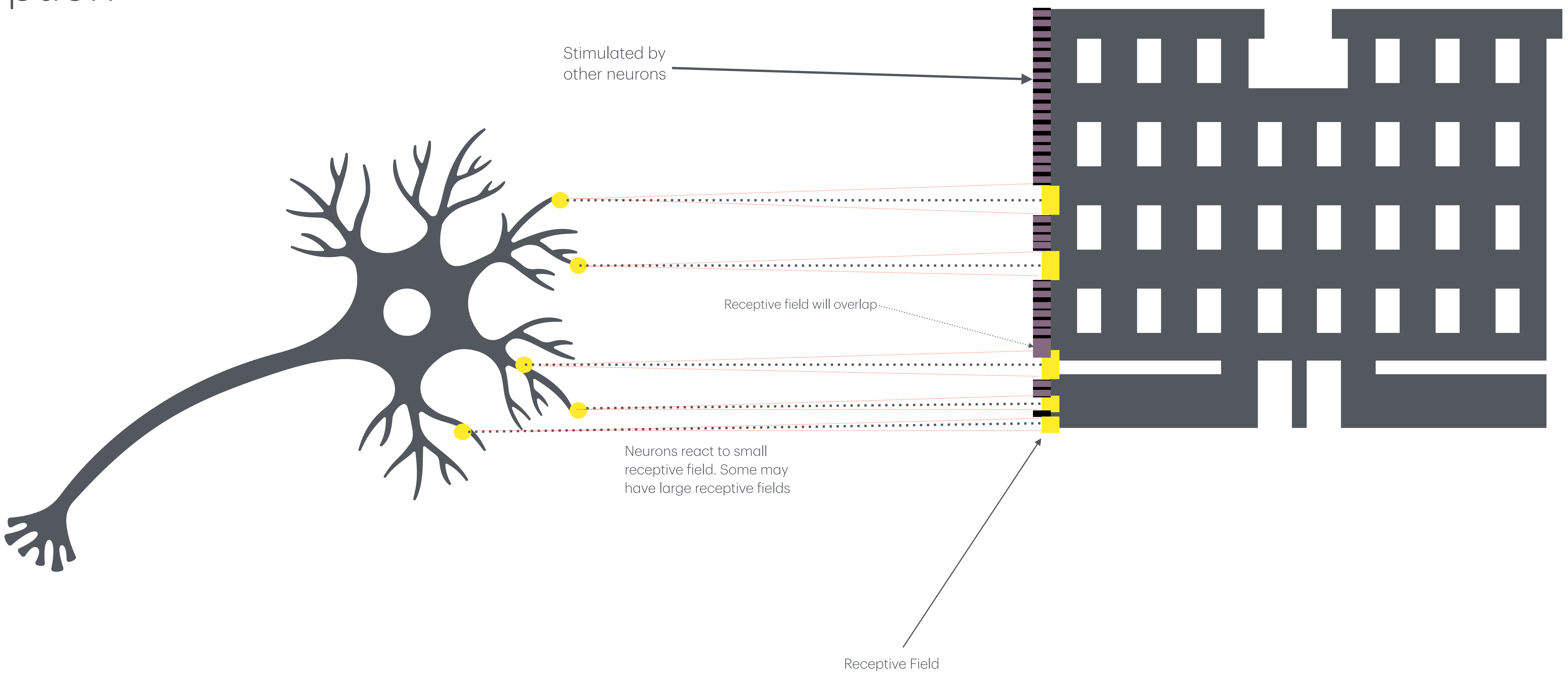
Color, Size, Volume, Height, Details

Decibels, high frequency, low frequency

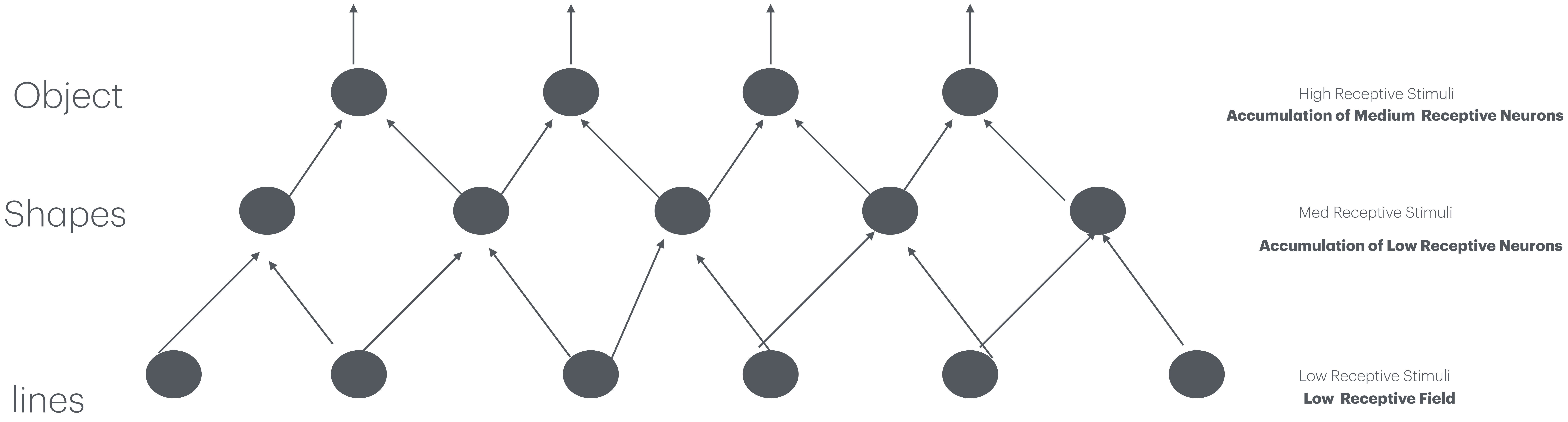
Ridged, smooth, wet, sticky, rough, sharp

Sensory information formulates high level features

Perception

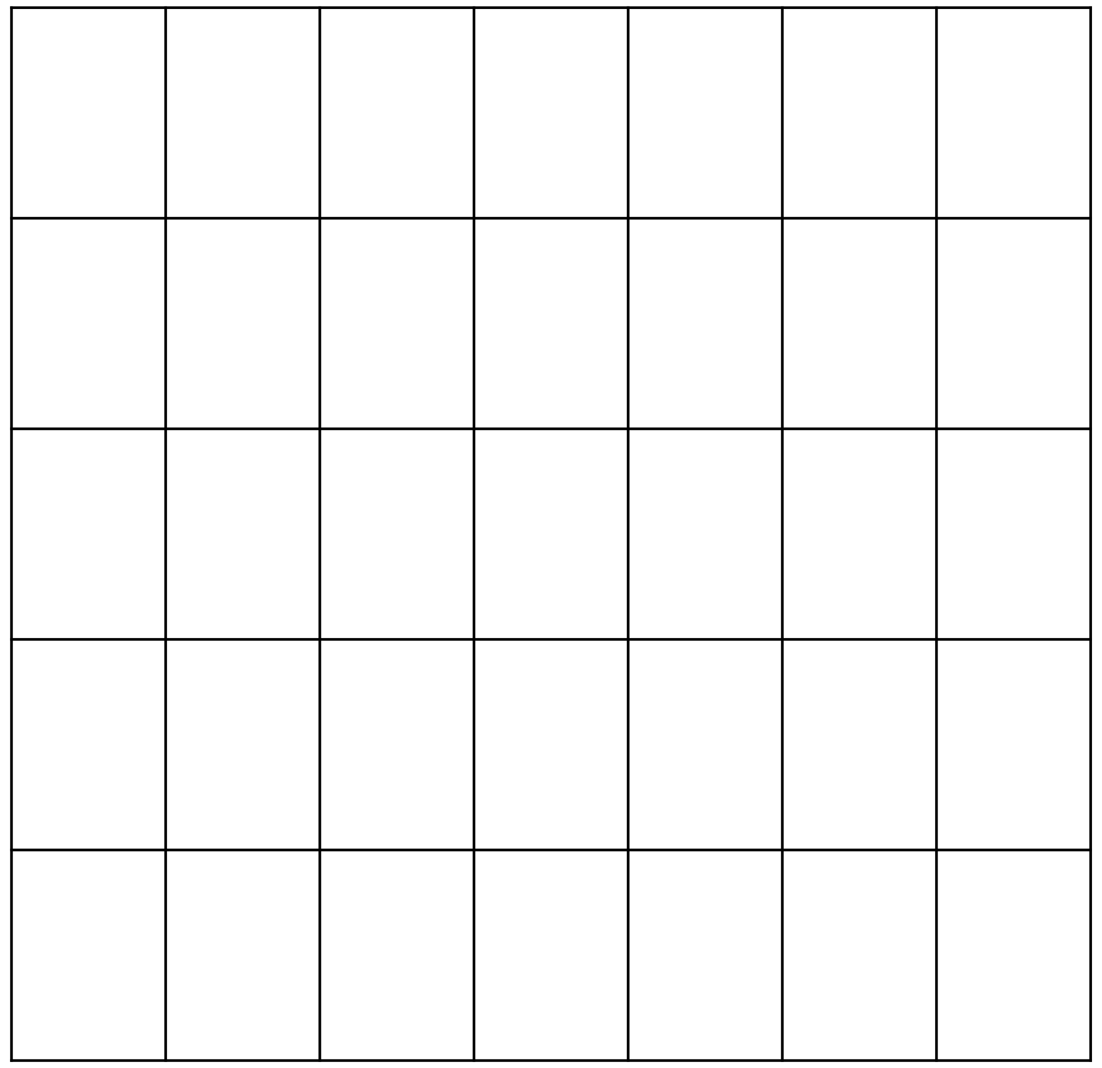


1981, Nobel Prize winner showed neurons at higher levels have higher receptive fields stimulated by complex patterns with sensory data from lower-level patterns (lower layer neurons)

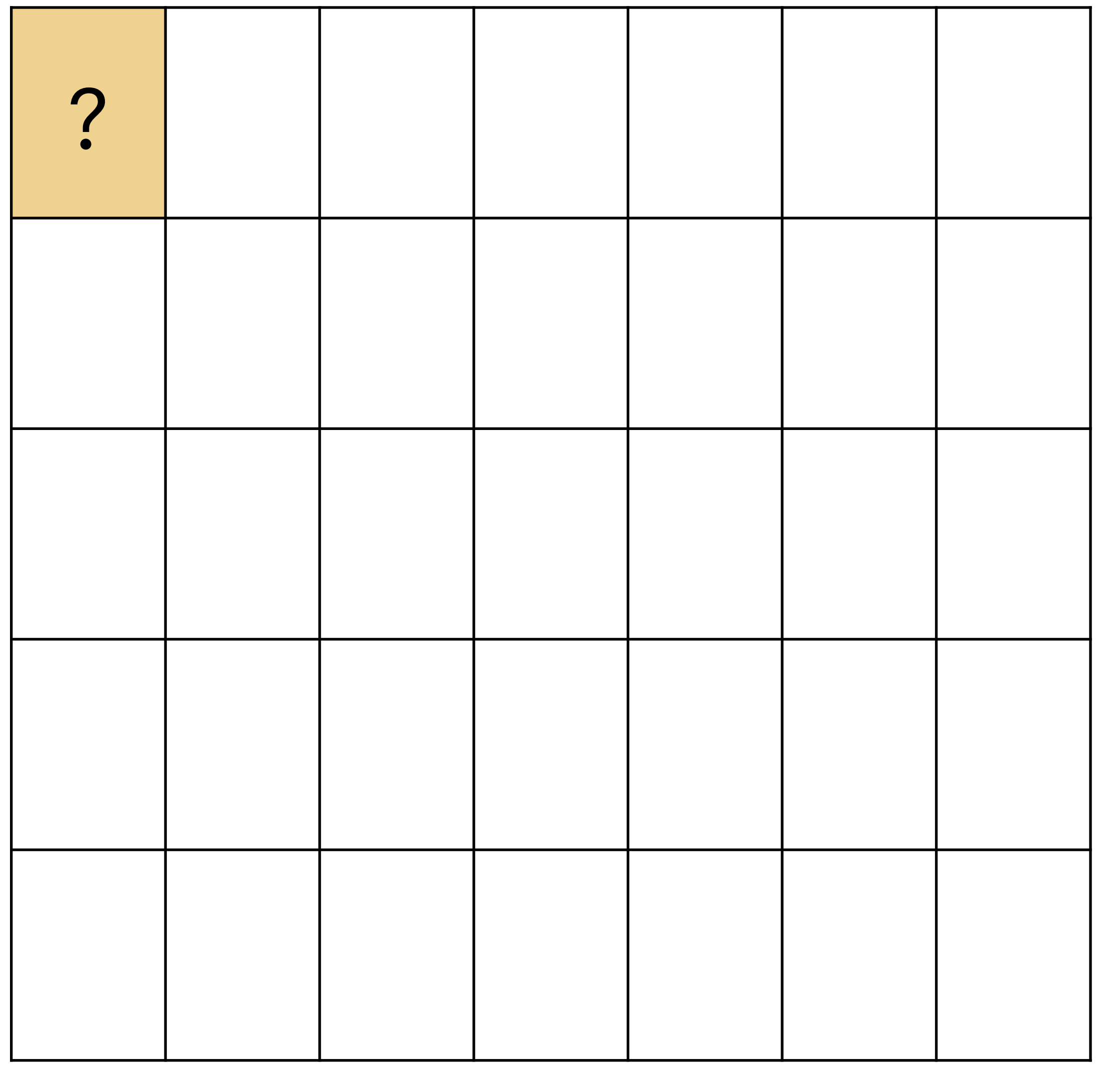

Object
Shapes
lines
High Receptive Stimuli
Accumulation of Medium Receptive Neurons
Med Receptive Stimuli
Accumulation of Low Receptive Neurons
Low Receptive Stimuli
Low Receptive Field

Receptive Field

Input (layer 0)



Hidden (layer 1)

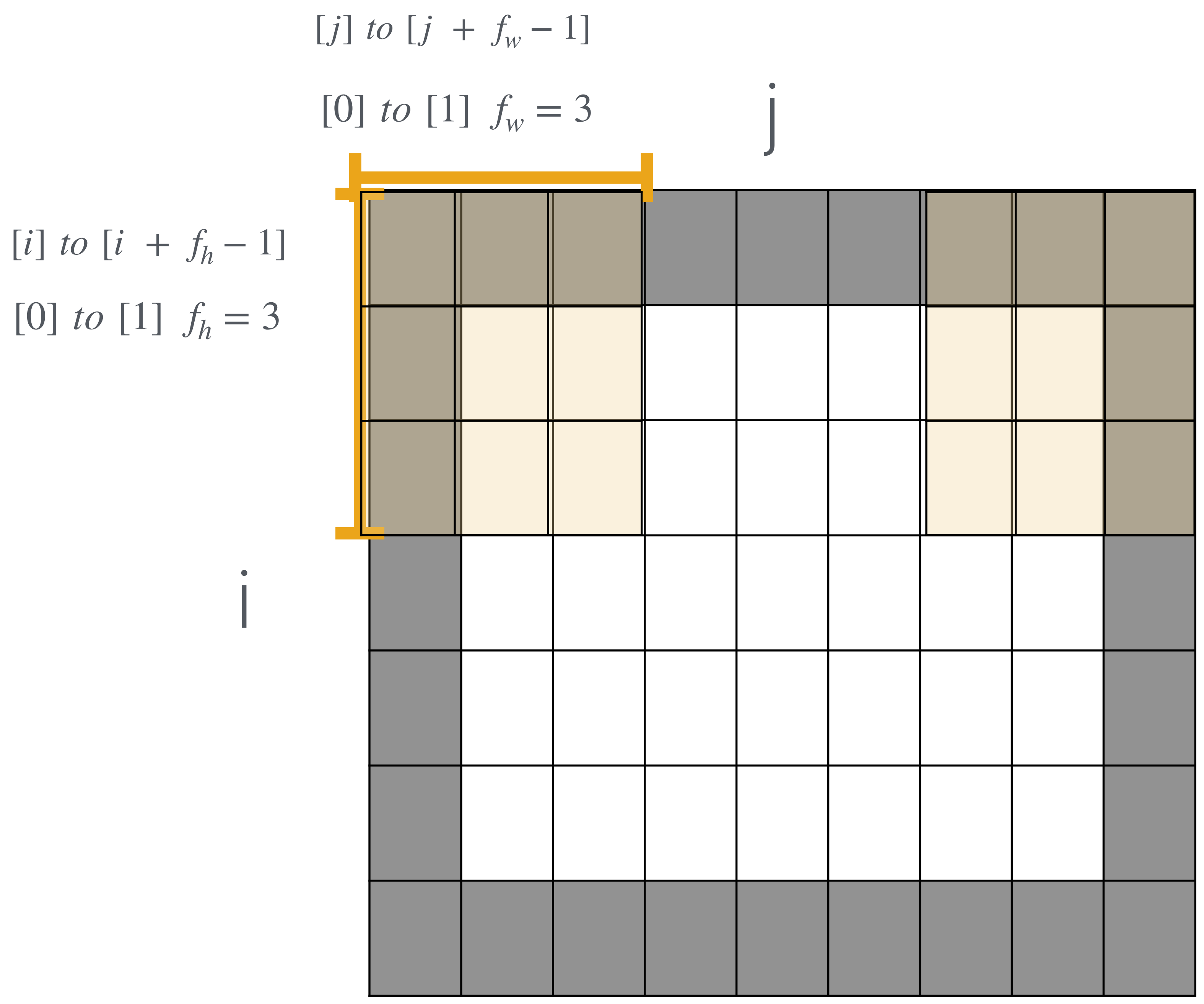


Each cell represents a neuron

Each cell in upper layer is connected to cell(s) in lower layer. Span is controlled by $f_h f_w$ receptive field height and width respectively

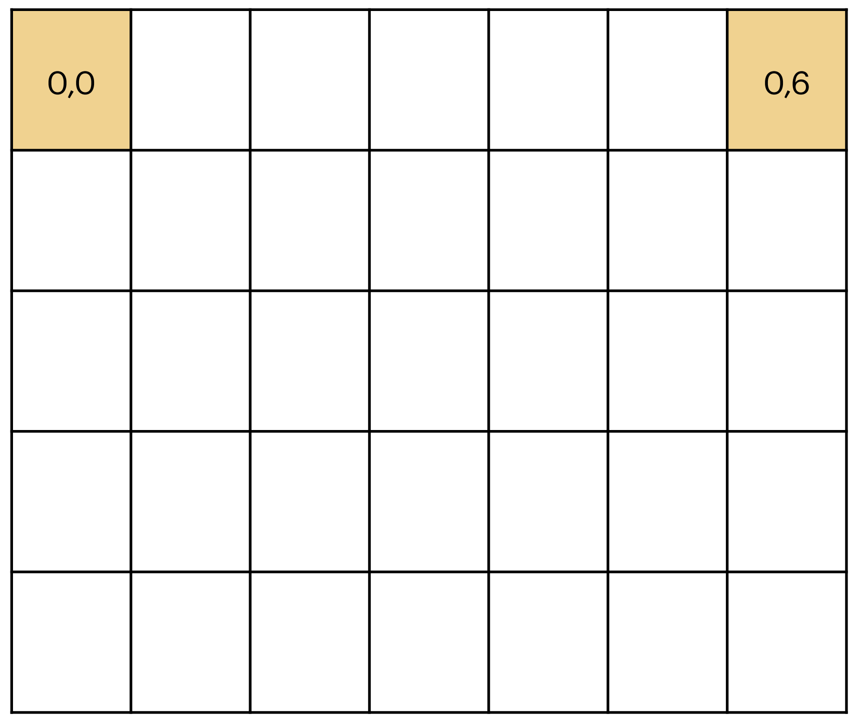
Input (layer 0)

Hidden (layer 1)

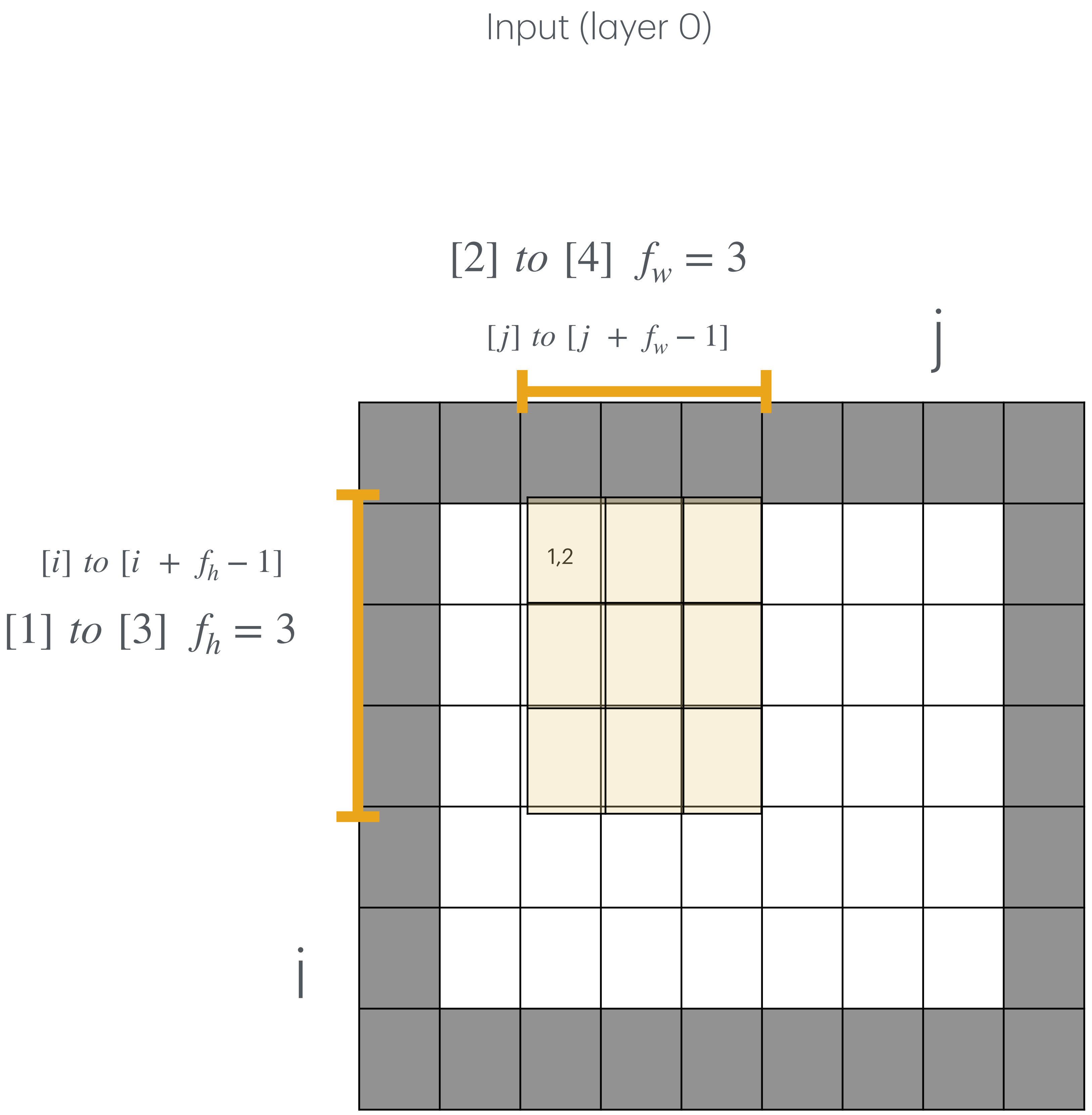


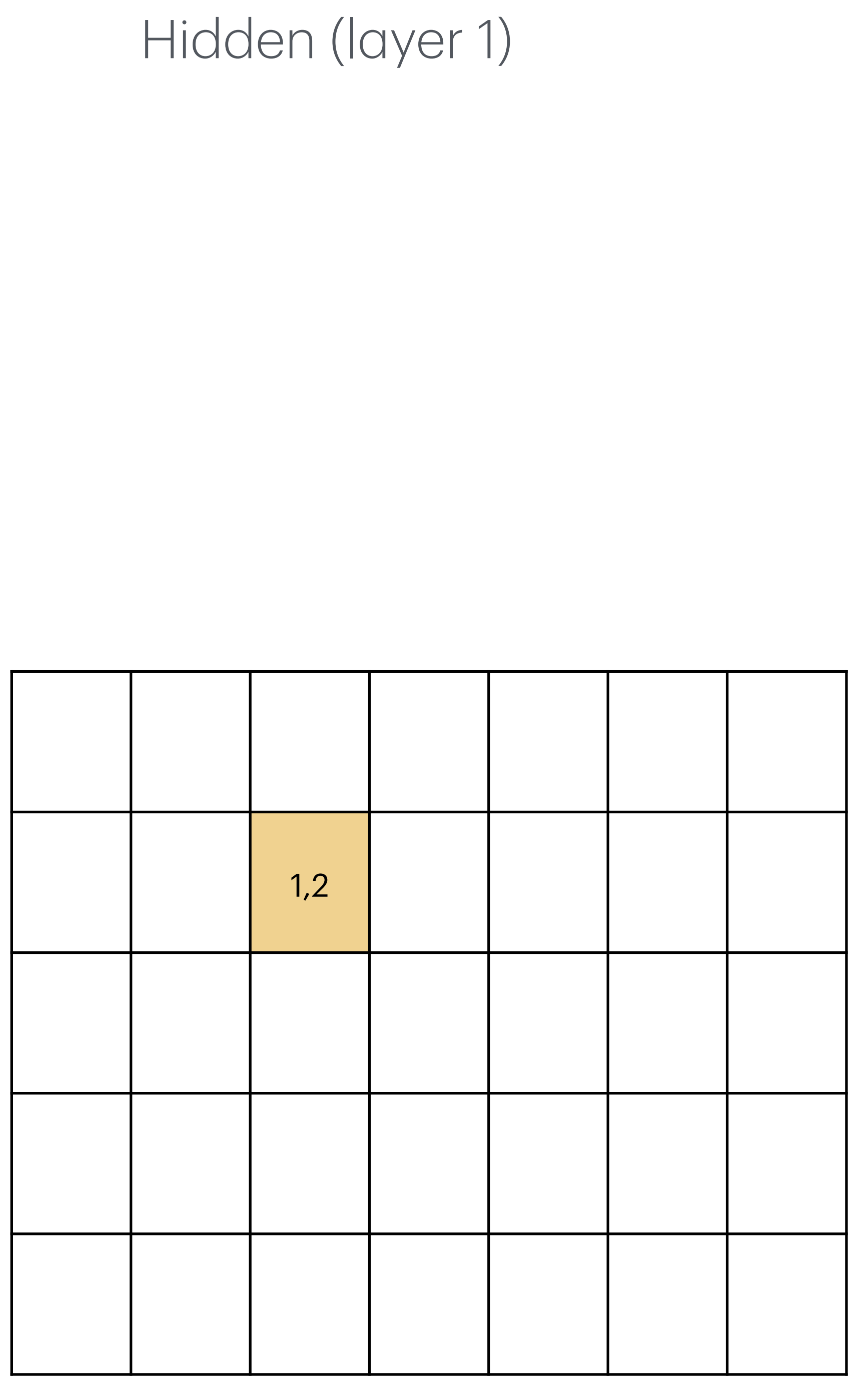
5x7 Layer w/o padding

7x9 Layer w/ padding



Zero padding a layer could make all neurons receptive fields fill previous layer





Width cells required = f_w + num_columns - 1. = 3 + 7 - 1 = 9

Height cells required = f_h + num_rows- 1 = 3 + 5 - 1 = 7

Stride

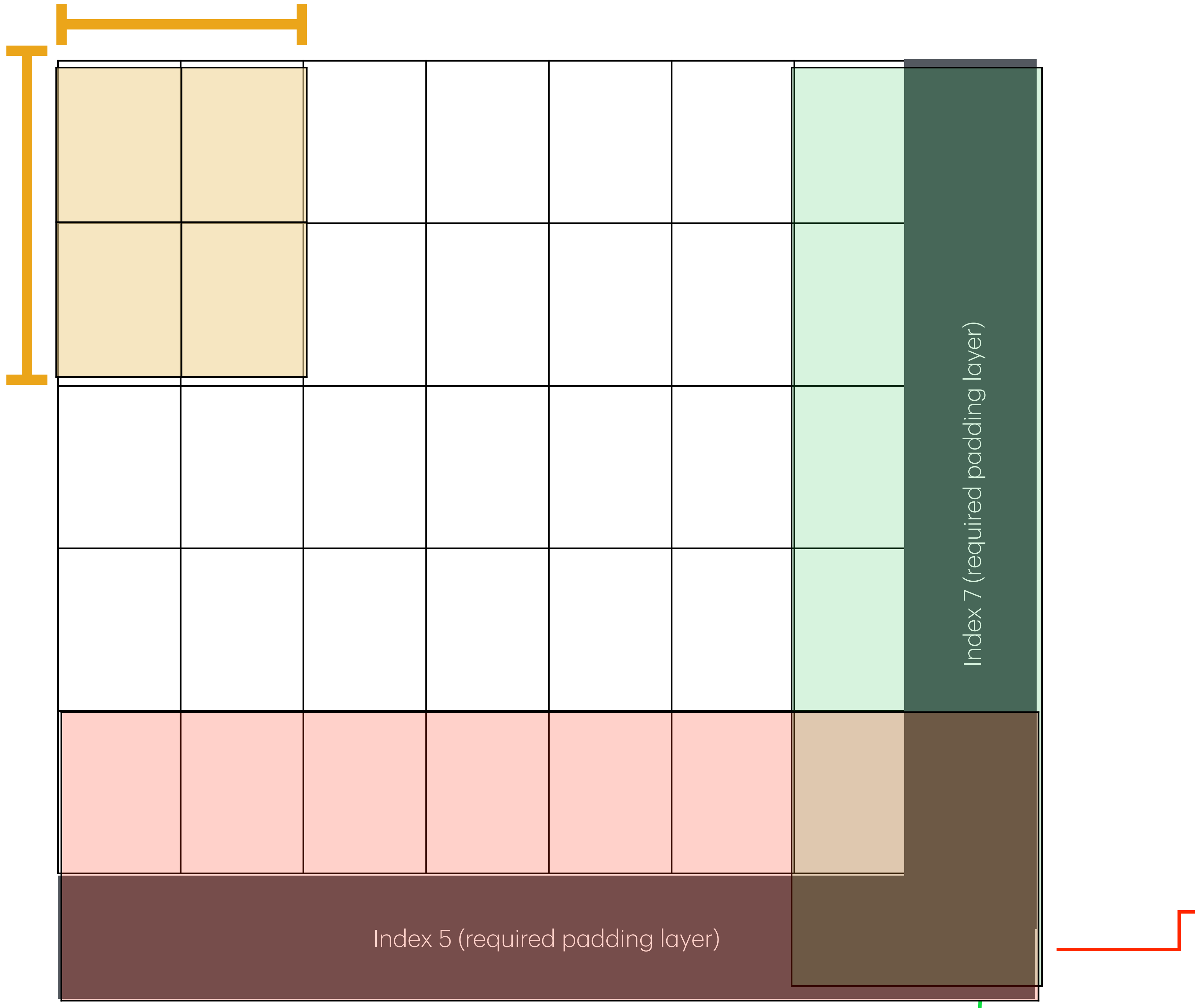
Input (layer 0)
5x7

$[0]\ to\ [1]$
$s_w = 2\ f_w = 2$

$[j \times s_w]\ to\ [j \times s_w\ +\ f_w - 1]$

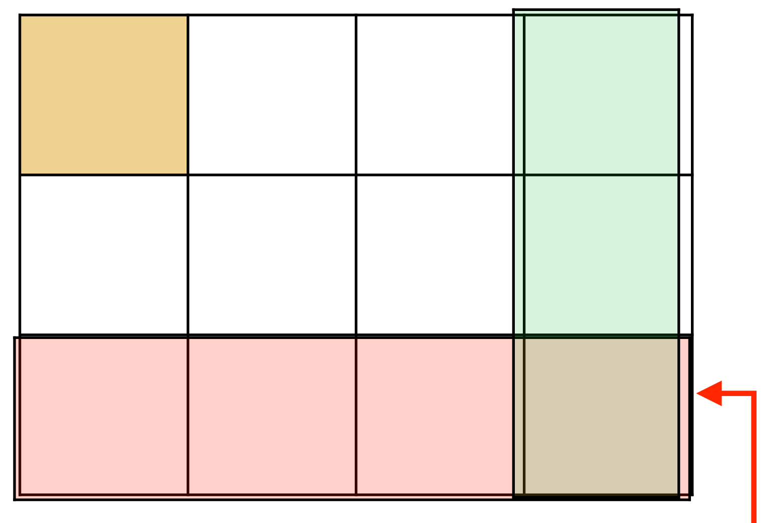
$[0]\ to\ [1]$
$s_h = 2\ f_h = 2$

$[i \times s_h]\ to\ [i \times s_h\ +\ s_h - 1]$



Hidden (layer 1)
3x4

Spacing out receptive fields allows smaller layers to ***field*** the entire lower layer using s_h, s_w height and width strides



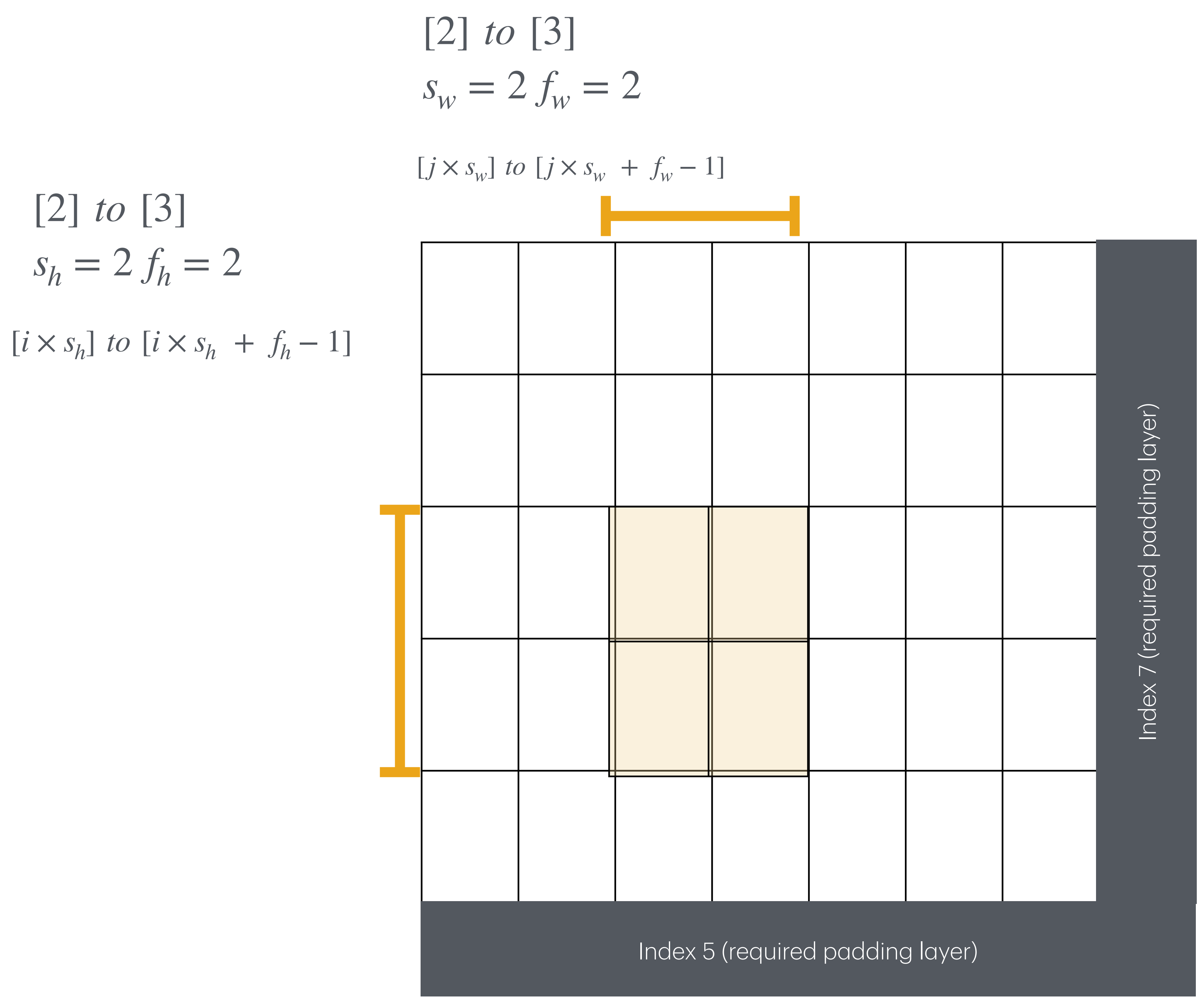
$row_{max-index} = 7$

Largest column **cell index required** < max index = 3> ==> $3_{j_{index}} \times 2_{span} + 2_{f_w} - 1$

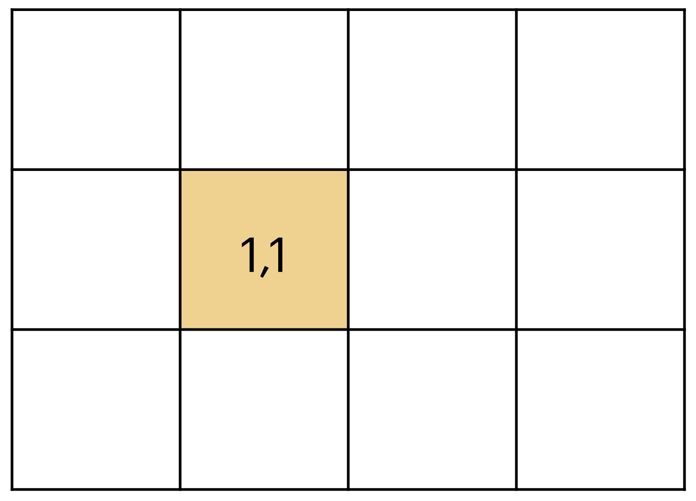
$row_{max-index} = 5$

Largest row **cell index required** < max index = 2 > ==> $2_{i_{index}} \times 2_{span} + 2_{f_h} - 1$

Input (layer 0)
5x7

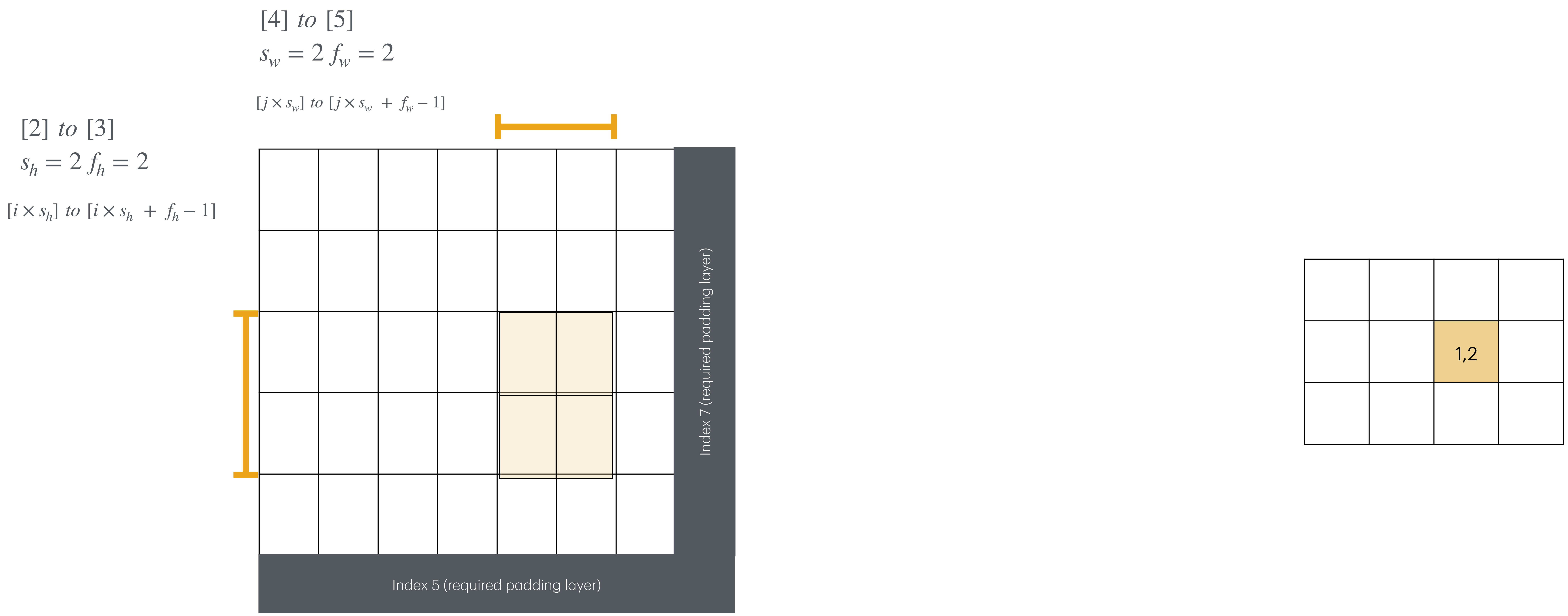


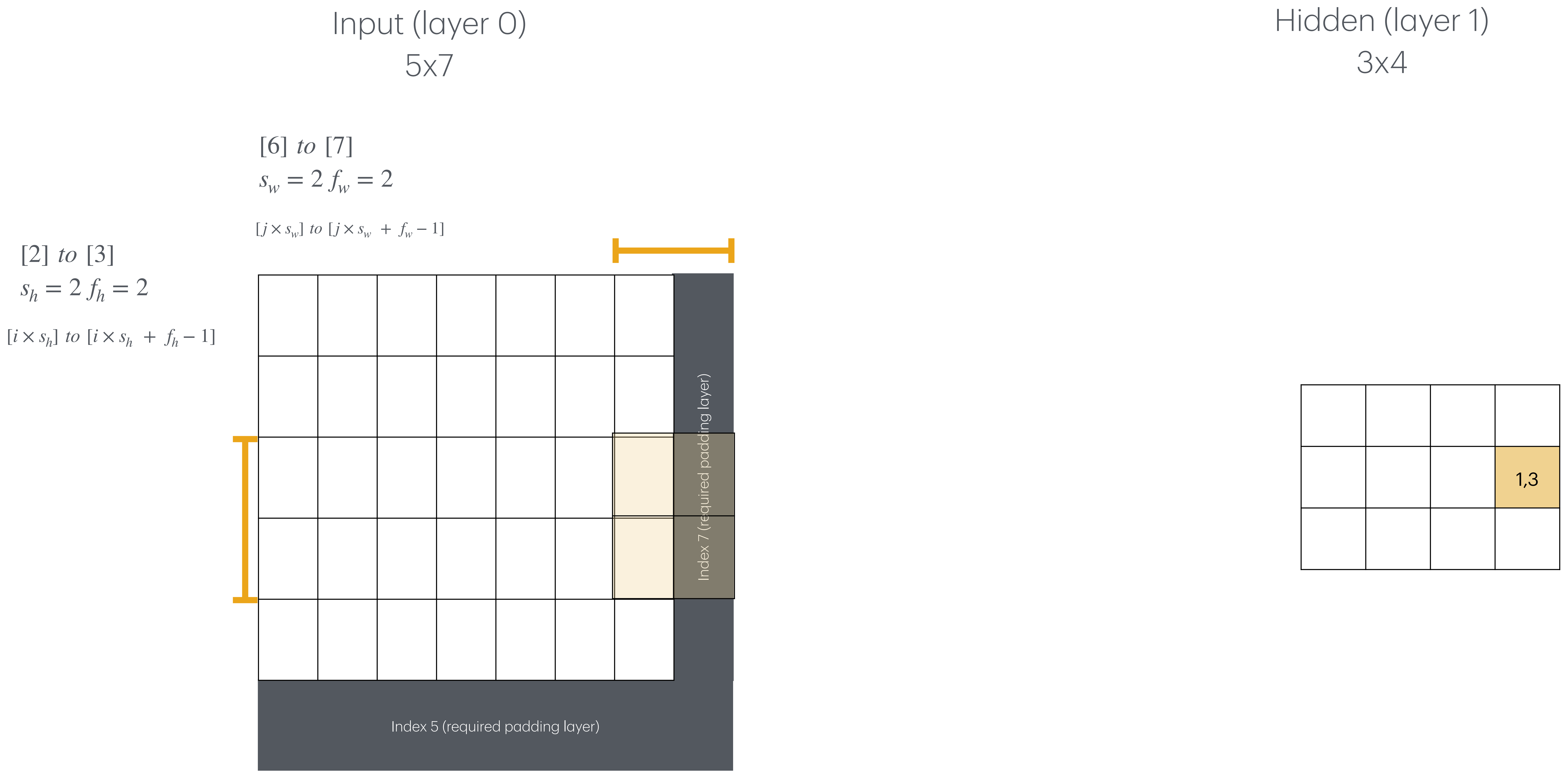
Hidden (layer 1)
3x4



Input (layer 0)
5x7

Hidden (layer 1)
3x4





Input (layer 0)
5x7
[6] to [7]
$s_w = 2 \; f_w = 2$
$[j \times s_w]$ to $[j \times s_w + f_w - 1]$
[2] to [3]
$s_h = 2 \; f_h = 2$
$[i \times s_h]$ to $[i \times s_h + f_h - 1]$
Index 7 (required padding layer)
Index 5 (required padding layer)
Hidden (layer 1)
3x4
1,3

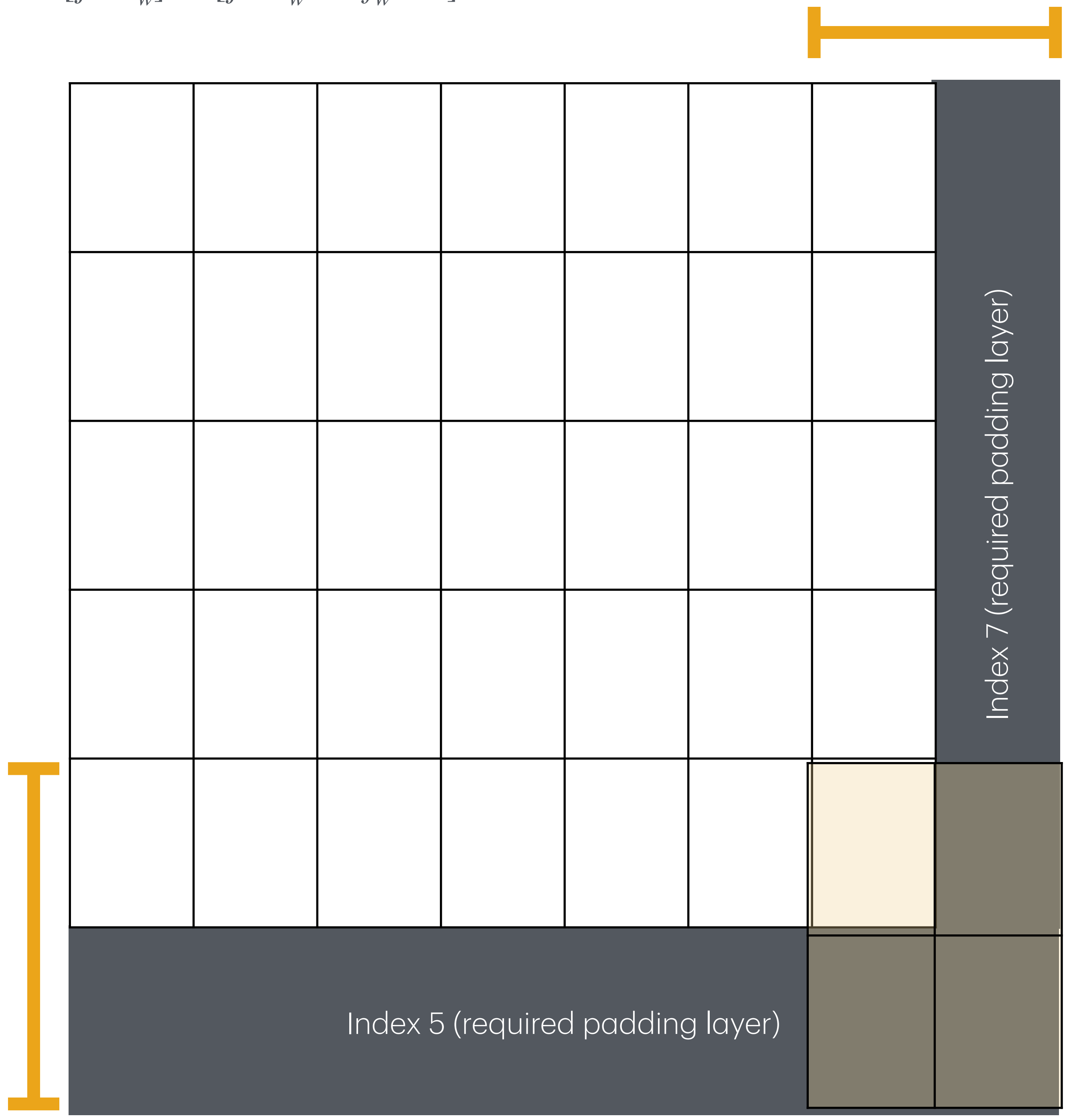
Input (layer 0)
5x7

$[6]\ to\ [7]$
$s_w = 2\ f_w = 2$

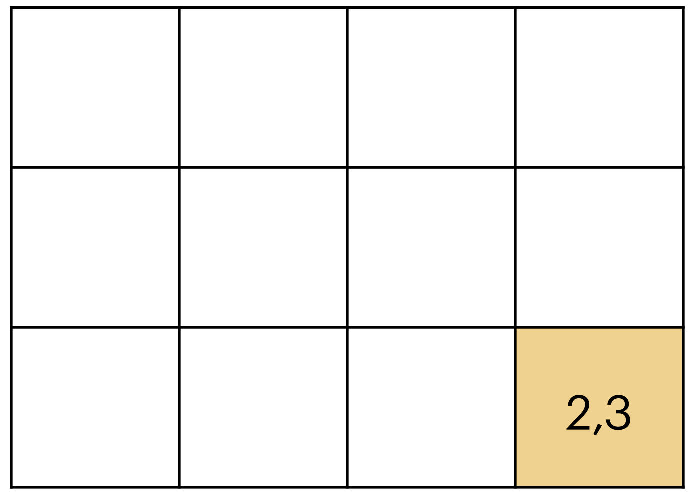
$[j \times s_w]\ to\ [j \times s_w\ +\ f_w - 1]$

$[4]\ to\ [5]$
$s_h = 2\ f_h = 2$

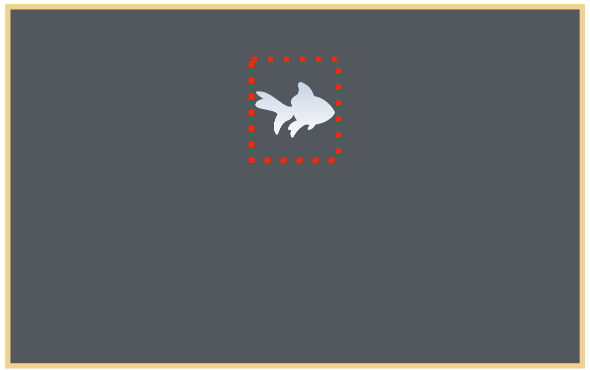
$[i \times s_h]\ to\ [i \times s_h\ +\ f_h - 1]$



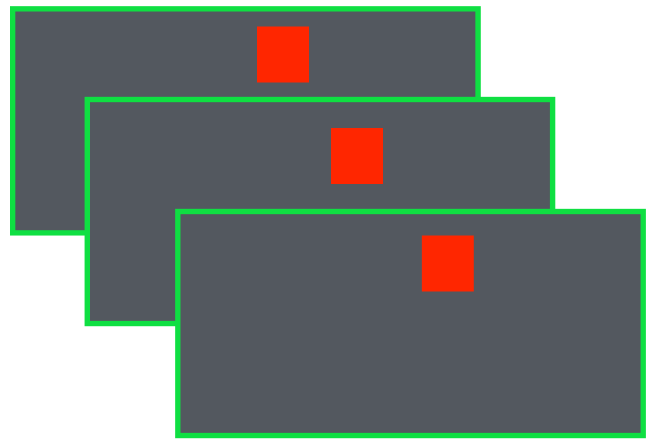
Hidden (layer 1)
3x4



Input Layer



Convolution layer is a stack of feature maps

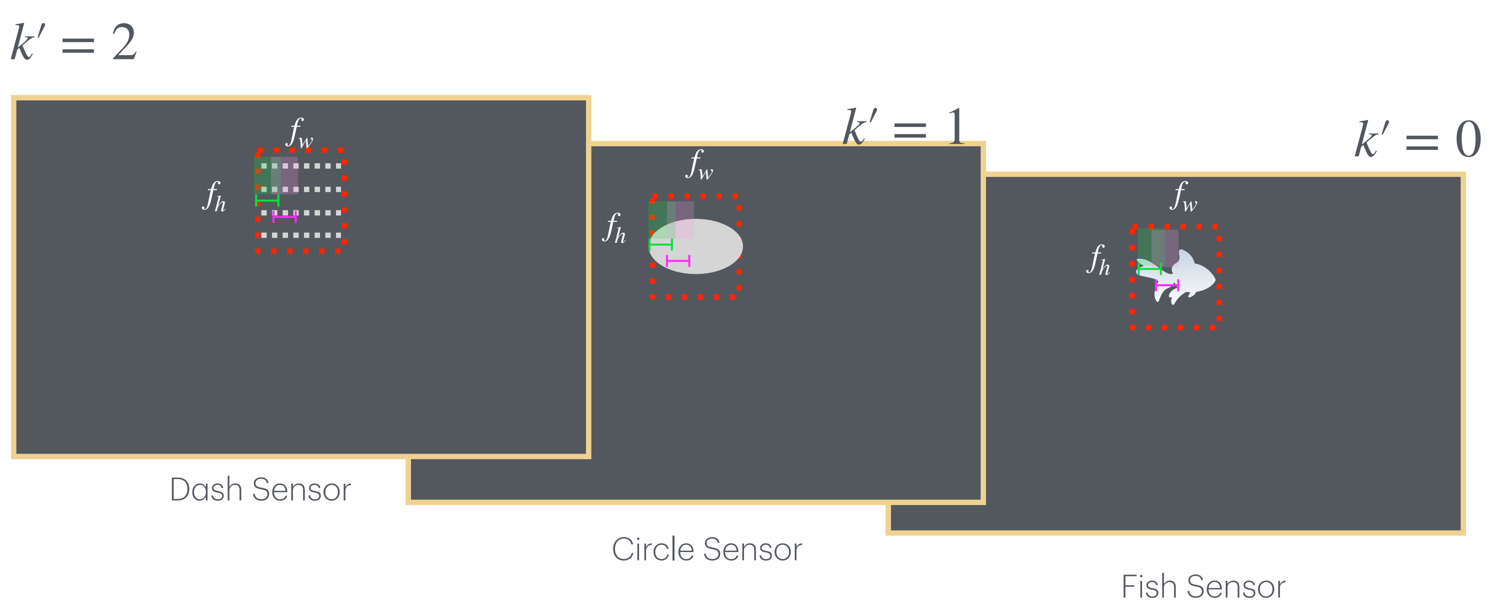


Feature map neurons at the same position map to the same receptive field in the previous layer

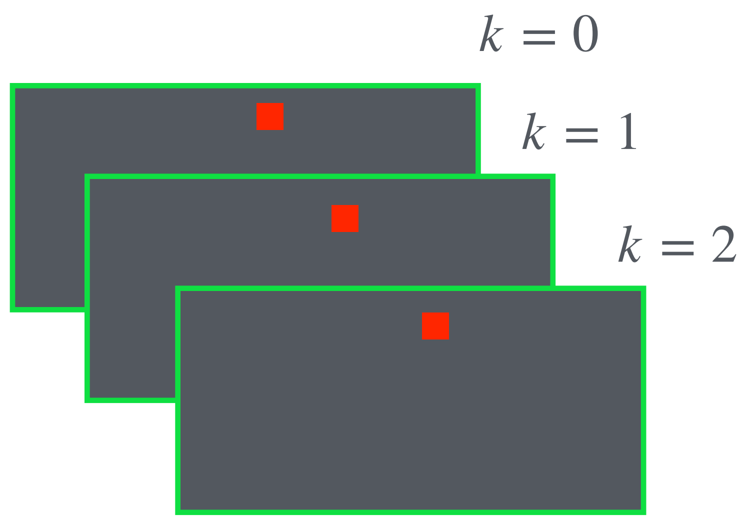
Each neuron would see and process/correlate the receptive field of the outlined fish in the input layer

Convolutional Layer /Feature Maps Layer

Convolutional Layer 1

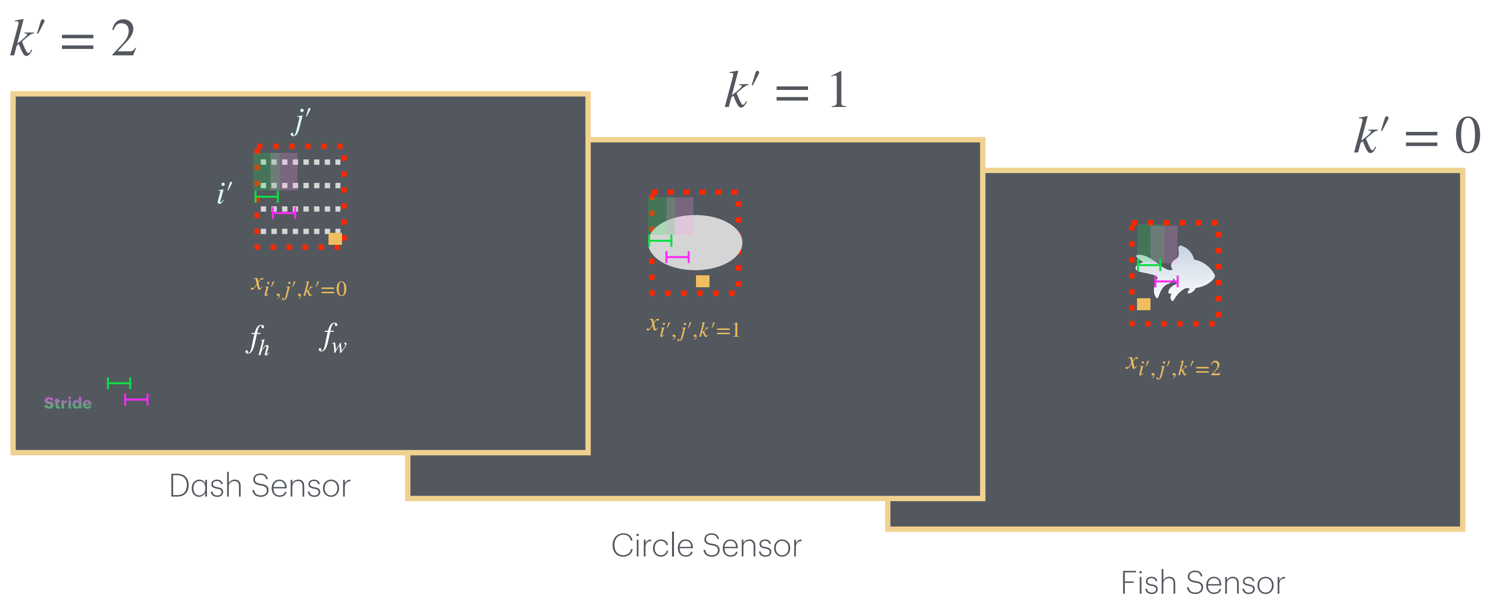


Convolutional Layer 2



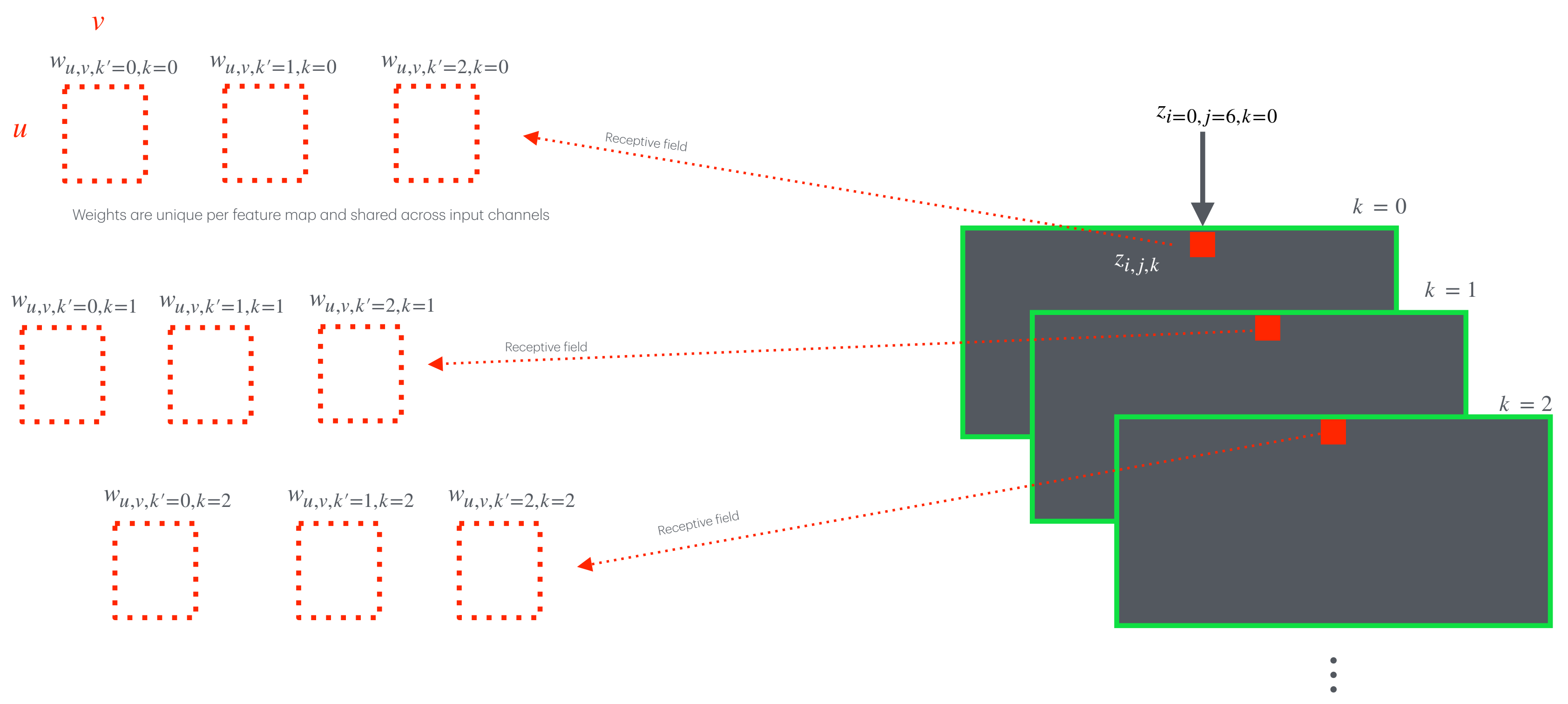
Convolutional Layer /Feature Maps Layer

Convolutional Layer 1



connections weights

Convolutional Layer 2



Possible Feature Map Sensors:
- fish in fish bowl
- fish in water
- first in a bowl with water
- etc

Output of z_{0, 6} in feature map k, assume

f_h, f_w, s_w, s_h

$z_{i,j,k}$

$$z_{i=0,j=6,k} = b_k + \sum_{0}^{f_h-1}\sum_{0}^{f_w-1}\sum_{0}^{f_n'-1} x_{i',j',k'} \times w_{u,v,k',k}$$

Weights are trainable parameters

$neurons_{featuremap-0}$
$$z_{i=0,j=6,k=0} = b_0 + \sum_{0}^{f_h-1}\sum_{0}^{f_w-1}\sum_{0}^{f_n'-1} x_{i',j',k'} \times w_{u,v,k',0}$$

$neurons_{featuremap-1}$
$$z_{i=0,j=6,k=1} = b_1 + \sum_{0}^{f_h-1}\sum_{0}^{f_w-1}\sum_{0}^{f_n'-1} x_{i',j',k'} \times w_{u,v,k',1}$$

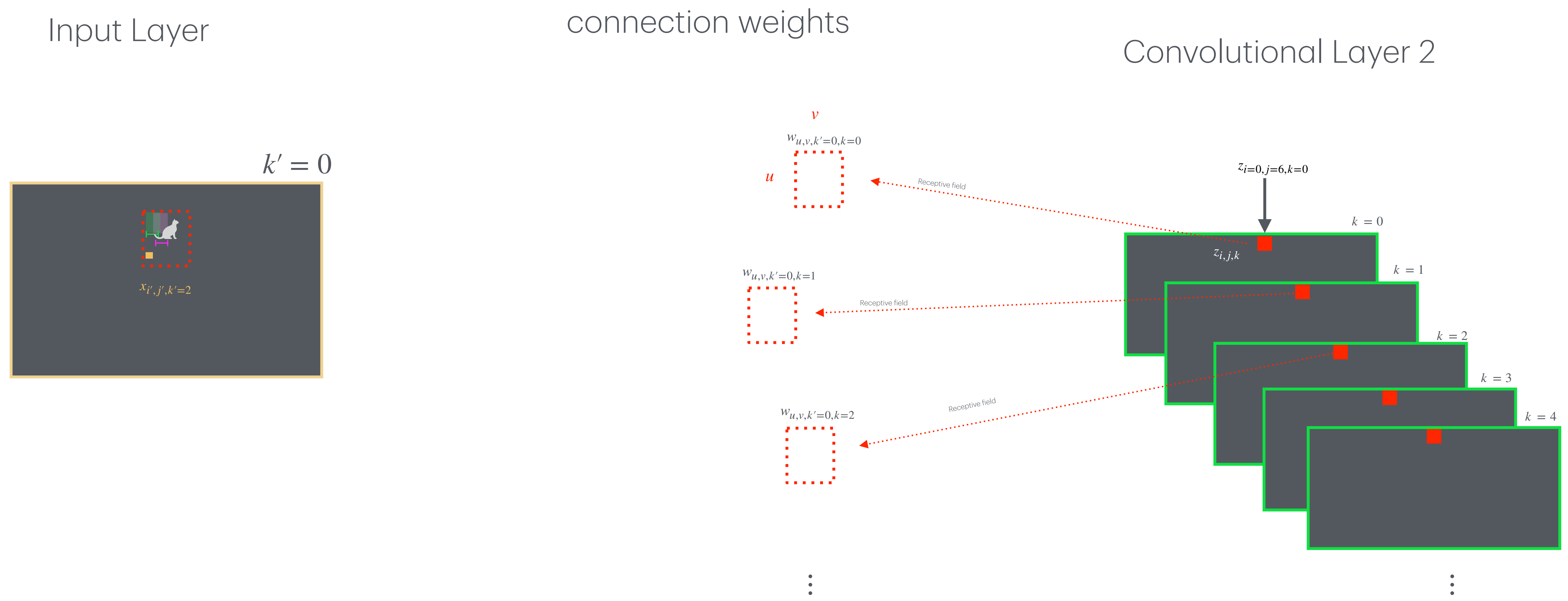
$neurons_{featuremap-2}$
$$z_{i=0,j=6,k=2} = b_2 + \sum_{0}^{f_h-1}\sum_{0}^{f_w-1}\sum_{0}^{f_n'-1} x_{i',j',k'} \times w_{u,v,k',2}$$

⋮

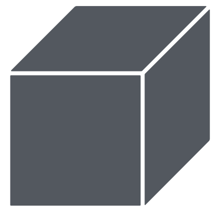
Input neuron cells per k' channel

Feature map kernel weights per k' channel

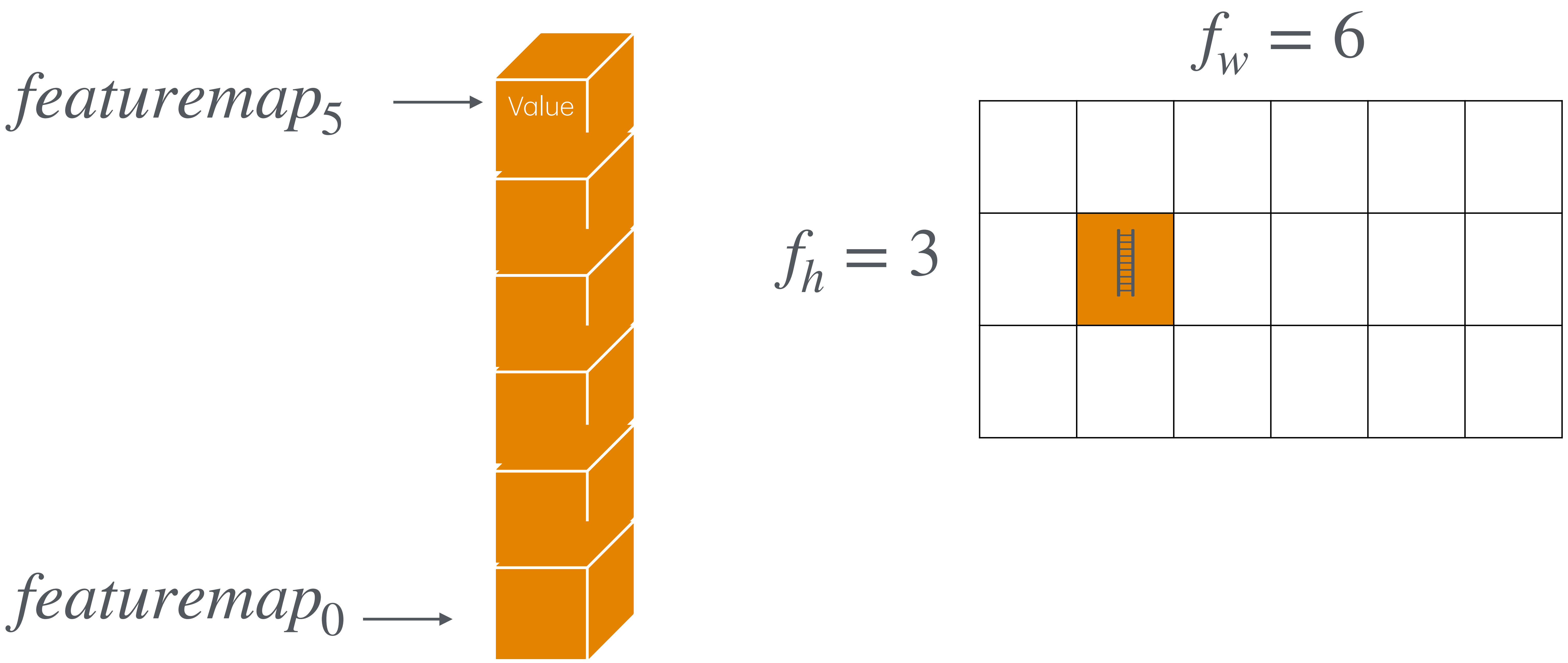
Convolutional Layer /Feature Maps Layer: Another Example



Weights Convolutional Layer

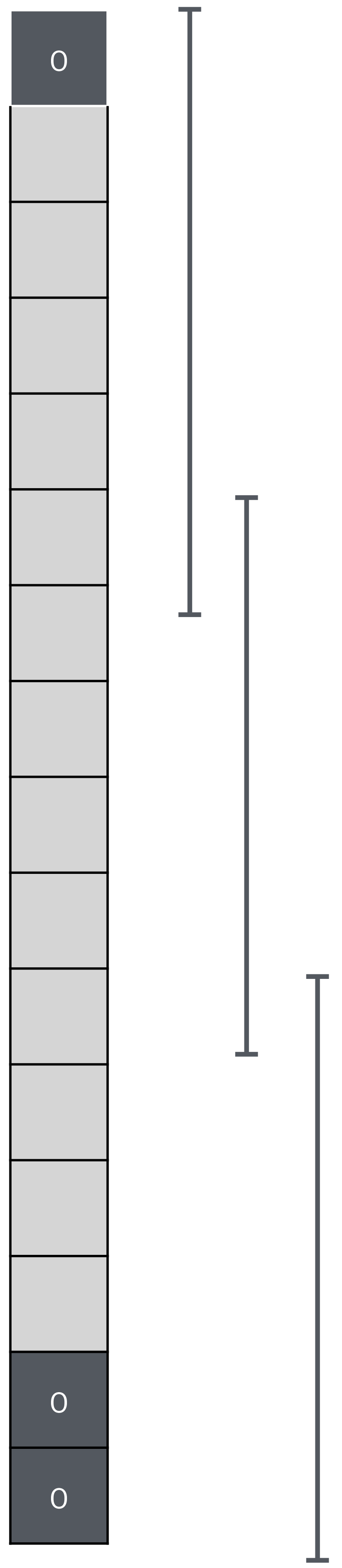


1D Tensor (neuron unit)

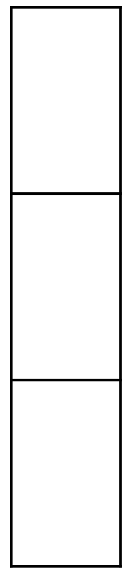


Tensorflow Padding: Same

$stride = 5 \qquad width = 6 \qquad Input_{length} = 13\ neurons$



$$output_{size} = \frac{13}{5} = 2.6 = 3$$



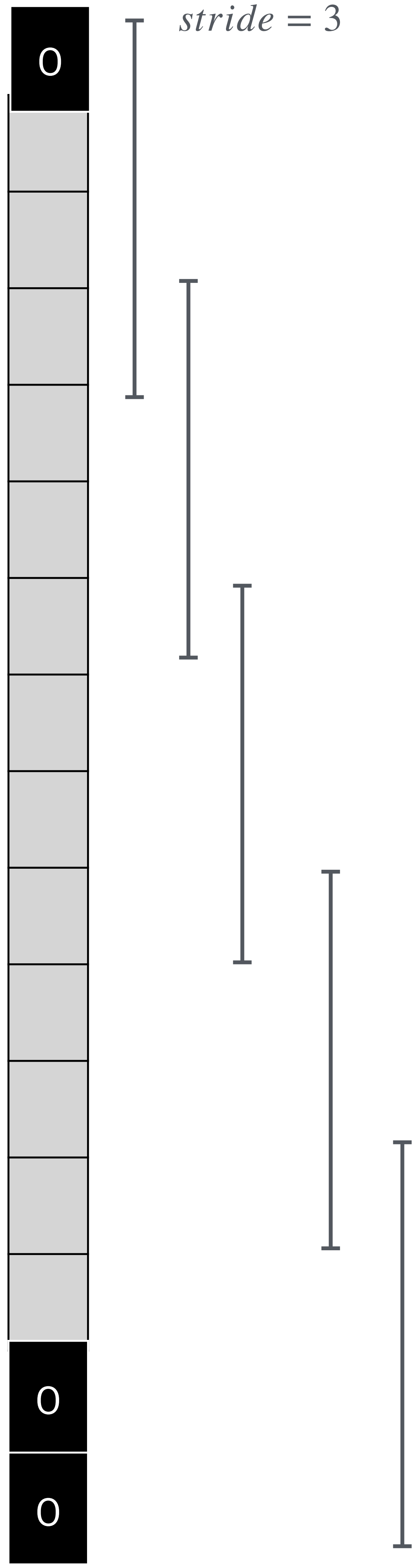
$$Input_{effective} = output_{size} \times width - (output_{size} - 1) = 3 \times 6 - (3 - 1) = 16$$

$$Input_{effectivelength} = 16\ neurons$$

$$Input_{length} = 13\ neurons \qquad \Delta = 3$$

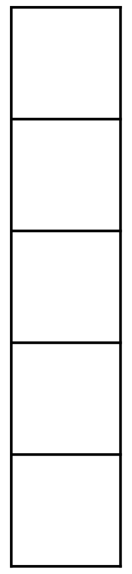
$$\begin{cases} odd_{\Delta} = floor(\frac{\Delta}{2}) \text{ zeros injected to front} \qquad ceil(\Delta/2) \text{ zeros injected to tail} \\ even_{\Delta} = \frac{\Delta}{2} \text{ zeros injected to front and tail} \end{cases}$$

Tensorflow Padding: Same



$stride = 3 \qquad width = 4 \qquad Input_{length} = 13\ neurons$

$$output_{size} = \frac{13}{3} = 5$$



Total Output Required to meet stride and receptive length requirements

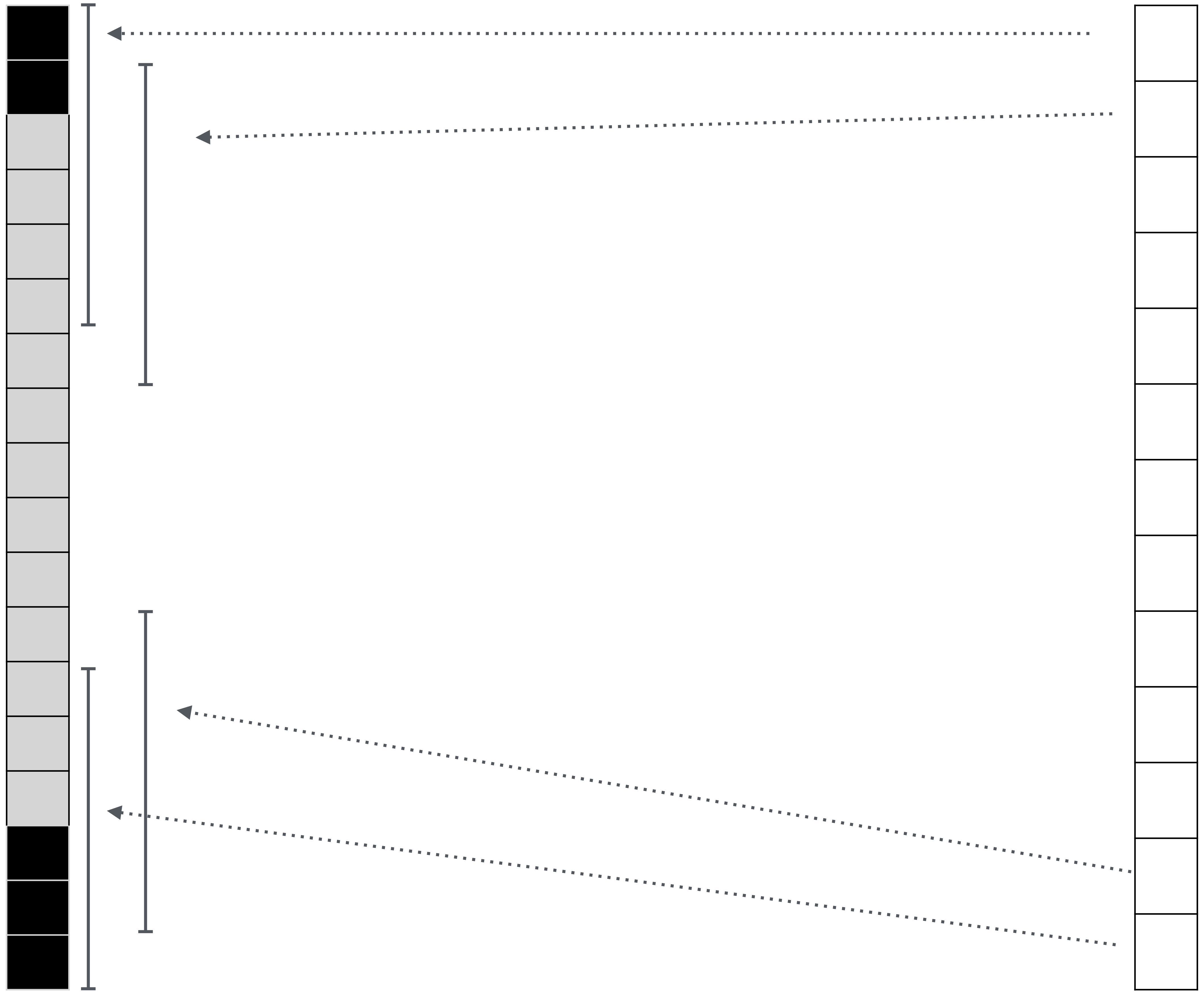
$$TotalInputNeurons = output_{size} \times width - (output_{size} - 1) = 5 \times 4 - (5 - 1) = 16$$

$$Input_{effectivelength} = 16\ neurons$$

$$Input_{length} = 13 \qquad \Delta = 3$$

Tensorflow Padding: Same

$stride = 1$ $width = 6$ $Input_{length} = 13\ neurons$ $output_{size} = \frac{13}{1} = 13$



$Input_{effectivelength} = 13 + (6 - 1) = 18$

$Input_{length} = 13$

$\Delta = 5$

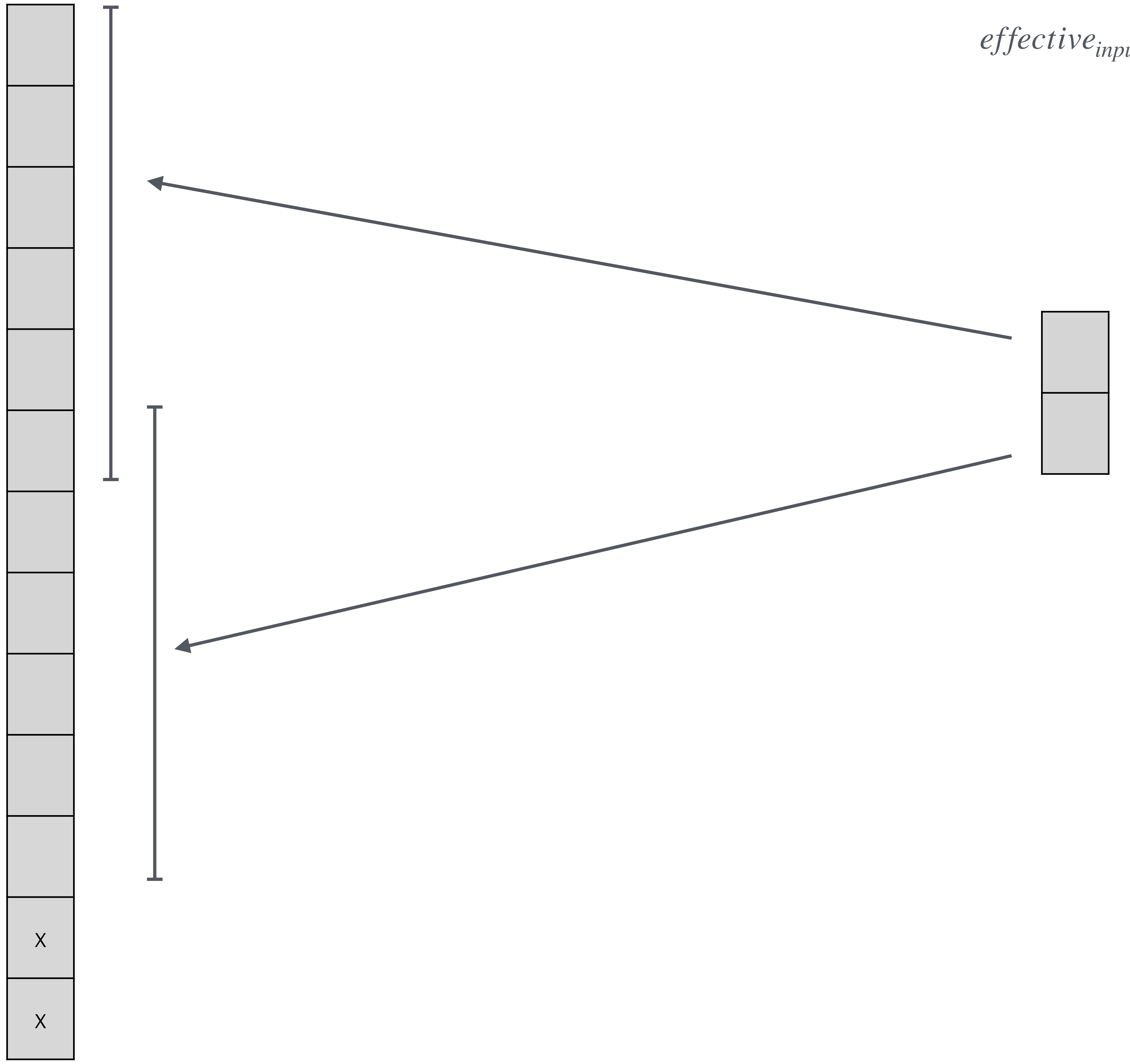
$$\begin{cases} stride = 1 & input_{effective_{length}} = length_{input} + width - 1 \\ stride! = 1 & output_{size} \times width - (output_{size} - 1) \end{cases}$$

$output_{size} = ceil(input_{length}/stride)$

Tensorflow Padding: Valid

$stride = 5$ $width = 6$ $Input_{length} = 13\ neurons$ $output_{size} = floor(\frac{13}{5}) = 2$

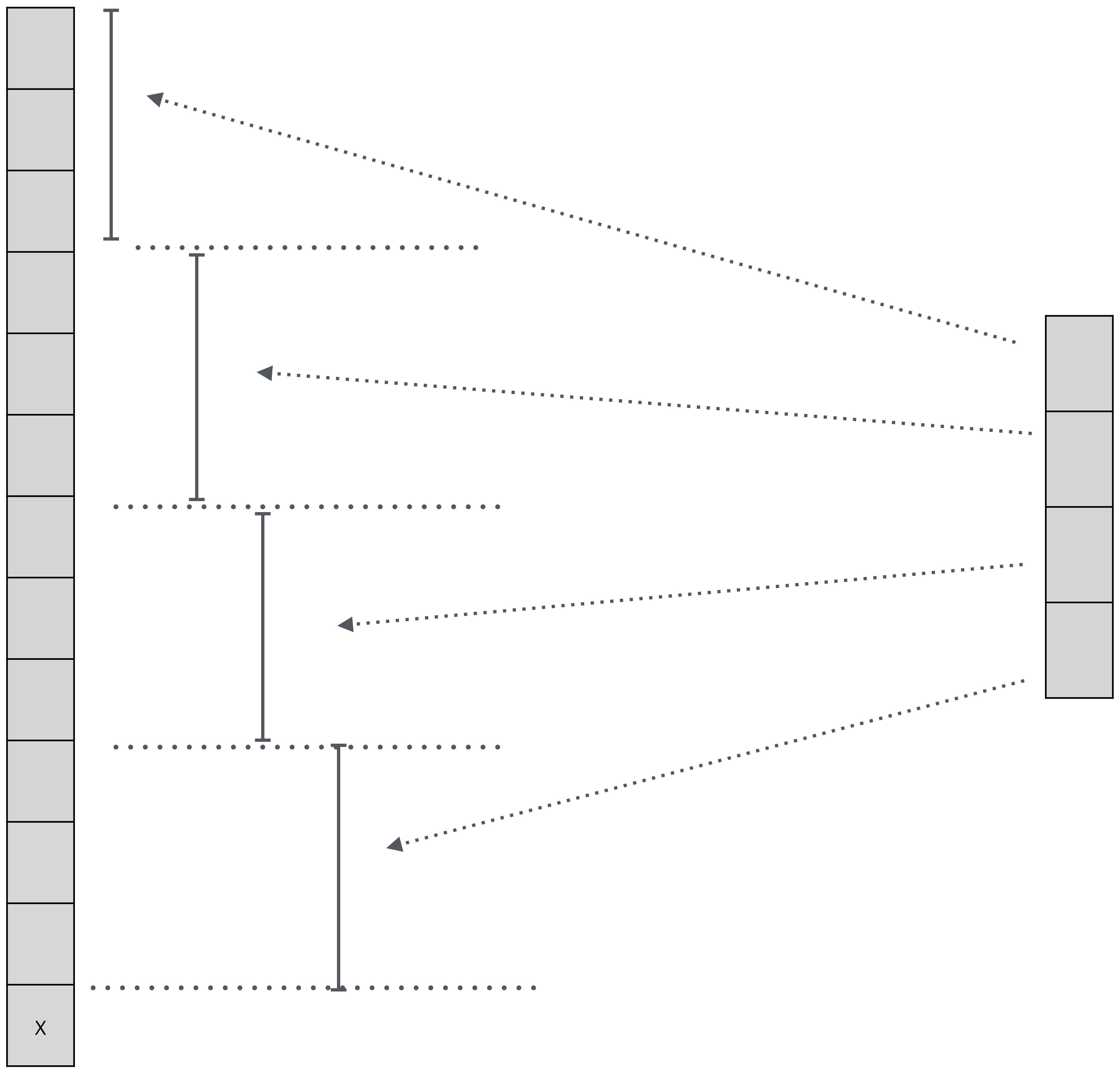
$$effective_{input-length} = width + (2-1) \times stride = 11$$



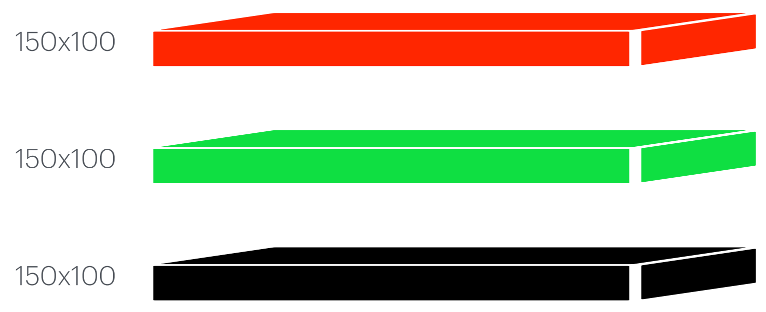
Tensorflow Padding: Valid

$stride = 3$ $\quad$ $width = 3$ $\quad$ $Input_{length} = 13\ neurons$ $\quad$ $output_{size} = floor(\frac{13}{3}) = 4$

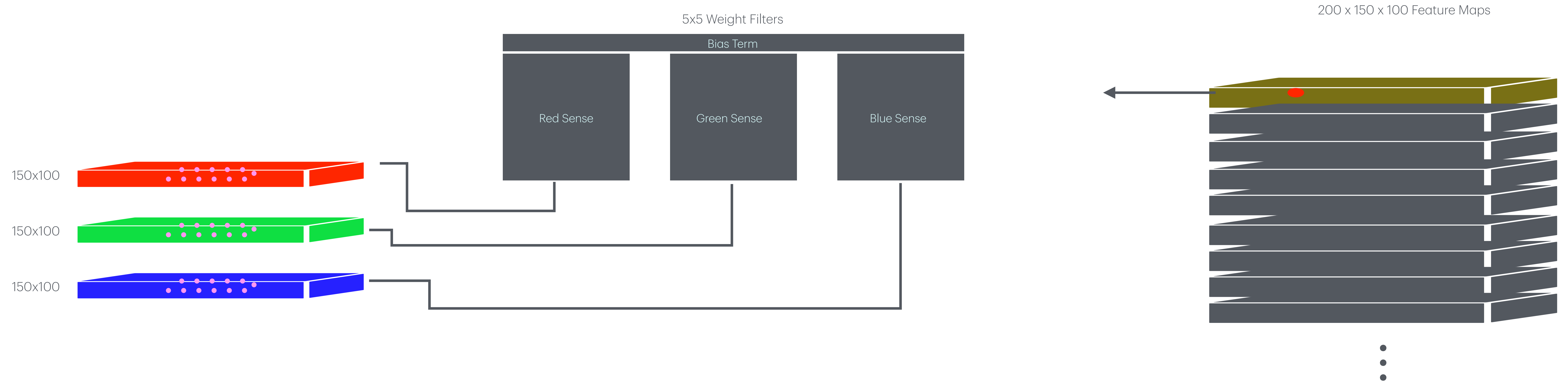
$$effective_{input-length} = width + (4-1) \times stride = 12$$



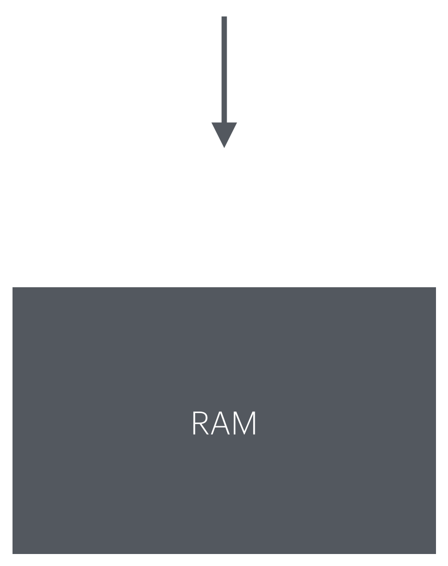
Memory



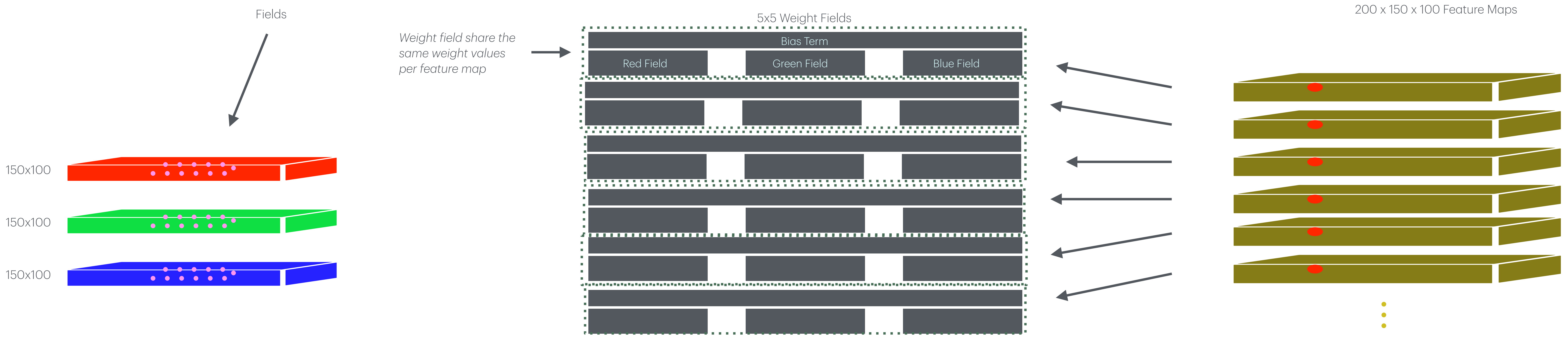
Memory



$$5 \times 5 \times 3 + 1 \ parameters$$



Memory ☣



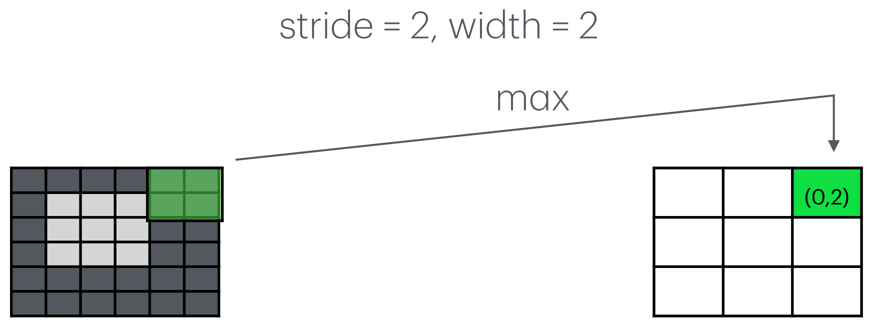
$(5 \times 5 \times 3 + 1) \times 200$ *parameters* in memory during each process clock cycle

↓

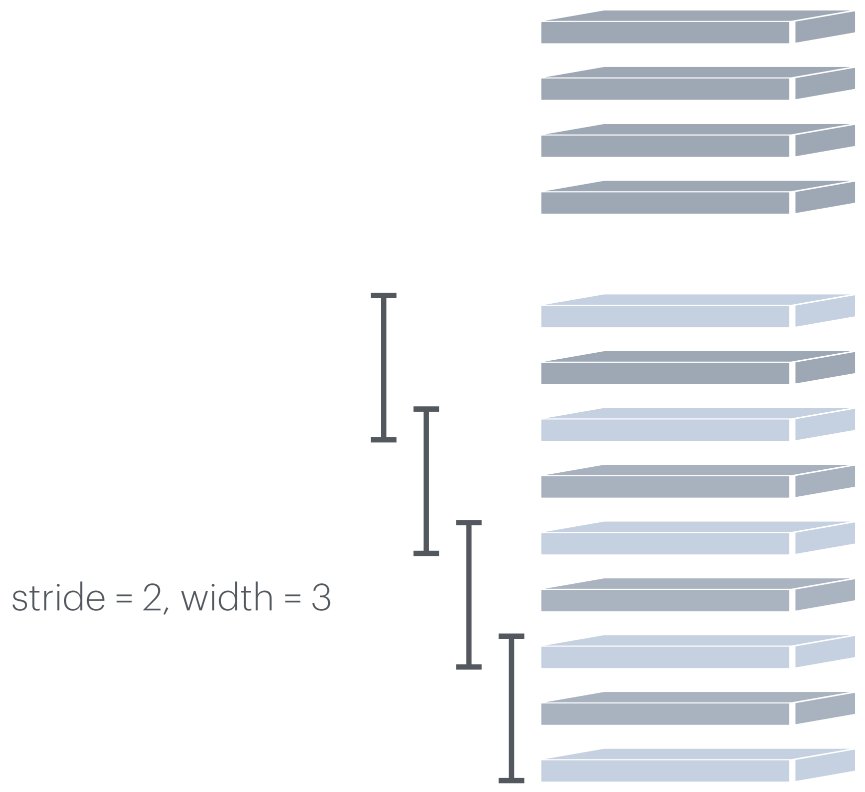
RAM

- 150x100 x 200 feature map neurons
- 5x5x3 field multiplications per neuron
- 5x5x3x100x150x200 = 225,000,000 multiplications
- 225,000,000 32 bit floats

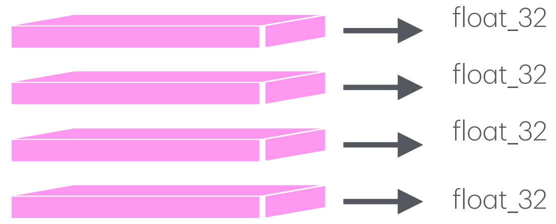
Max Pooling



Reduce computation load
and output size



Depthwise Pooling



Global Average Pooling

CNN

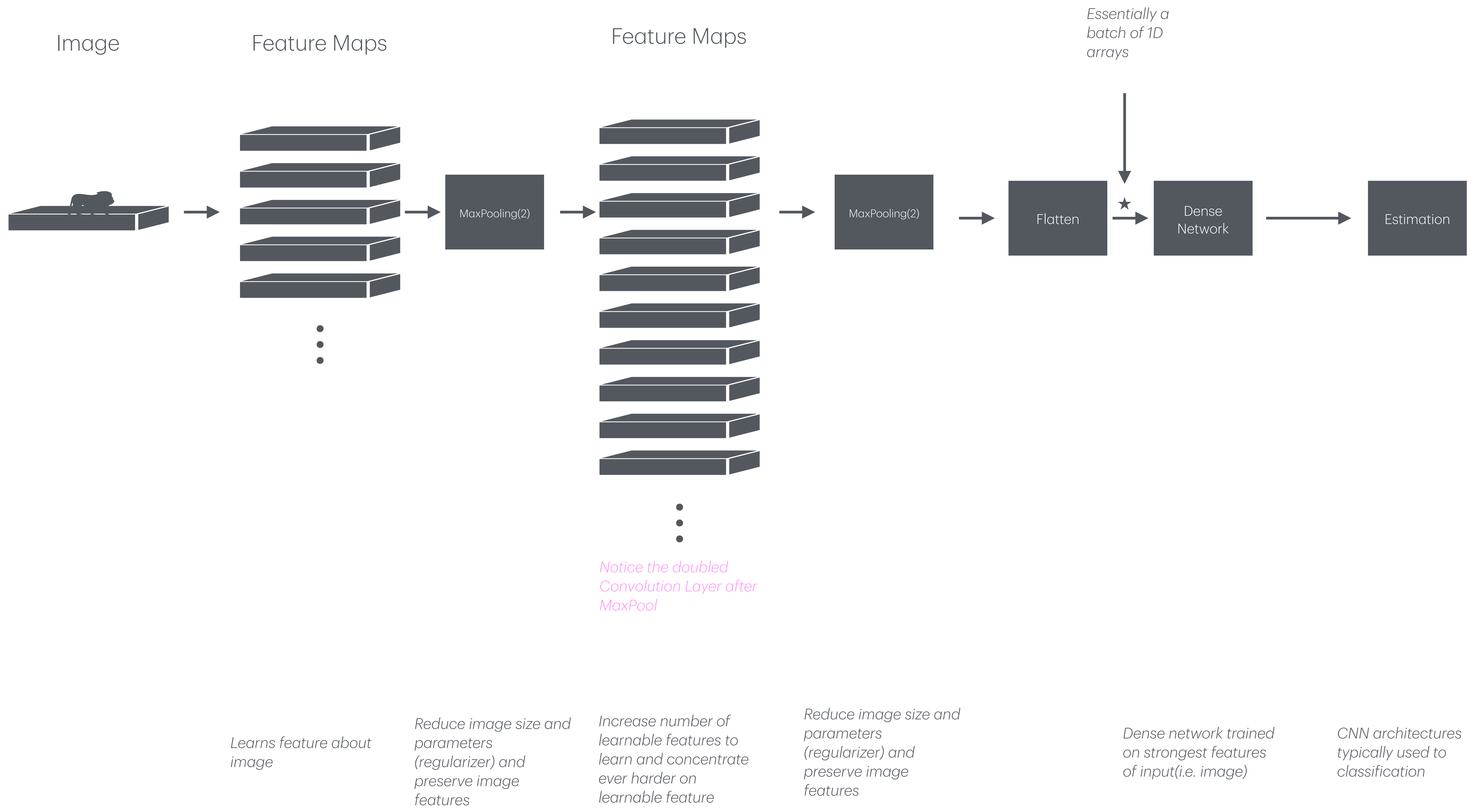
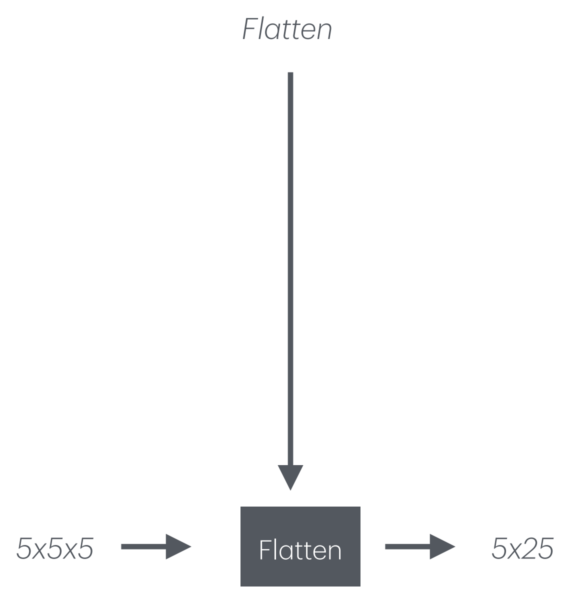


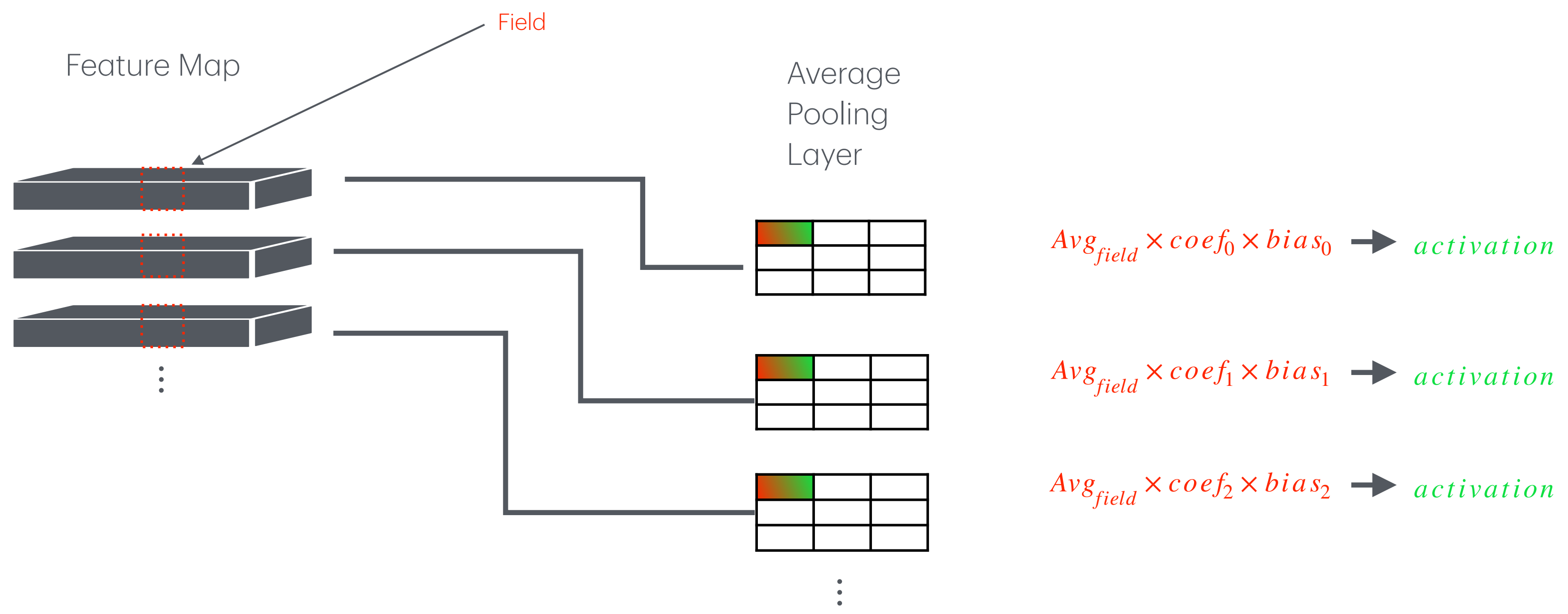
Image
Feature Maps
Feature Maps
Essentially a batch of 1D arrays
MaxPooling(2)
MaxPooling(2)
Flatten
★
Dense Network
Estimation
Notice the doubled Convolution Layer after MaxPool
Learns feature about image
Reduce image size and parameters (regularizer) and preserve image features
Increase number of learnable features to learn and concentrate ever harder on learnable feature
Reduce image size and parameters (regularizer) and preserve image features
Dense network trained on strongest features of input(i.e. image)
CNN architectures typically used to classification


Flatten
5x5x5
Flatten
5x25

LeNet-5: Special Feature

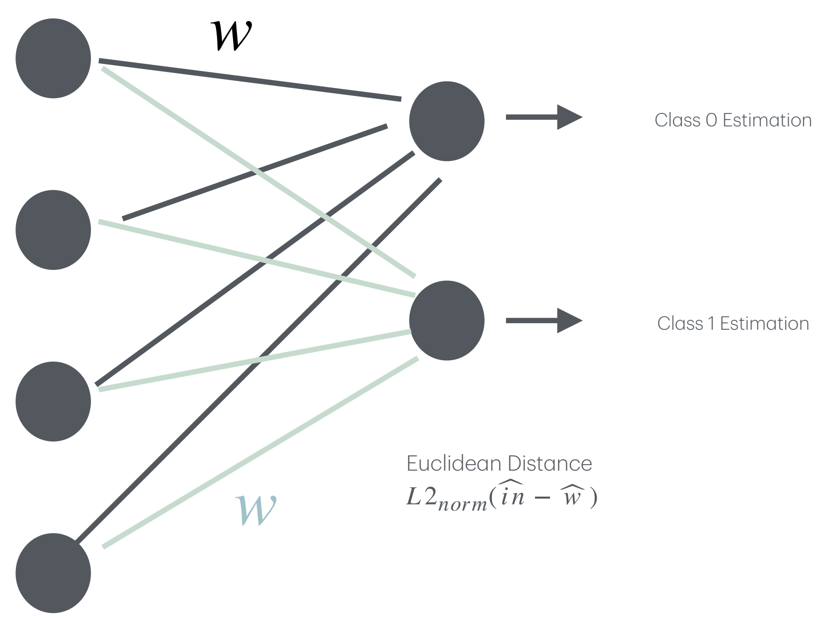
Popular widely known
CNN architecture

Average Pooling Layers



Output Layer

$$w = \begin{bmatrix} w_{11} \\ w_{21} \\ w_{31} \\ w_{41} \end{bmatrix}$$



Distance between input vector and weight vector measures how much image belongs to a digit class

$$in = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \qquad w = \begin{bmatrix} w_{12} \\ w_{22} \\ w_{32} \\ w_{42} \end{bmatrix}$$

AlexNet: Special Feature

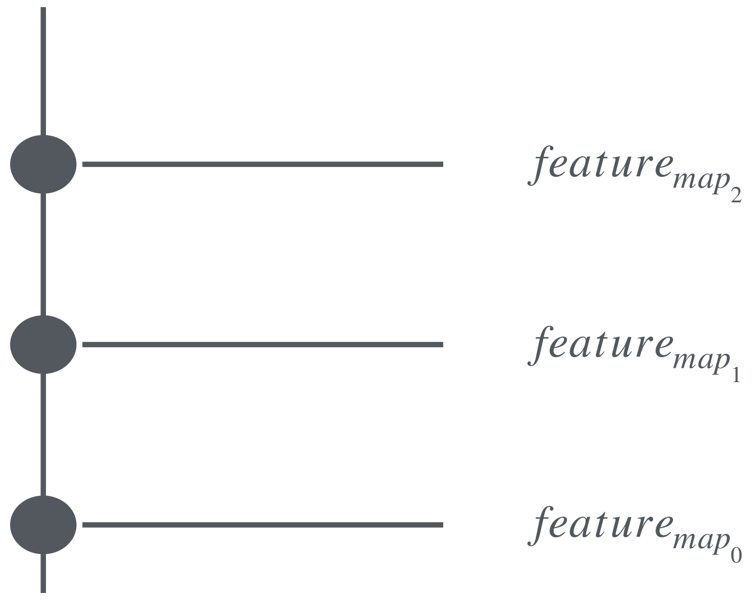
Similar to Le-Net, but includes stacked convolution layers

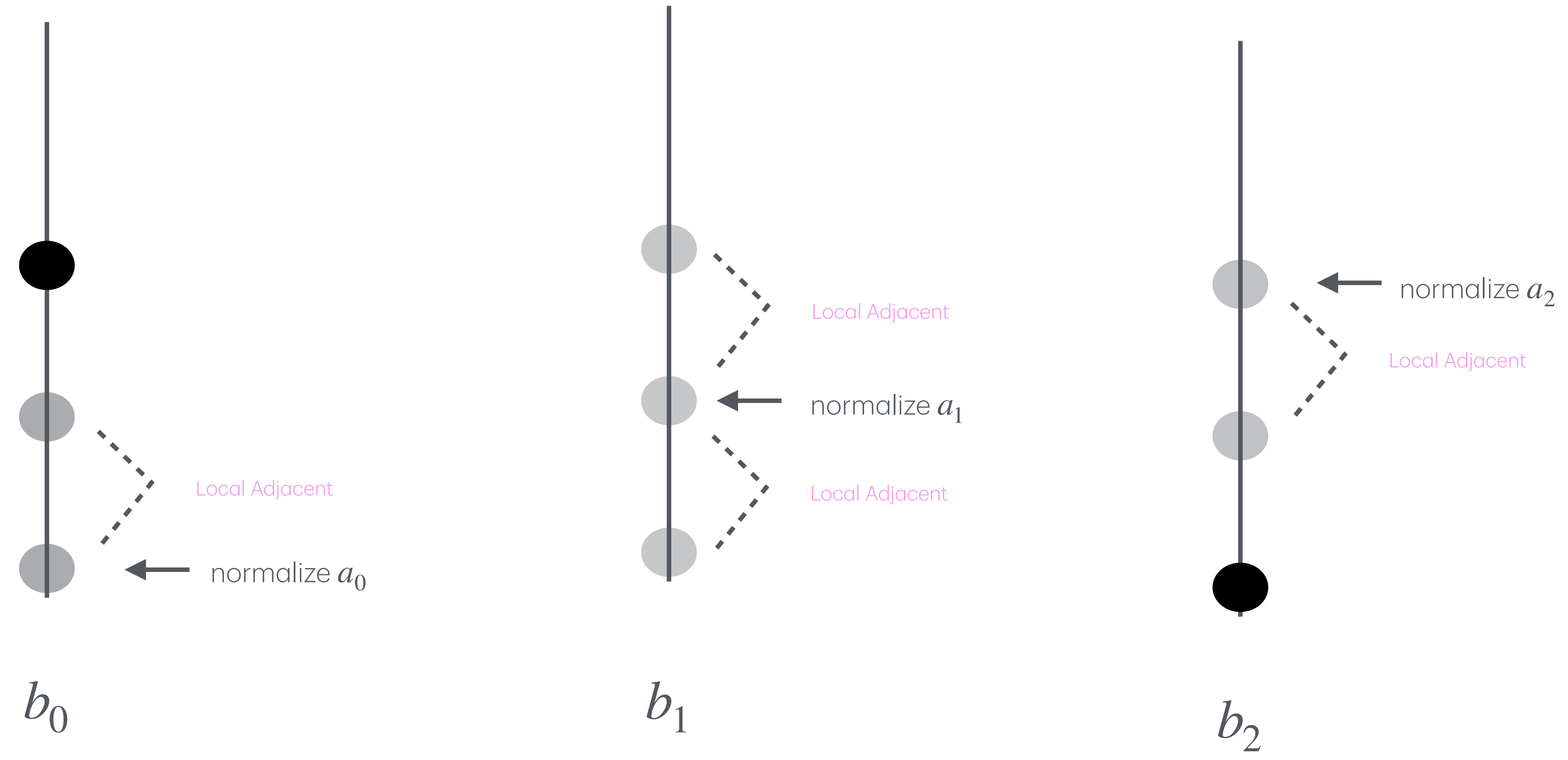
Normalization:

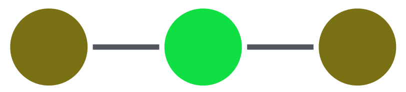
Local Response Normalization (LRN) used to bring out the strongest neurons which inhibit others along feature maps at position i,j

$$r = 2$$

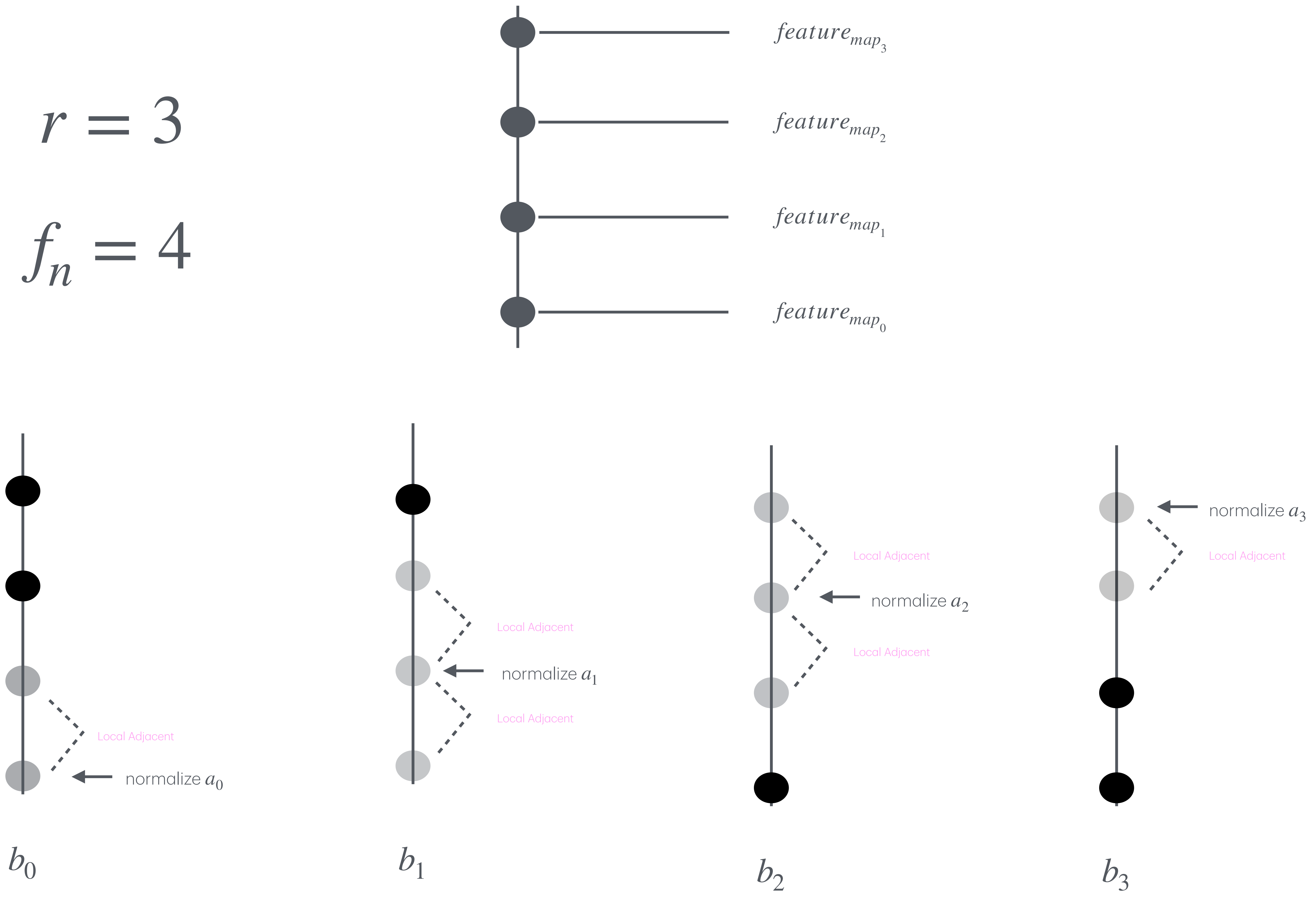
$$f_n = 3$$







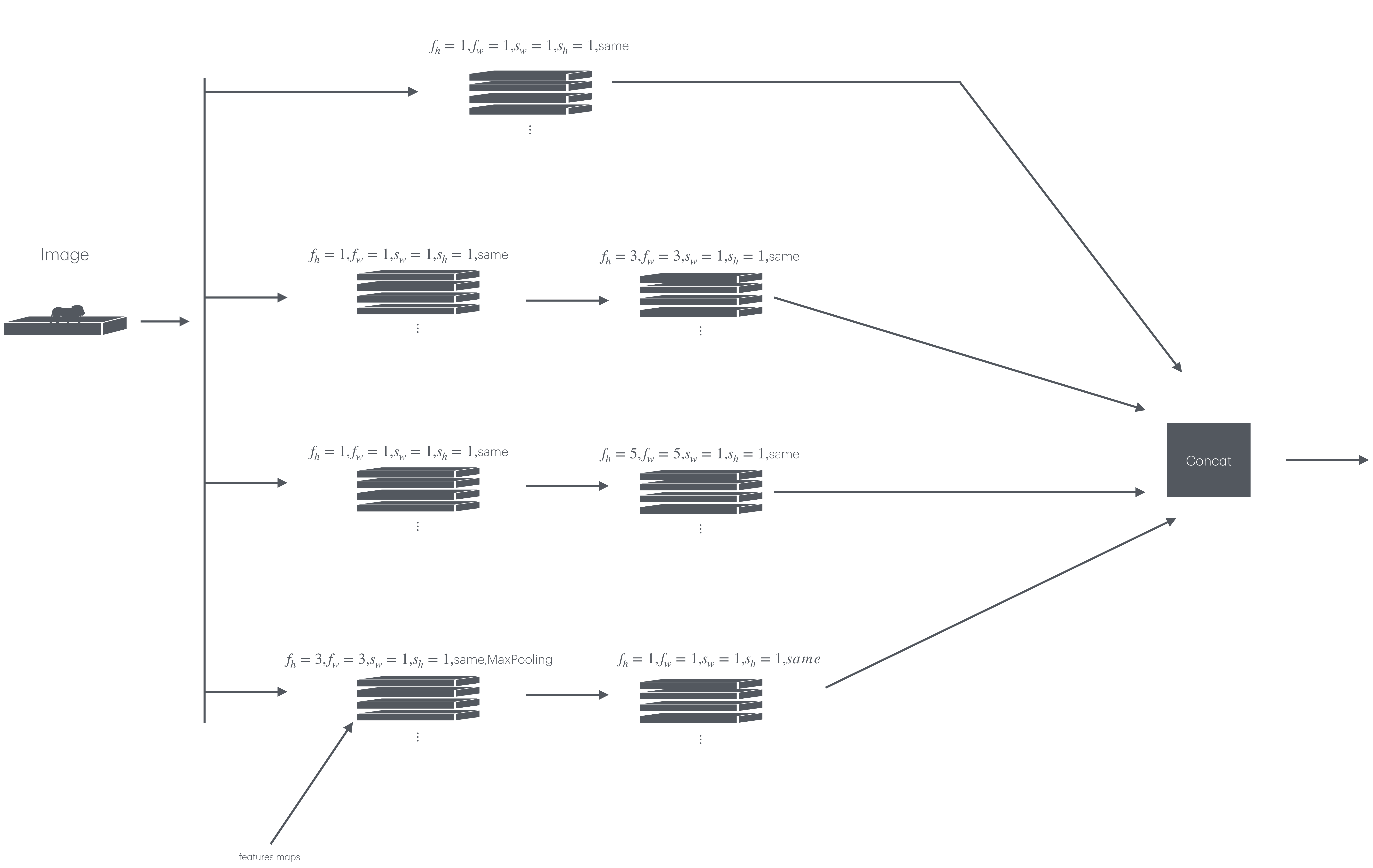
Strong neurons inhibit neighboring neurons. This is something observed in biological neurons. This normalization technique models this naturally occurring phenomenon to improve feature map diversity



$r = 3$
$f_n = 4$
$feature_{map_3}$
$feature_{map_2}$
$feature_{map_1}$
$feature_{map_0}$
Local Adjacent
normalize a_0
normalize a_1
normalize a_2
normalize a_3
b_0
b_1
b_2
b_3

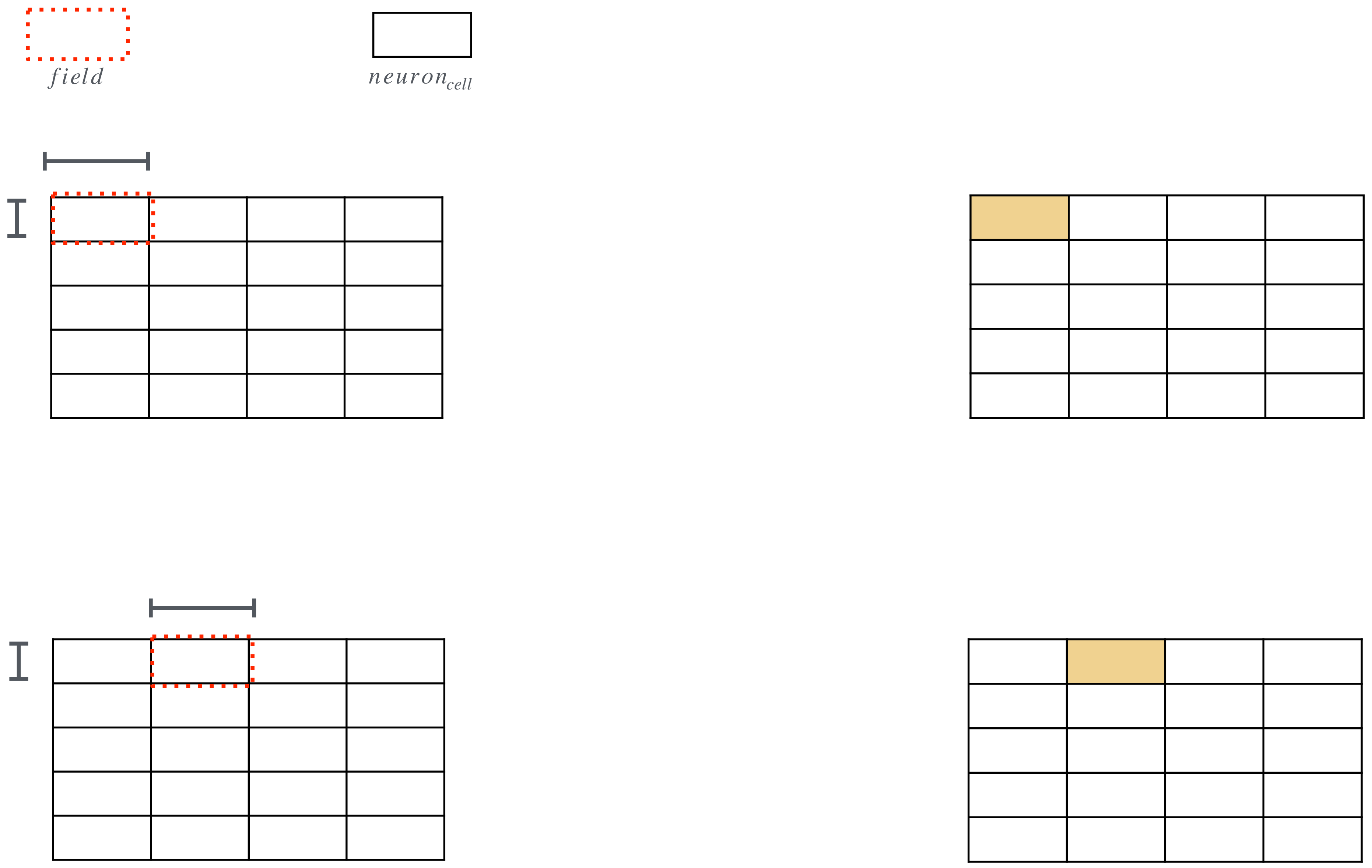
GoogleNet: Special Feature

Uses subnetworks called inception modules



GoogleNet: Special Feature

$f_h = 1, f_w = 1, s_w = 1, s_h = 1,$same

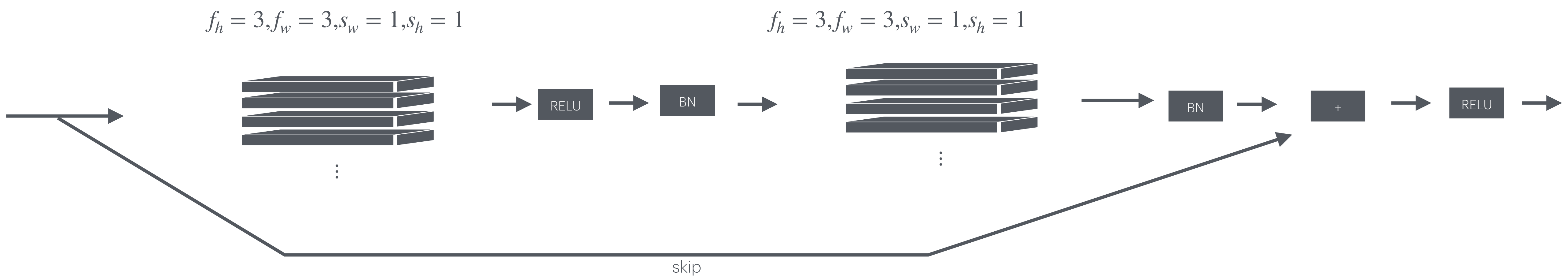


Note:

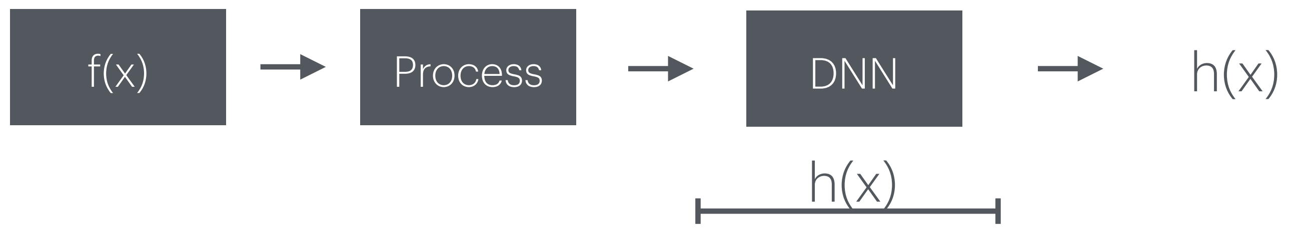
Feature map neurons are weighted sums with trainable receptive weight field and neuron cell(s) from input , that are then fed to RELU activation function

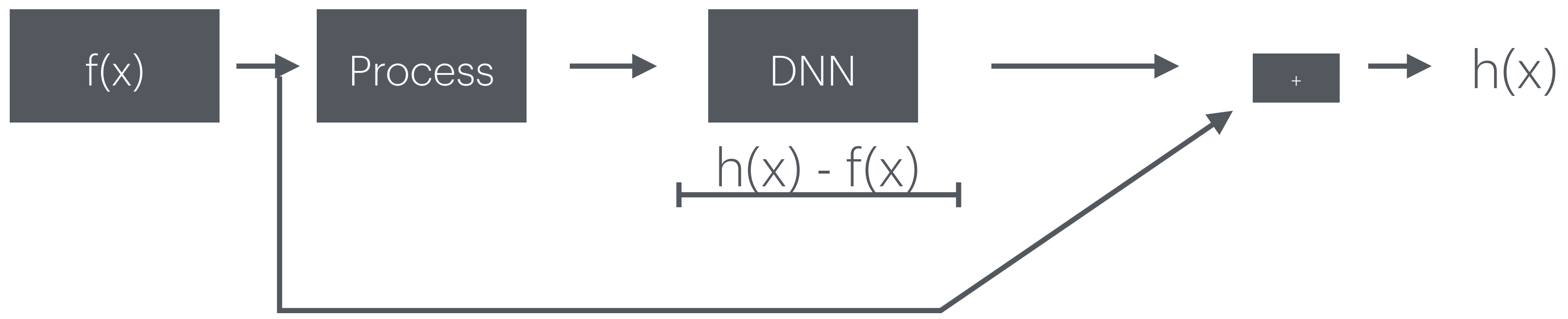
ResNet: Special Feature

Residual Units (Skip Connections



Note:
Skip connections supports the units ability to model h(x) by forcing it to learn h(x) - x. This aids in the models ability to converge to its target model h(x)





Note:
Skip forces DNN to learn target model - input

If target model is close to the input of identity function then the DNN will converge faster during training

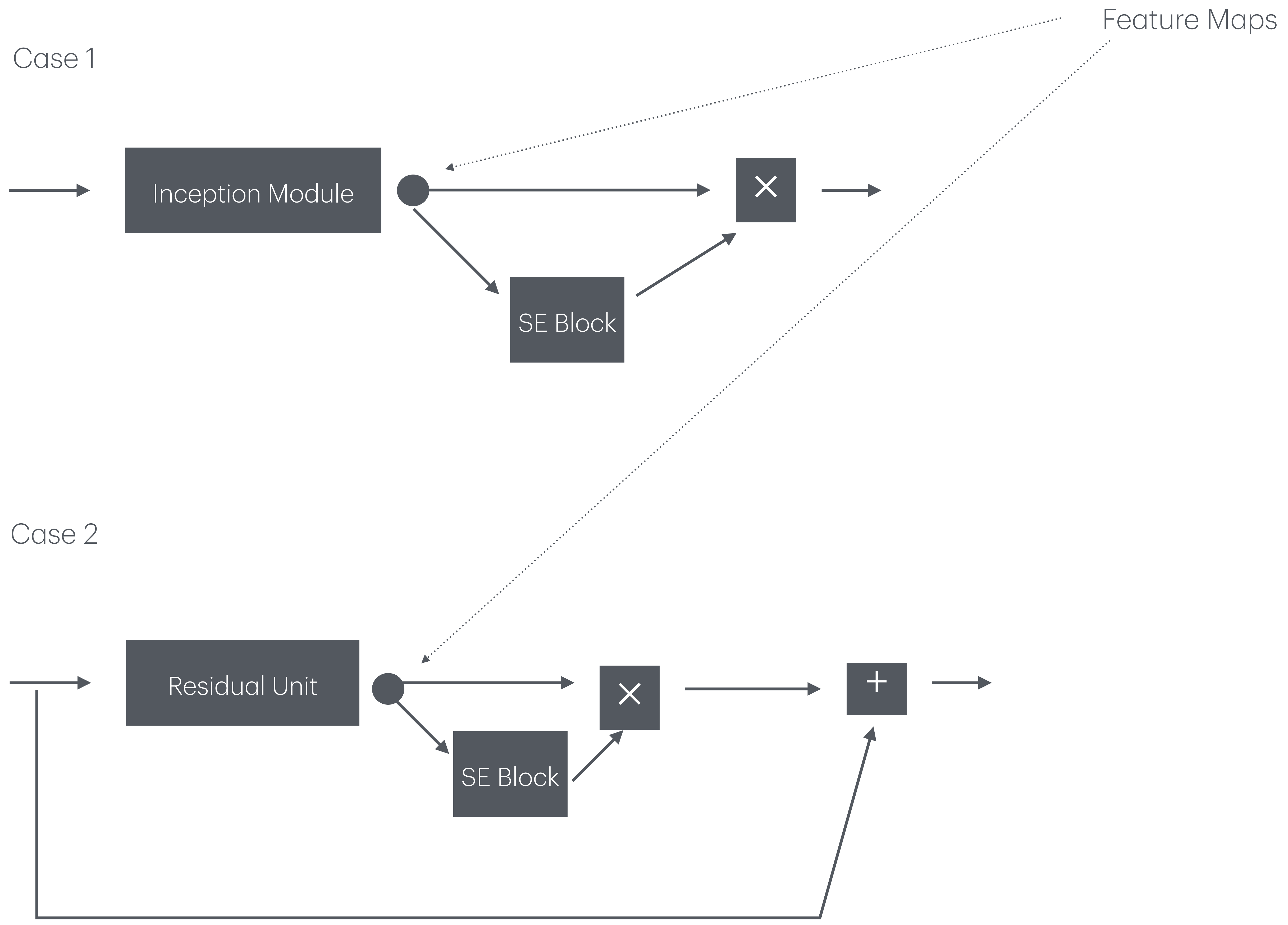
- f(x) = clean image function
- Process - noise system
- h(x) = recover clean image function
- h(x) - f(x) - NN creates network which essentially is the the image function reconstruction error. Network models distance between clean function and recovered function

Xception: Special Feature

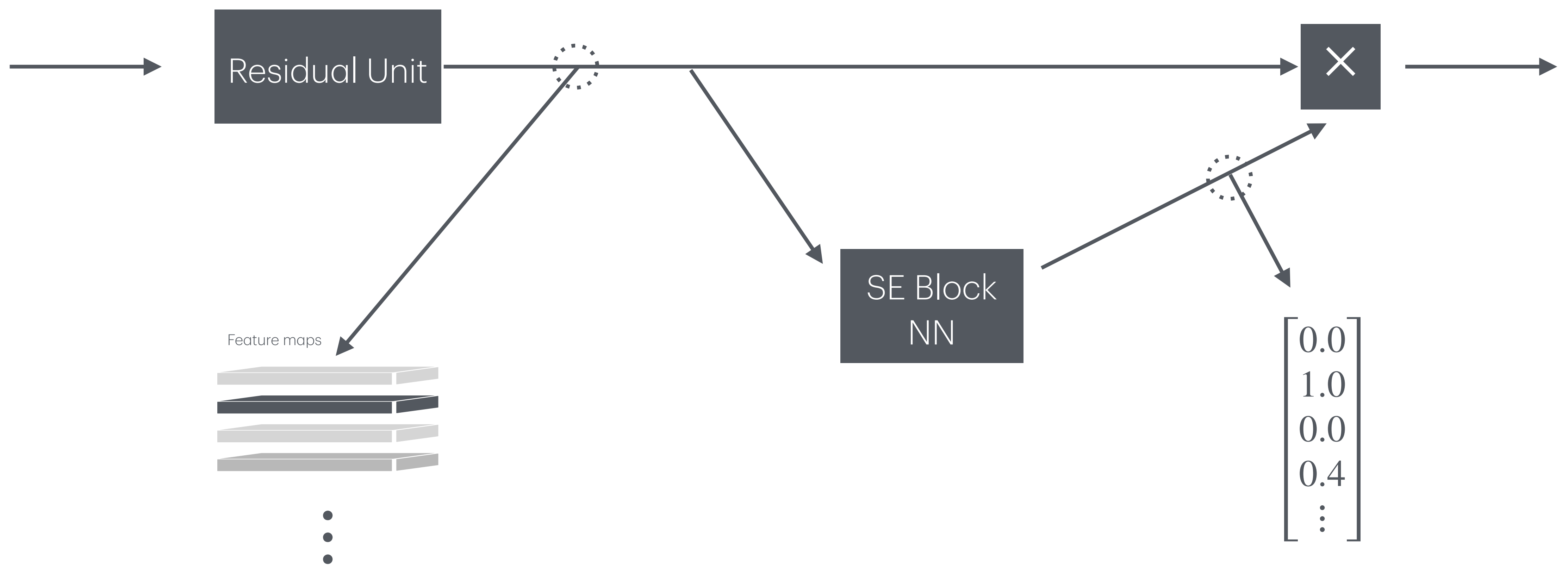
Replaces inception (GoogleNet) with depthwise separable convolutional layer

SeNet: Special Feature

Squeeze and Excitation Network SENet



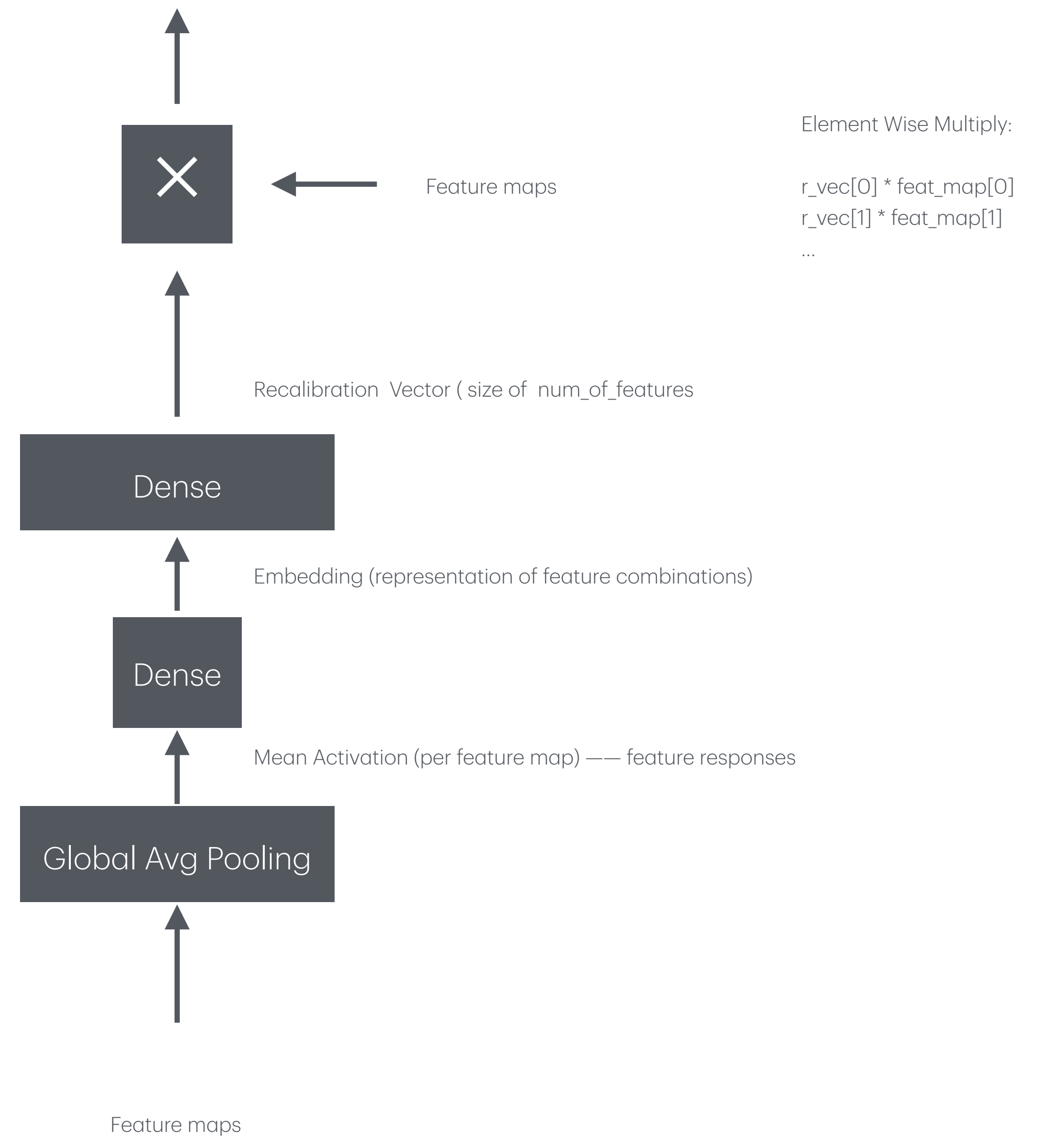
SeNet: Special Feature



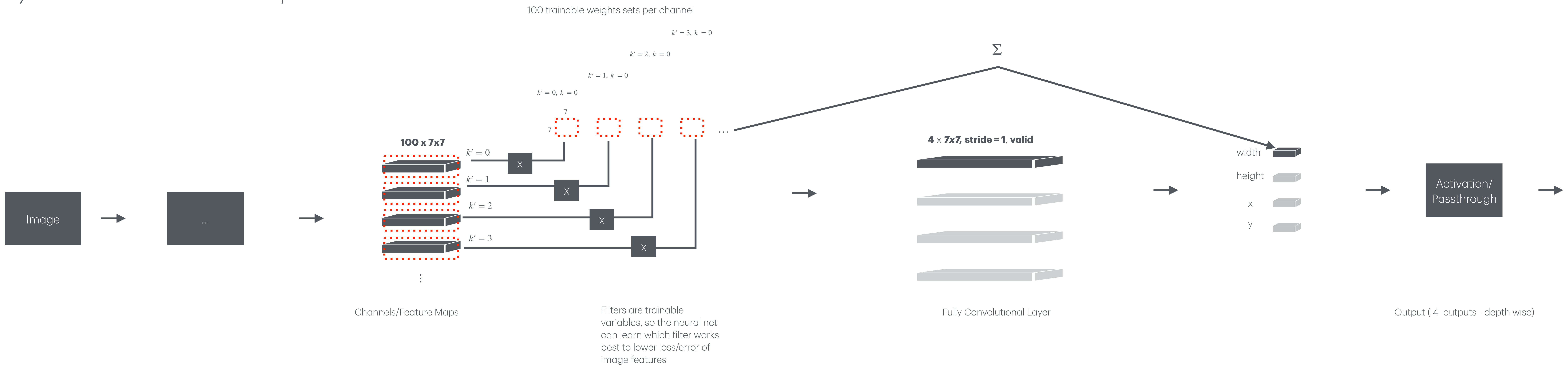
During training SE Blocks learn which features are most active together

During inference the NN will use the learned feature relationships and boost activation of certain features and reduce irrelevant ones

SE Block



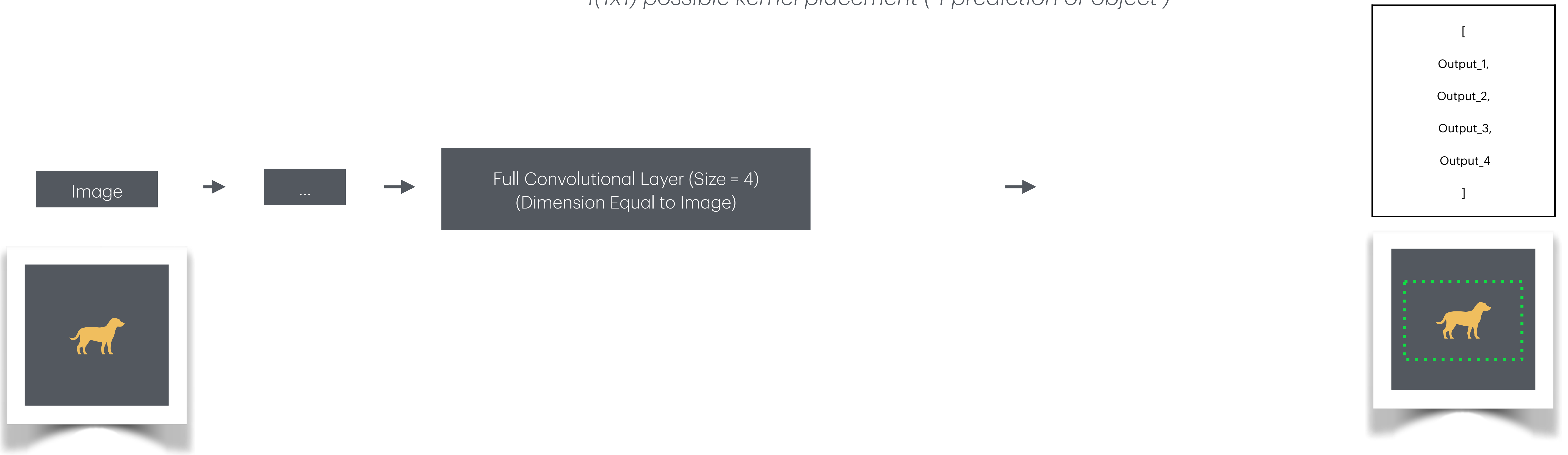
Fully Convolutional Networks: Special Feature



Imagine turning input image into a 7x7 grid (image)

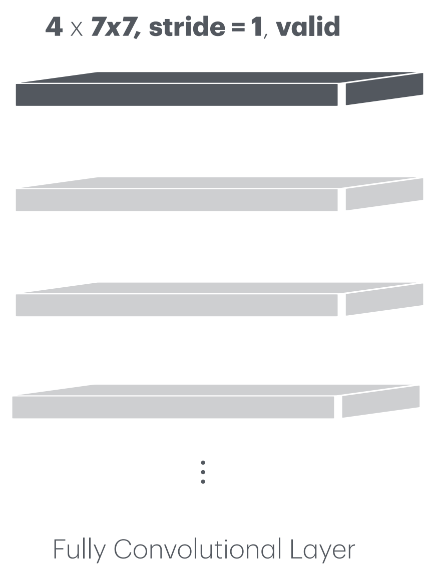
Then sliding unique 7x7 kernel(e.g. dog contour kernel) across 7x7 image

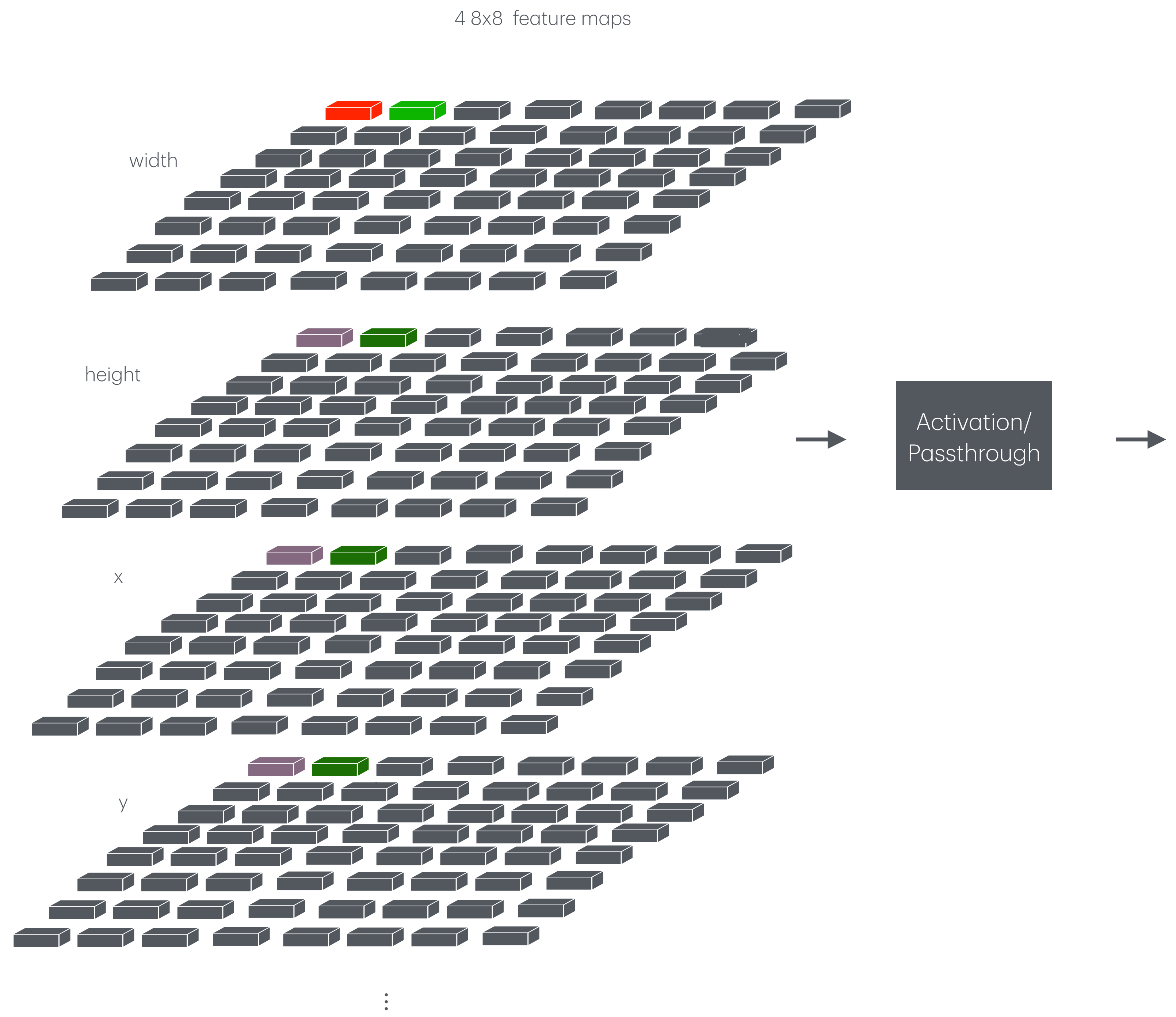
1(1x1) possible kernel placement (1 prediction of object)



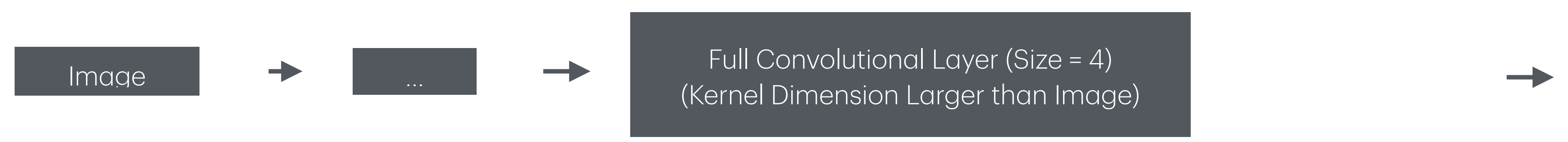
Output neuron activity pertains to bounding box estimation image

Fully Convolutional Networks: Special Feature

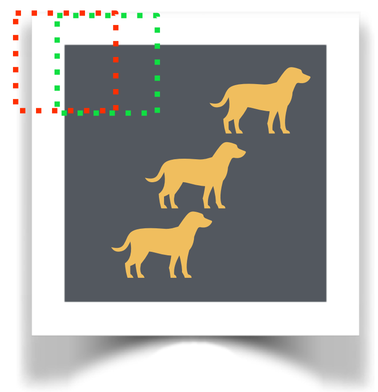




Imagine turning input image into a 14x14 grid (image)
Then sliding unique 7x7 kernel(e.g. dog contour kernel) across 14x14 image
64(8x8) possible kernel placements (64 predictions)

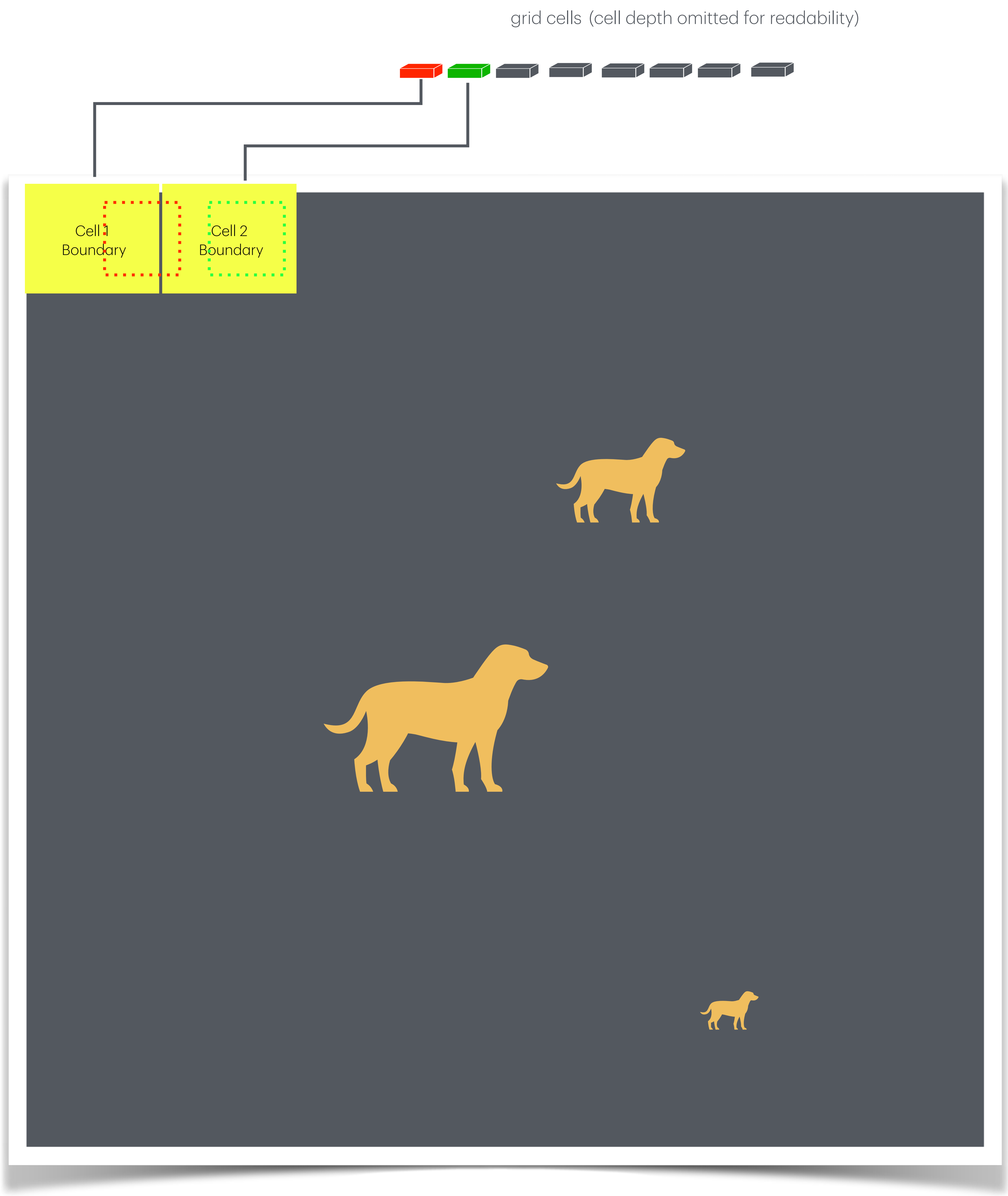


[Out1, Out2, Out3, Out4]							

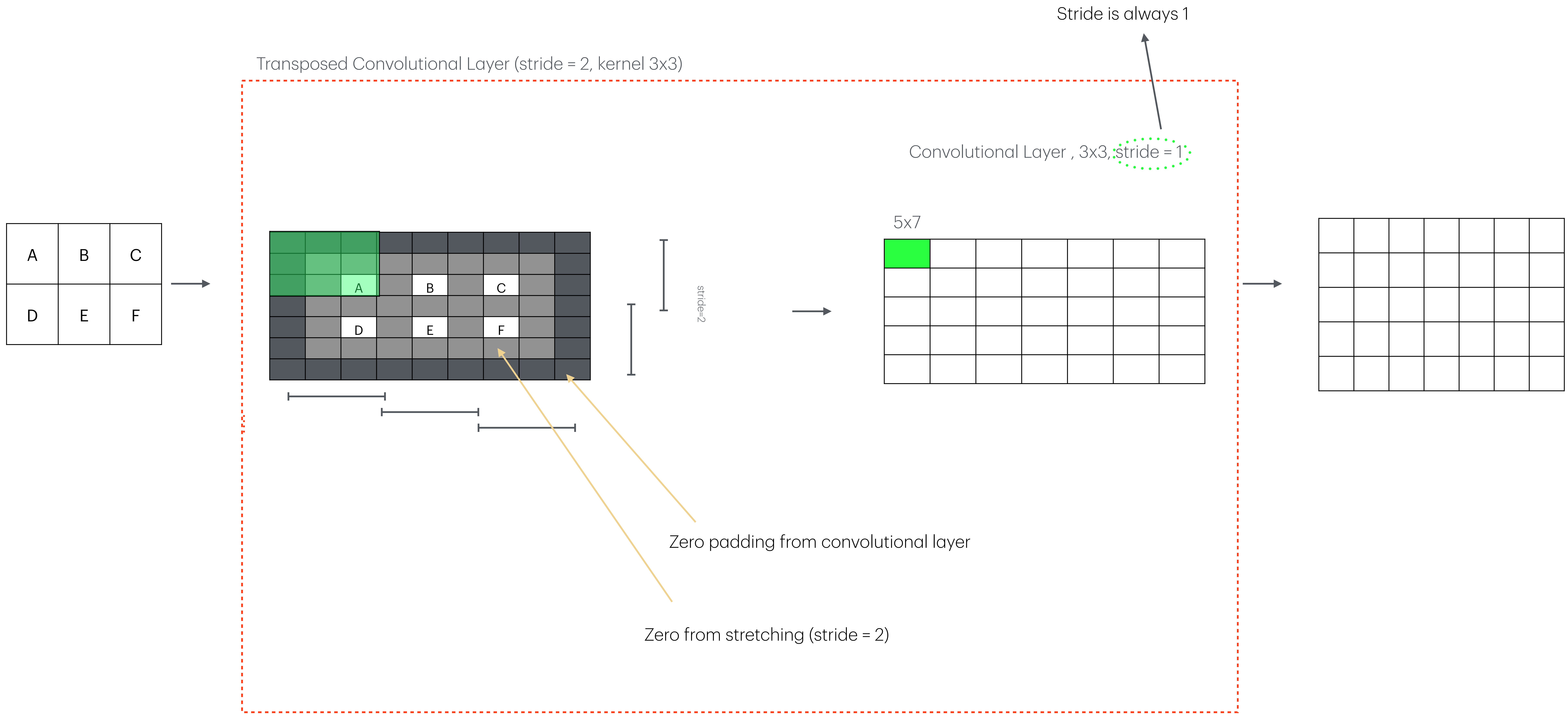


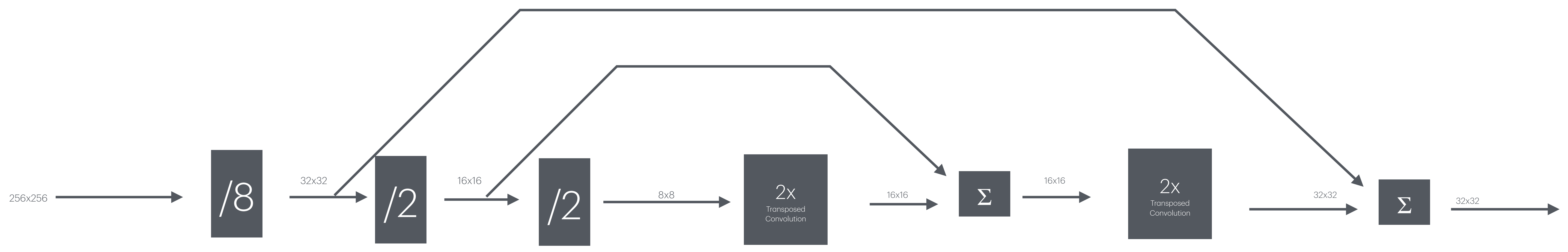
- Output neuron activity per grid cell .
- Bounding box created by grid cell estimations
- An additional output is required, the objectiveness score; to tell if object is inside bounding box

Algorithms like YOLO find boundary box centers anywhere within the cell boundary. It uses offsets relative to grid cell.



Transposed Convolution: Upsampling

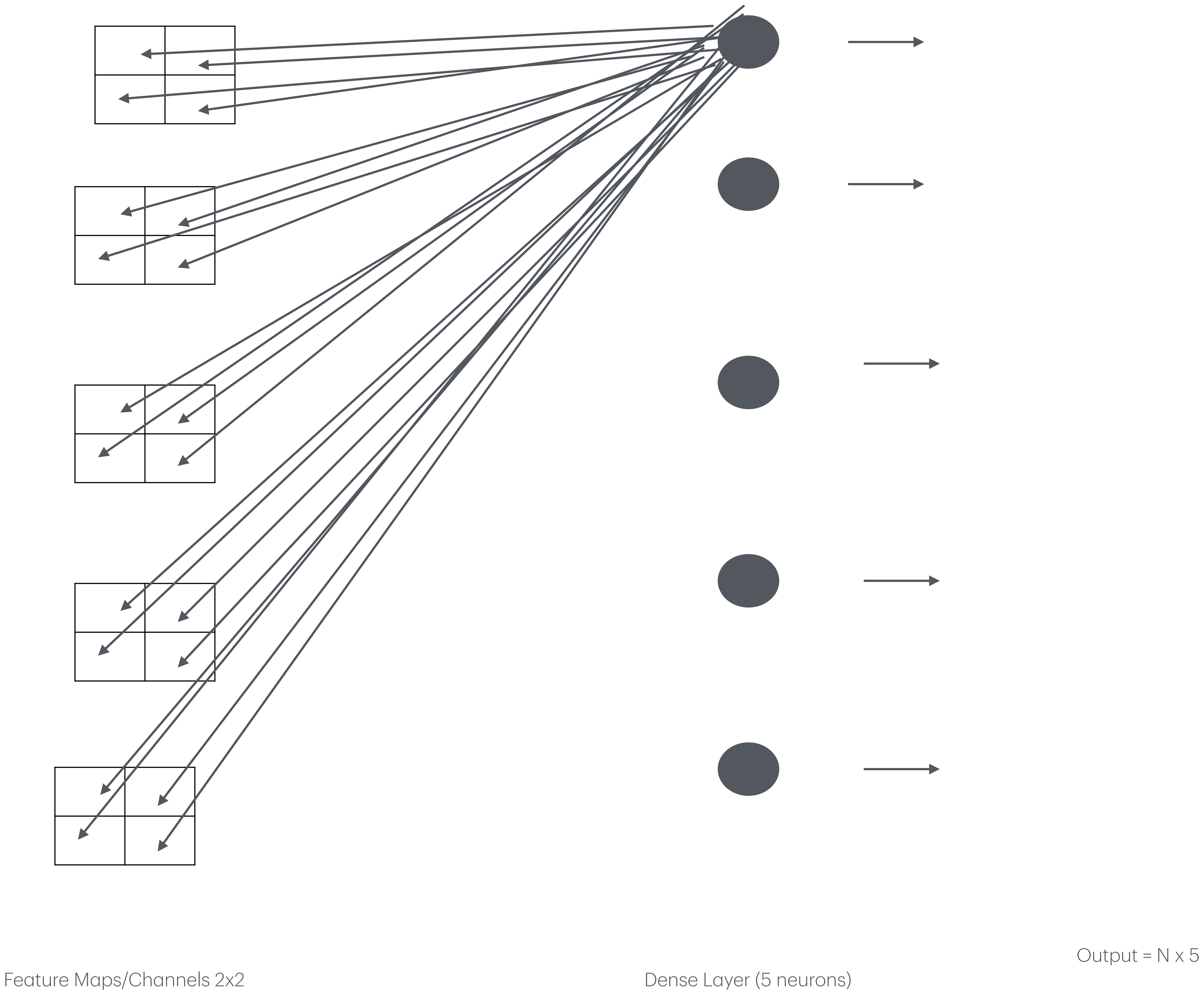




Images through CNN lose spatial resolution (caused by strides).

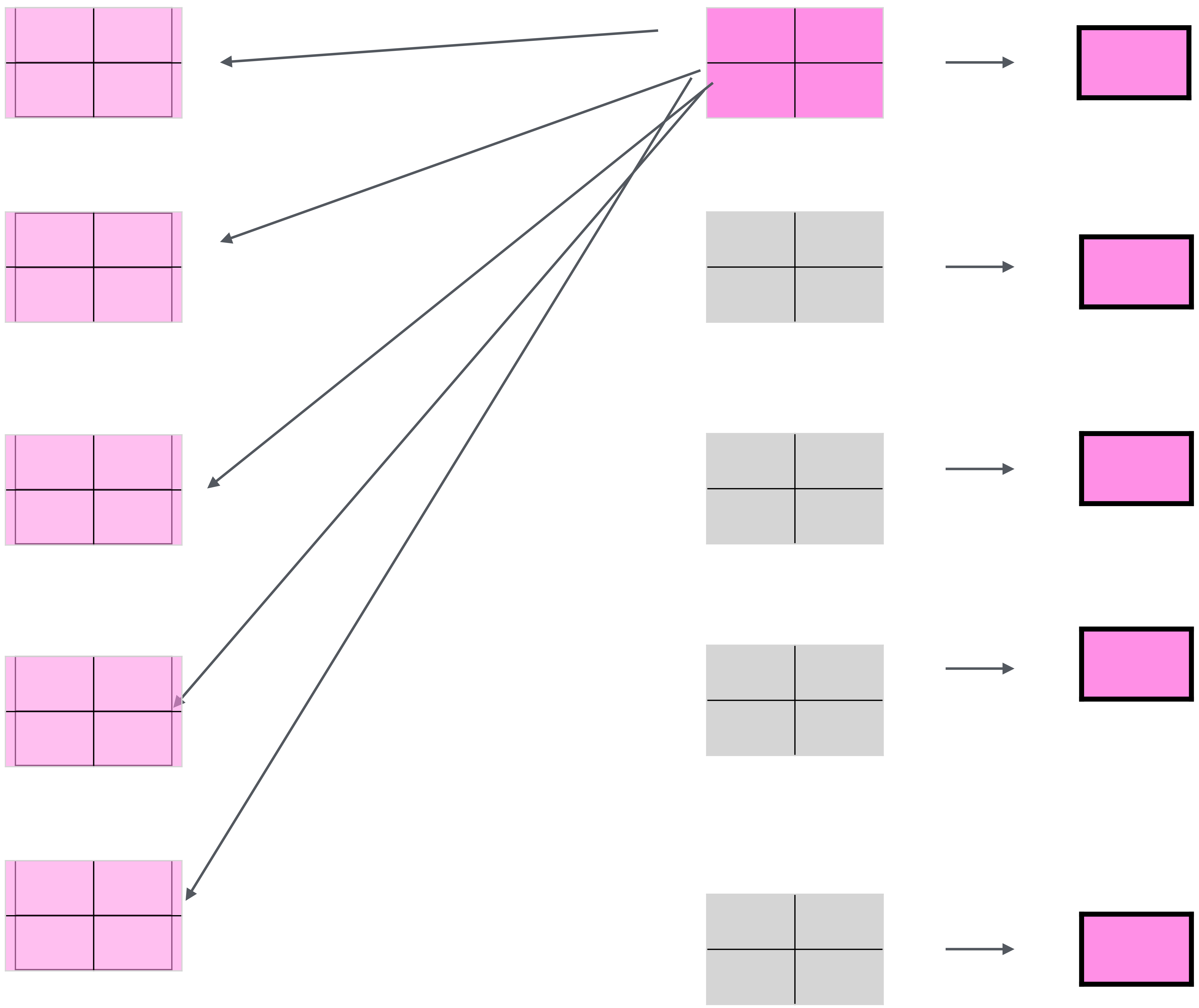
Spatial resolution can be recovered using skip layers with transposed convolution.

Why FCN:



The renaming neurons also compute weighted sum of each input neuron

Why FCN:



Feature Maps/Channels 2x2

Convolutional Layer (5 Filters)

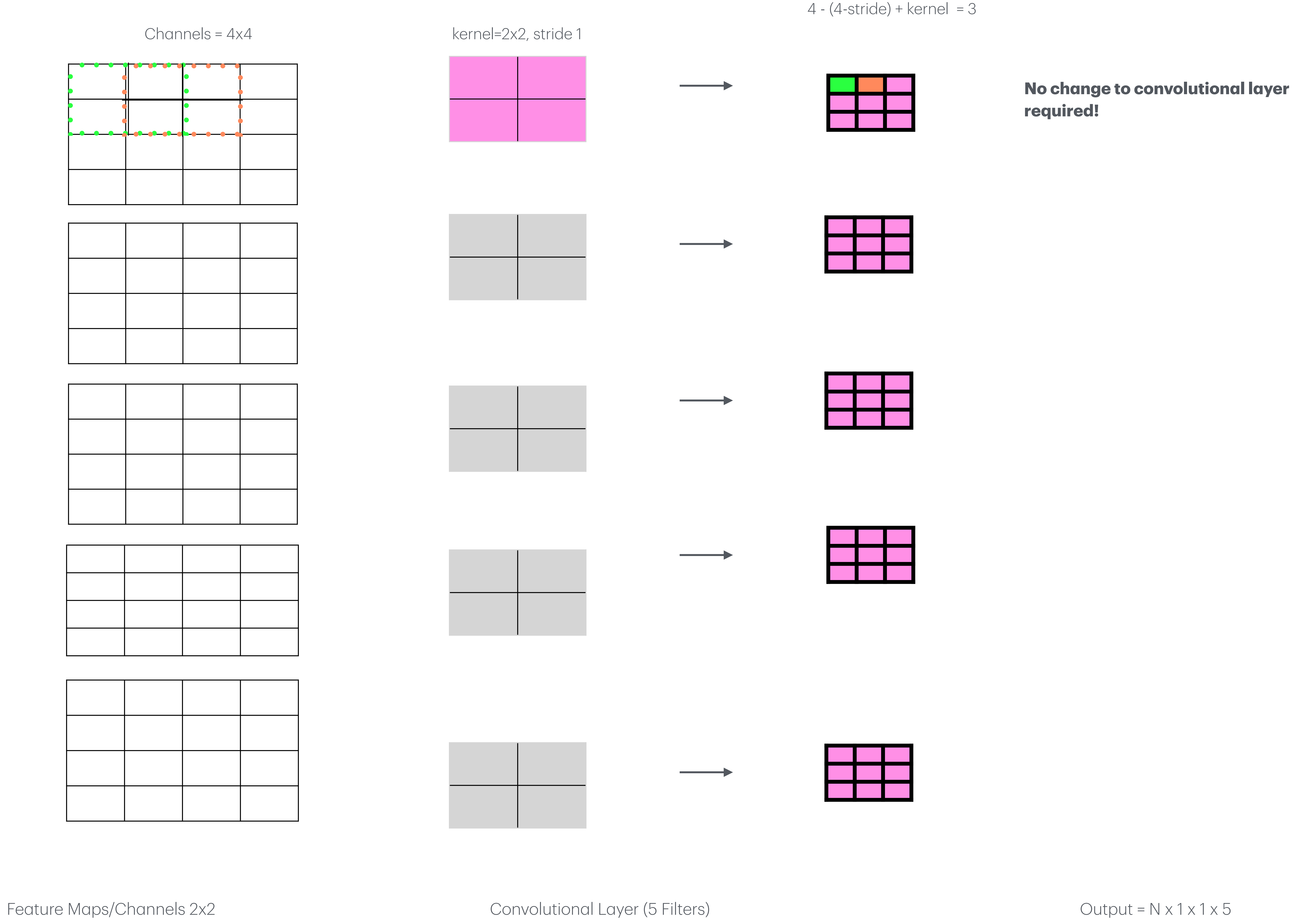
Output = N x 1 x 1 x 5

Weighted sum is equivalent to dense model in previous slide

The only difference is the output shape. Dense Layer returns batch of vectors, while convolutional layer returns a batch of depth-wise matrices (size = 1x1 , depth = 5, in this example)

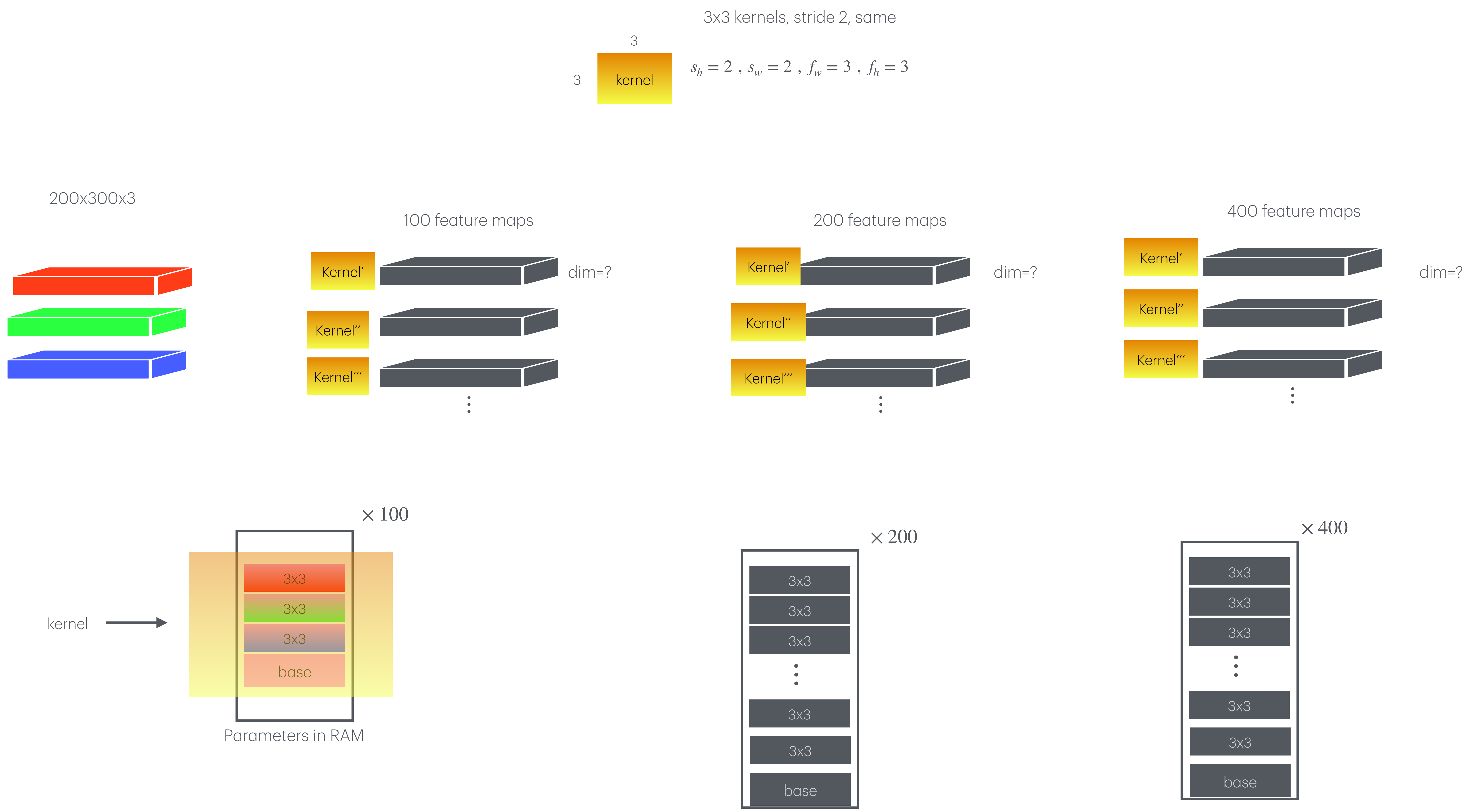
Convolutioanl layer will process any input, while dense layer expect a size.

Why FCN:



4 - (4-stride) + kernel = 3
Channels = 4x4
kernel=2x2, stride 1
No change to convolutional layer required!
Feature Maps/Channels 2x2
Convolutional Layer (5 Filters)
Output = N x 1 x 1 x 5

Flow Example



((3x3) * 3 +1)*100 = 2,800 parameters

((3x3) * 100 +1) * 200 =180,200 parameters

((3x3) * 200 +1) * 400 = 720,400 parameters

Remember kernel 'slides' and its parameters are trained

Remember kernel is shared across input channel receptive field (depth, unique per feature map

903,400 total parameters persists throughout runtime

Flow Example

Once a layer's neurons are computed, previous layer can be removed from memory

Kernel parameters bytes -= 903,400 * 4 = 3,613,600 bytes

Custom CNN

Args = fw, fh, sw, sh,

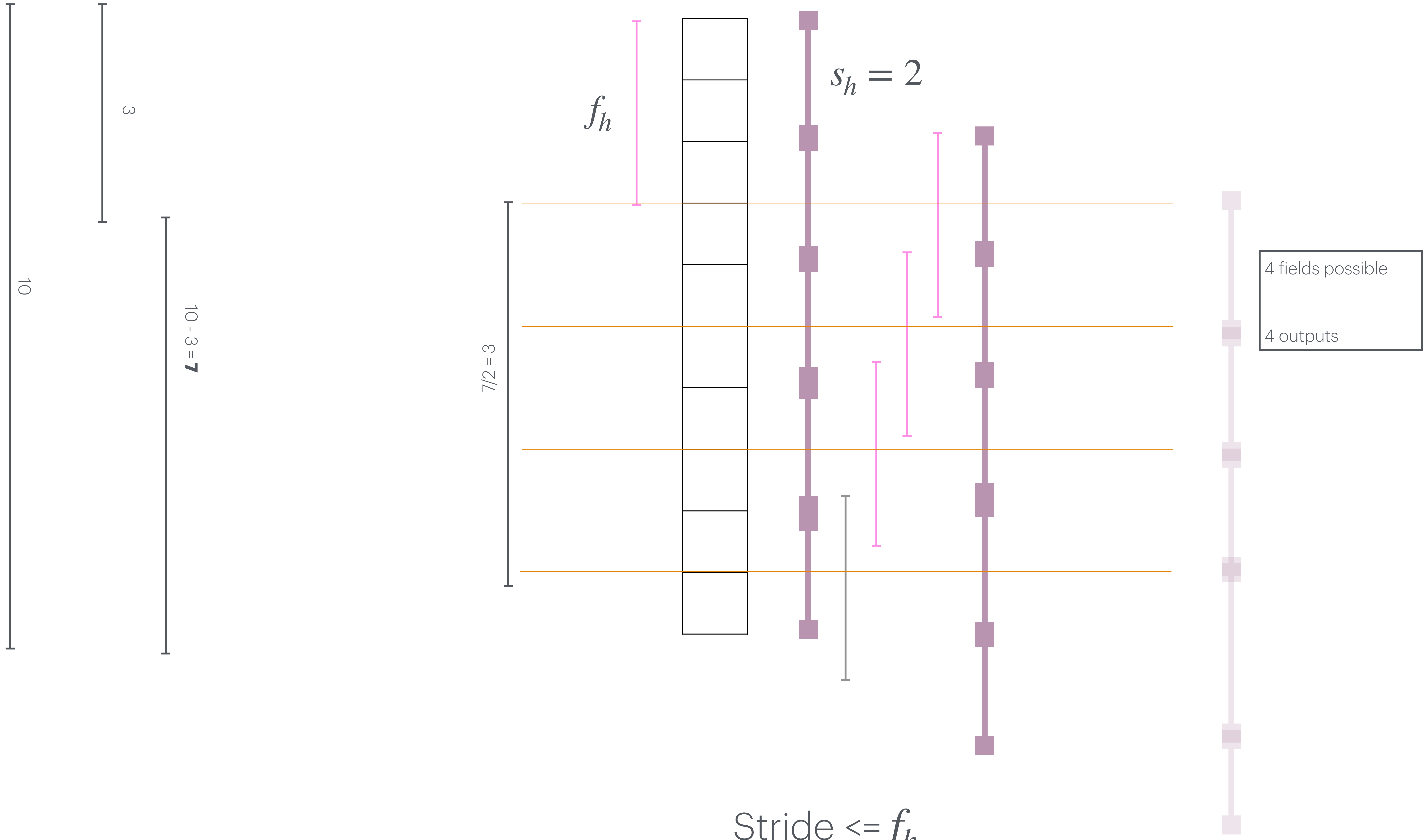
Batch x 200 x300

Image → compute output width/ height → compute kernels →

Viewer

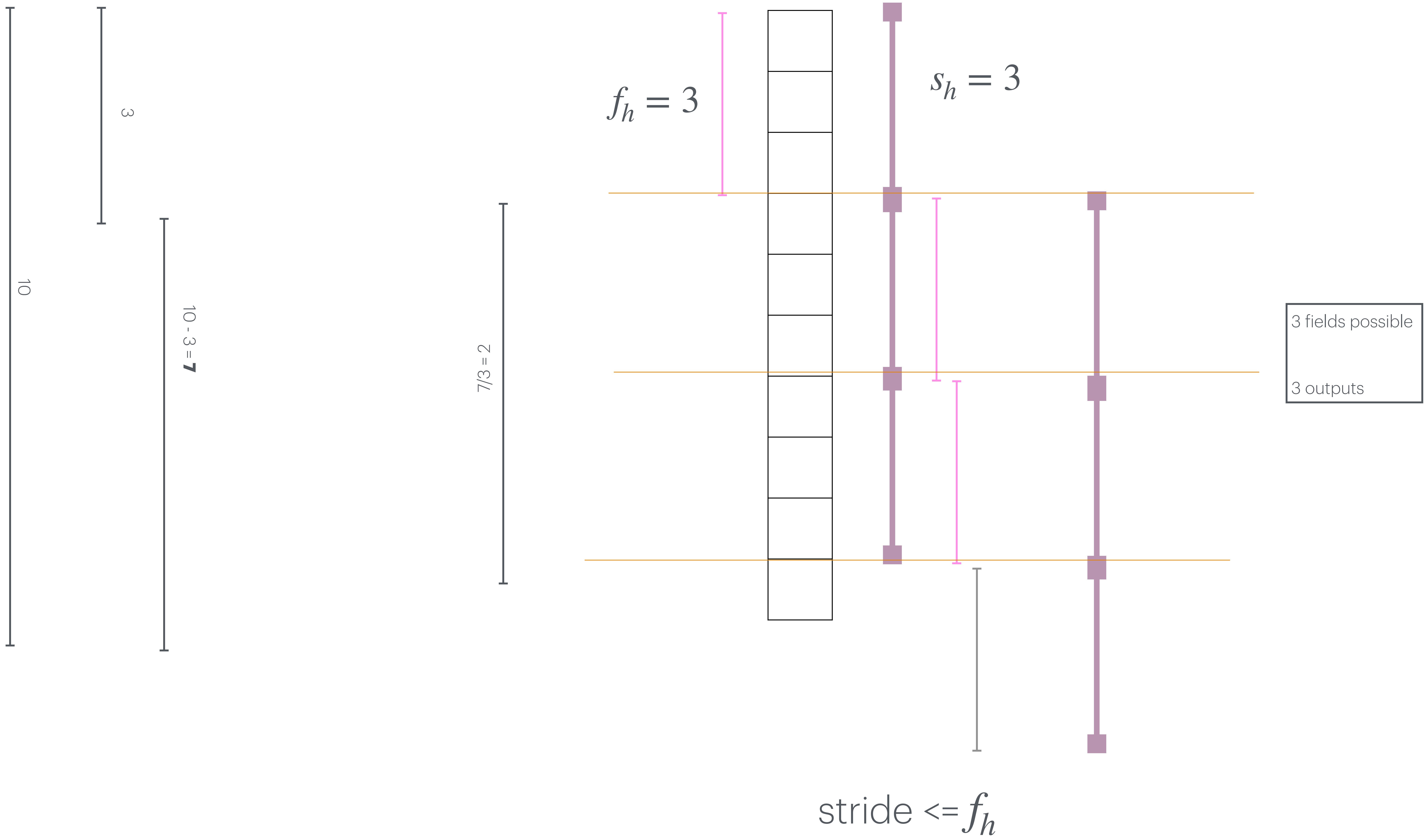
View input and active convolution layer

Output Shape : Case 1



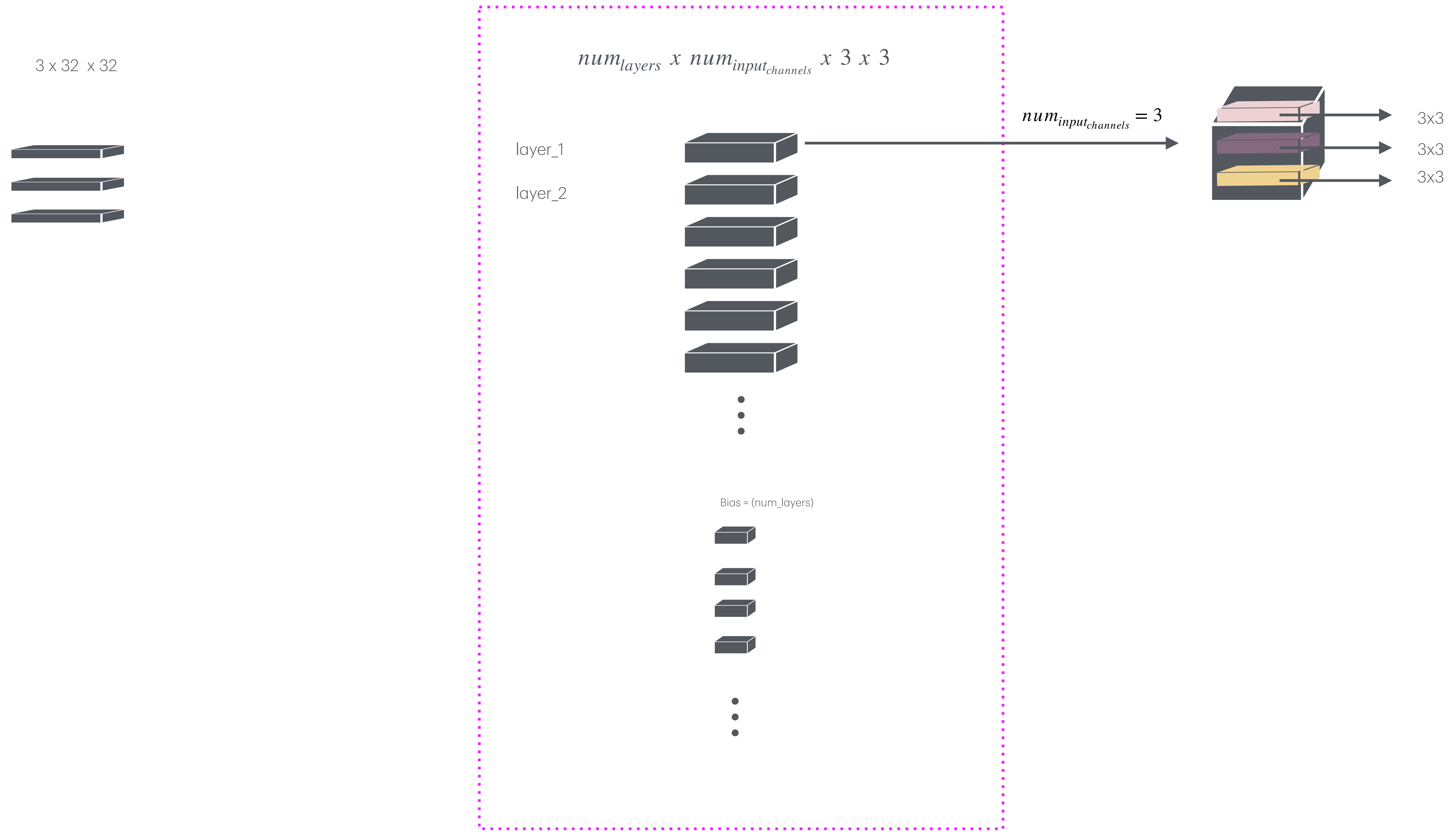
Stride <= f_h

Output Shape : Case 2



Convolutional Layer: Build

$batch_{size} = 1$ Channel First Format



Convolutional Layer: Output

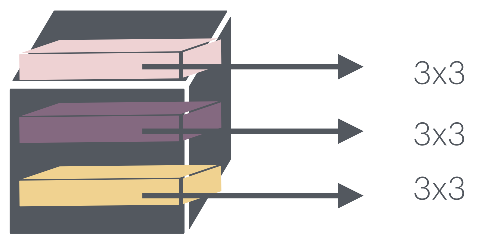
$batch_{size} = 1$

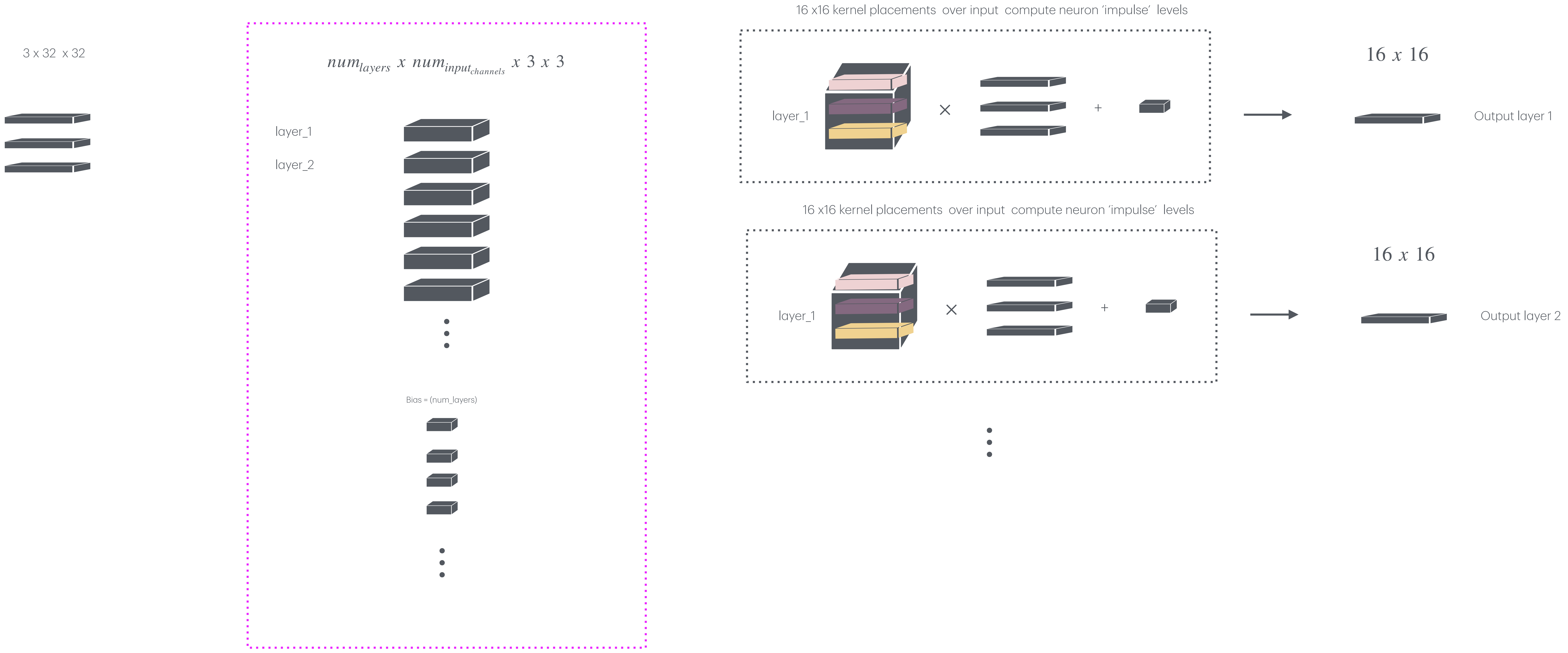
Channel First Format

Kernel - 3x3

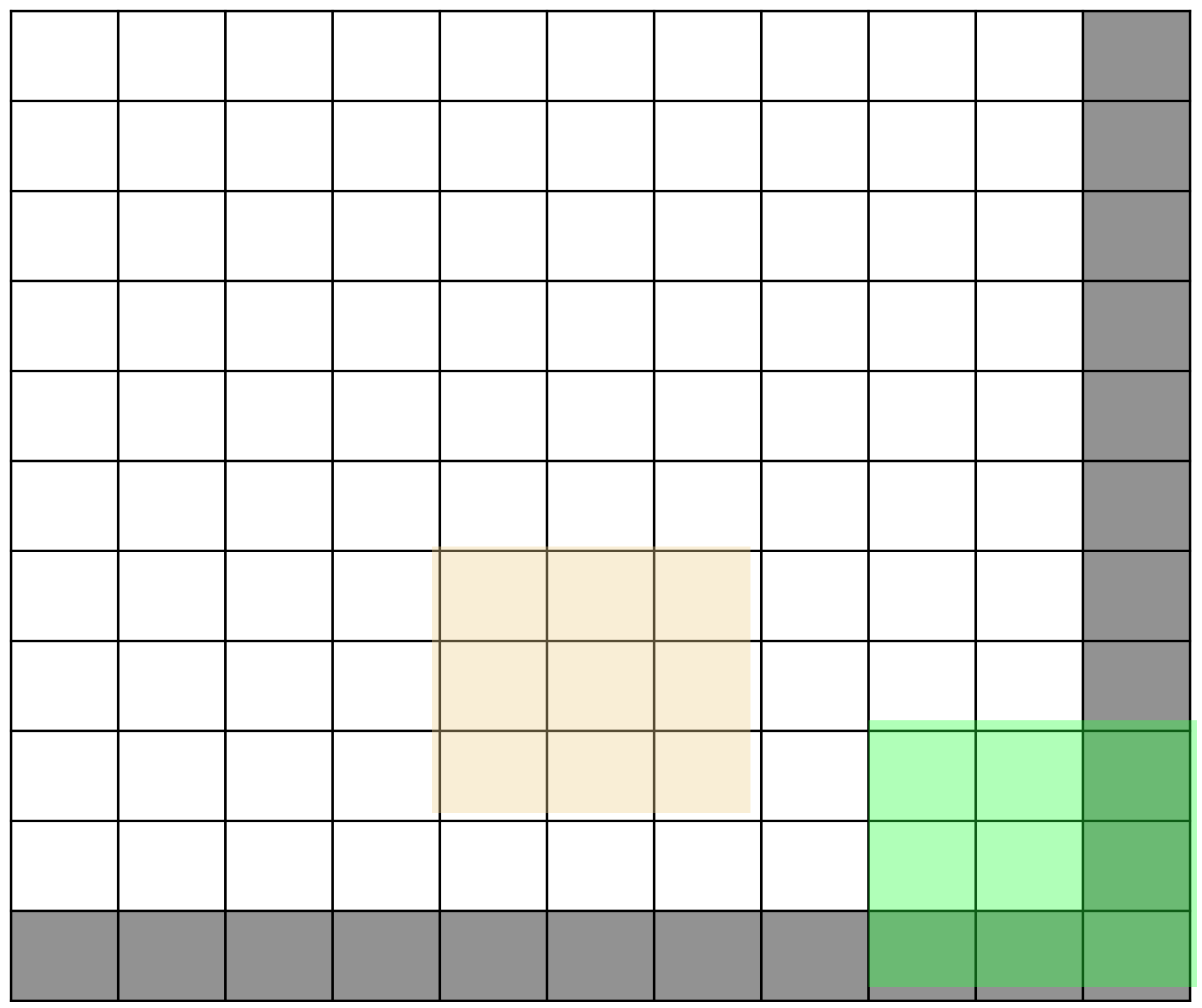
Layers_N = 20

Stride = 2

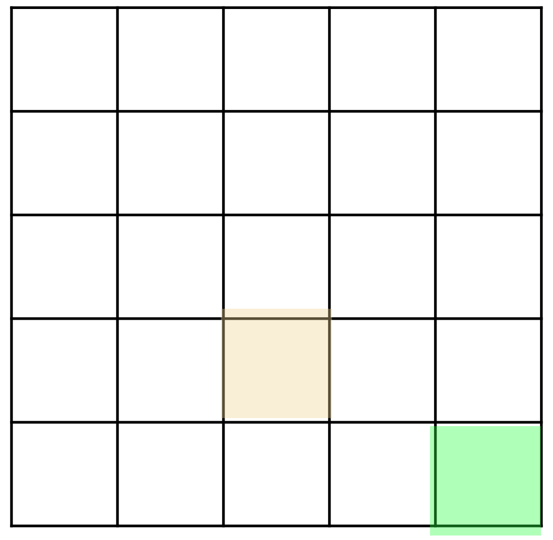




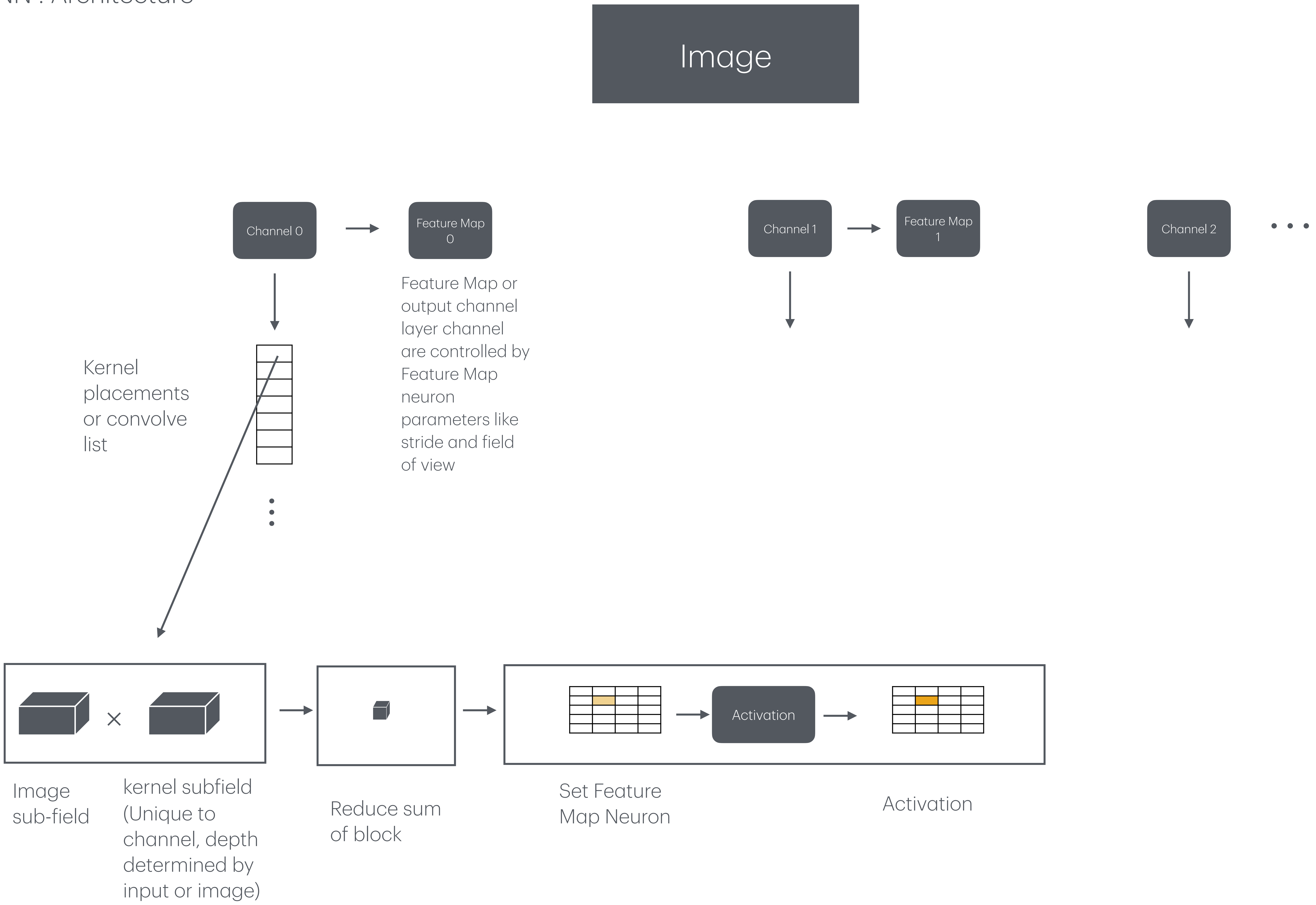
10x10x1 w/ padding



Single Feature/Channel



CNN : Architecture



Two example placements are shown in previous slide

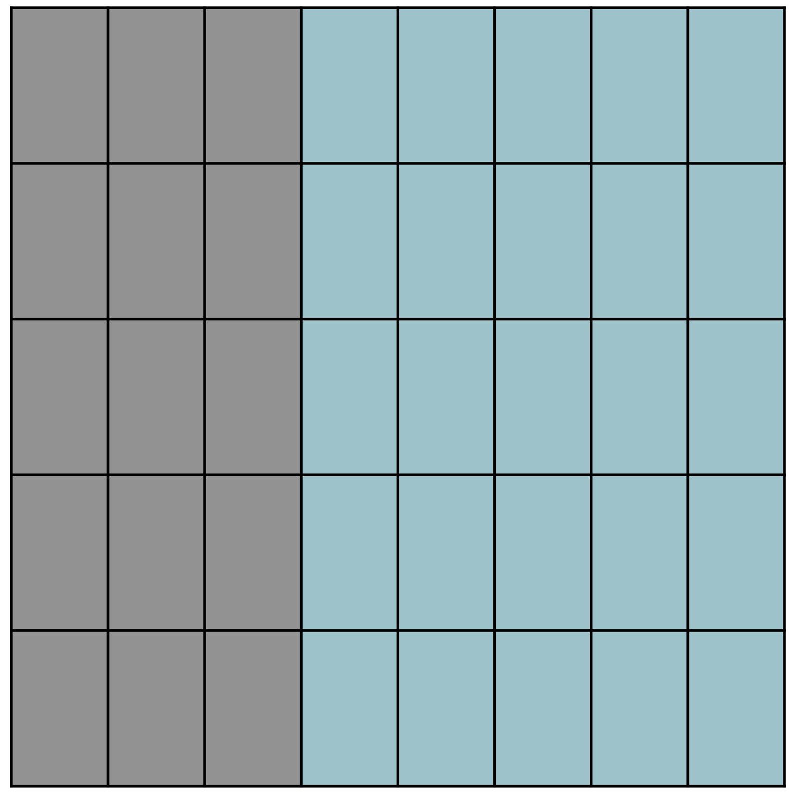
e.g. 784 neurons with 28x28 receptive field

Padding

tf.pad(object, [0,0] , [3,0], [0,0]

```
[[[253]
  [964]
  [ 93]
  [227]
  [ 89]]

 [[512]
  [874]
  [437]
  [292]
  [691]]

 [[229]
  [879]
  [382]
  [796]
  [580]]

 [[435]
  [885]
  [980]
  [143]
  [454]]

 [[689]
  [558]
  [976]
  [681]
  [197]]], shape=(5, 5, 1), dtype=int32)
```

253	964	93	227	89
512	874	437	292	691
229	879	382	796	580
435	885	980	143	454
689	558	976	681	197



```
<tf.Tensor: shape=(5, 8, 1), dtype=int32, numpy=
array([[[  0],
        [  0],
        [  0],
        [689],
        [180],
        [221],
        [131],
        [133]],

       [[  0],
        [  0],
        [  0],
        [520],
        [ 35],
        [623],
        [848],
        [ 61]],

       [[  0],
        [  0],
        [  0],
        [702],
        [674],
        [140],
        [404],
        [142]],

       [[  0],
        [  0],
        [  0],
        [ 35],
        [884],
        [803],
        [ 47],
        [461]],

       [[  0],
        [  0],
        [  0],
        [104],
        [403],
        [ 18],
        [832],
        [106]]], dtype=int32)>
```

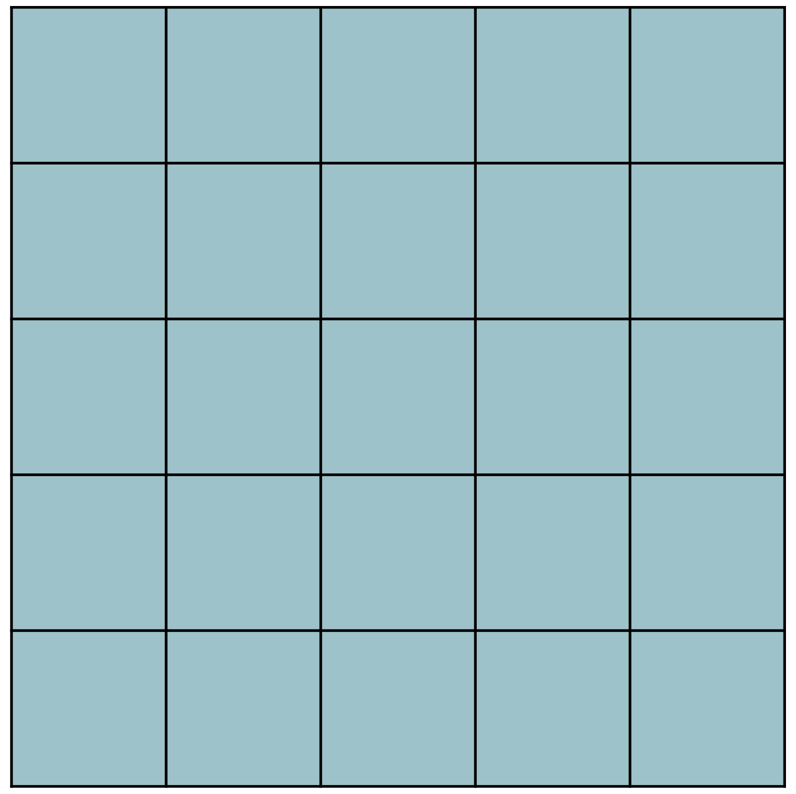
Padding

tf.pad(object, [0,0] , [3,3], [0,0]

			129	663	523	3	584			
			368	751	590	472	770			
			380	442	792	705	409			
			149	456	188	763	254			
			187	150	3	789	465			

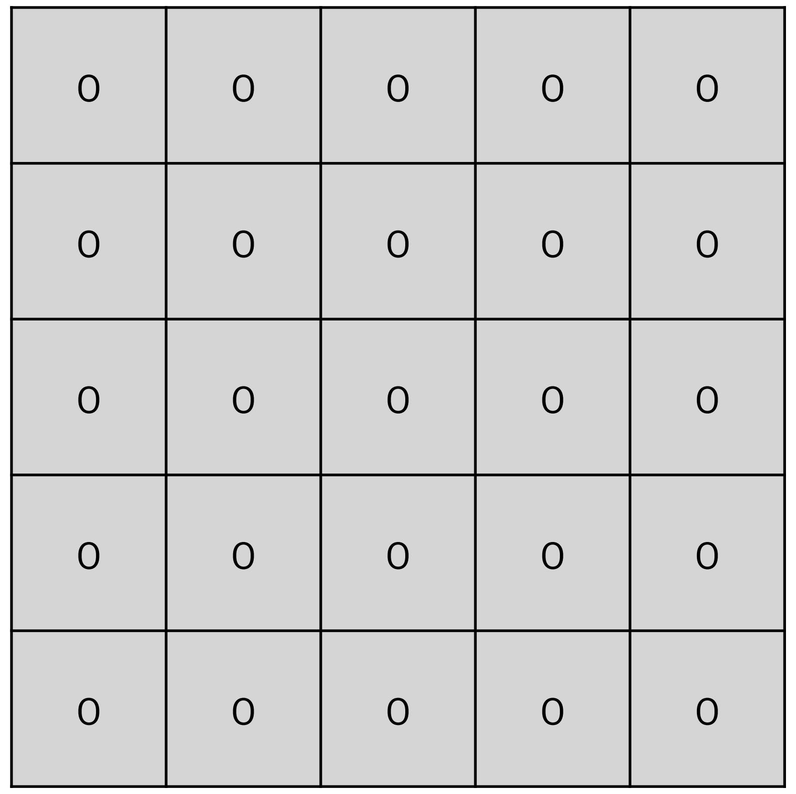
```
<tf.Tensor: shape=(5, 11, 1), dtype=int32, numpy=
array([[[  0],
        [  0],
        [  0],
        [129],
        [663],
        [523],
        [  3],
        [584],
        [  0],
        [  0],
        [  0]],

       [[  0],
        [  0],
        [  0],
        [368],
        [751],
        [598],
        [472],
        [770],
        [  0],
        [  0],
        [  0]],

       [[  0],
        [  0],
        [  0],
        [380],
        [442],
        [792],
        [705],
        [409],
        [  0],
        [  0],
        [  0]],

       [[  0],
        [  0],
        [  0],
        [149],
        [456],
        [188],
        [763],
        [254],
        [  0],
        [  0],
        [  0]],

       [[  0],
        [  0],
        [  0],
        [187],
        [150],
        [  3],
        [789],
        [465],
        [  0],
        [  0],
        [  0]]], dtype=int32)>
```

Padding

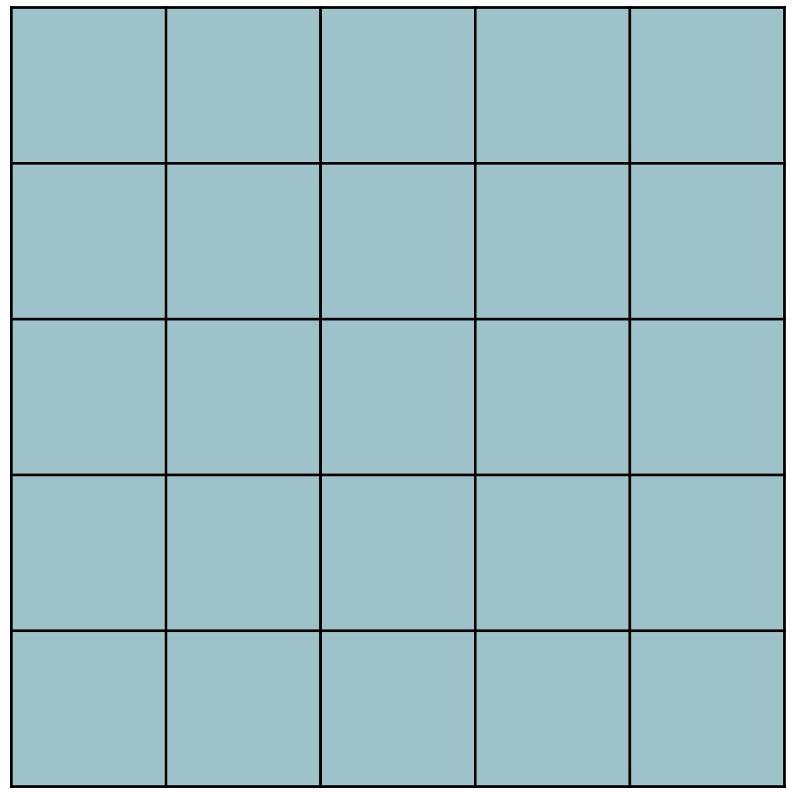


tf.pad(a, [[0,0], [0,0], [1,0]])

Top Layer

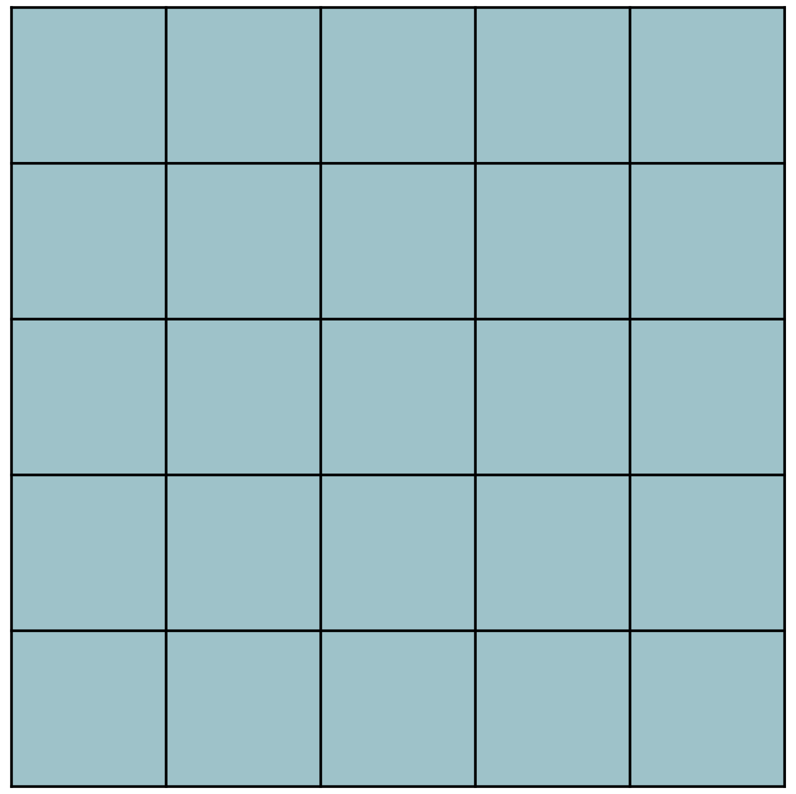


Top Layer



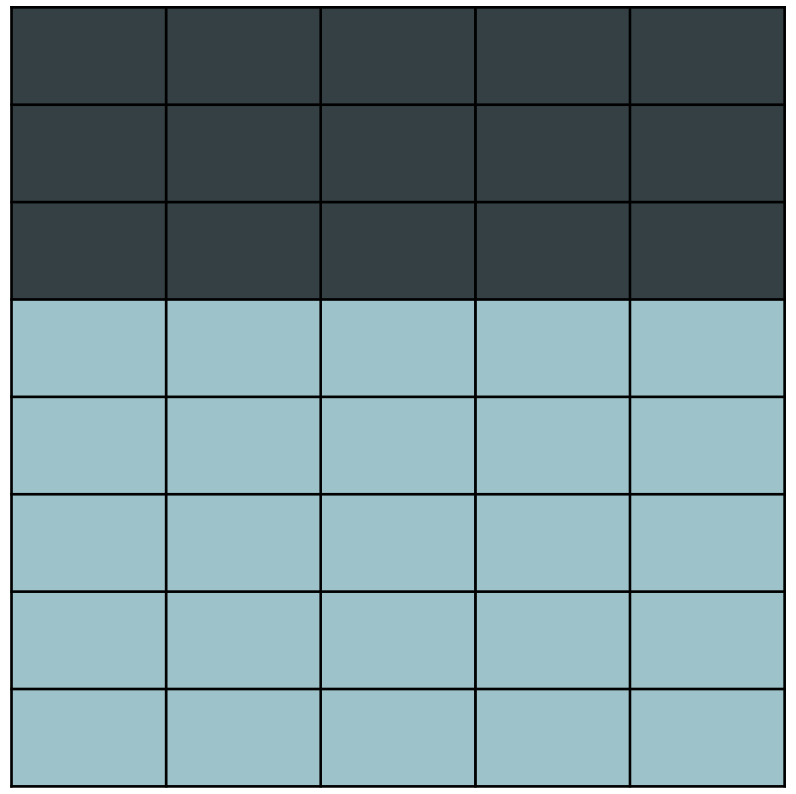
```
<tf.Tensor: shape=(5, 5, 2), dtype=int32, numpy=
array([[[  0, 146],
        [  0, 632],
        [  0, 326],
        [  0, 871],
        [  0, 990]],

       [[  0, 996],
        [  0,  80],
        [  0, 553],
        [  0, 221],
        [  0, 701]],

       [[  0, 572],
        [  0, 374],
        [  0, 113],
        [  0, 197],
        [  0, 723]],

       [[  0, 693],
        [  0, 409],
        [  0, 450],
        [  0, 934],
        [  0, 491]],

       [[  0, 144],
        [  0, 834],
        [  0, 695],
        [  0, 399],
        [  0, 279]]], dtype=int32)>
```

Padding



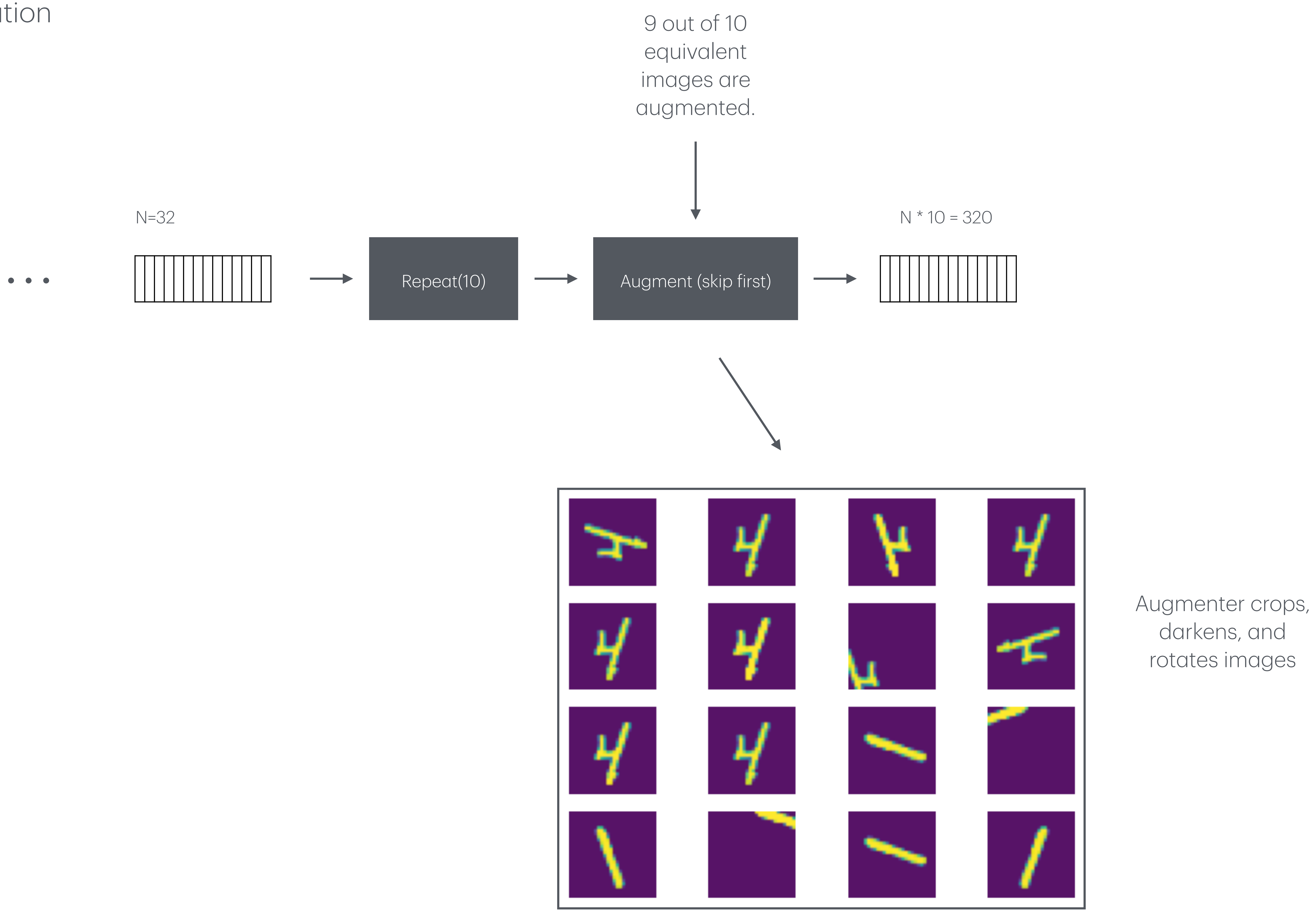
```
tf.Tensor(
[[[264]
  [410]
  [274]
  [ 88]
  [571]]

 [[113]
  [176]
  [232]
  [732]
  [820]]

 [[681]
  [921]
  [623]
  [530]
  [600]]

 [[740]
  [265]
  [855]
  [144]
  [845]]

 [[471]
  [536]
  [241]
  [948]
  [487]]], shape=(5, 5, 1), dtype=int32)
```

tf.pad(a, [[3,0], [0,0], [0,0]])

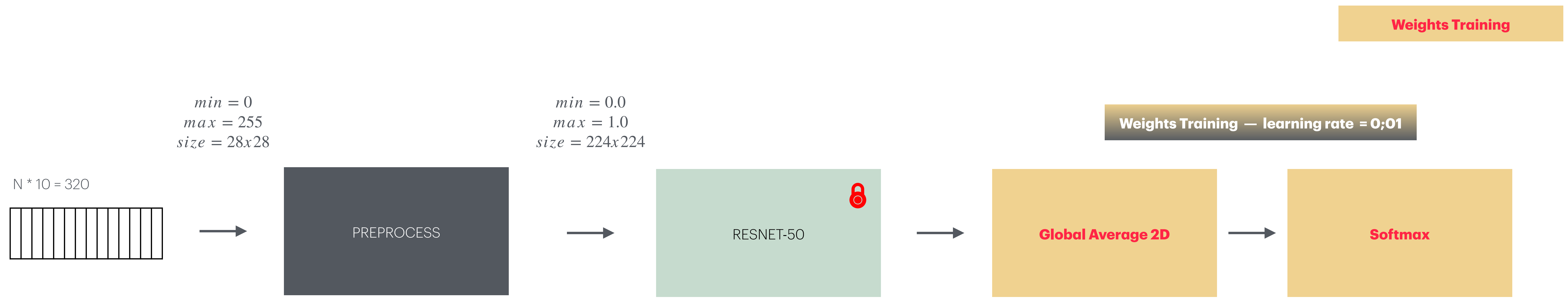


```
tf.Tensor(
[[[  0]
  [  0]
  [  0]
  [  0]
  [  0]]

 [[  0]
  [  0]
  [  0]
  [  0]
  [  0]]

 [[264]
  [410]
  [274]
  [ 88]
  [571]]

 [[113]
  [176]
  [232]
  [732]
  [820]]

 [[681]
  [921]
  [623]
  [530]
  [600]]

 [[740]
  [265]
  [855]
  [144]
  [845]]

 [[471]
  [536]
  [241]
  [948]
  [487]]], shape=(7, 5, 1), dtype=int32)
<tf.Tensor: shape=(), dtype=float32, numpy=-1.632741093635559>
```

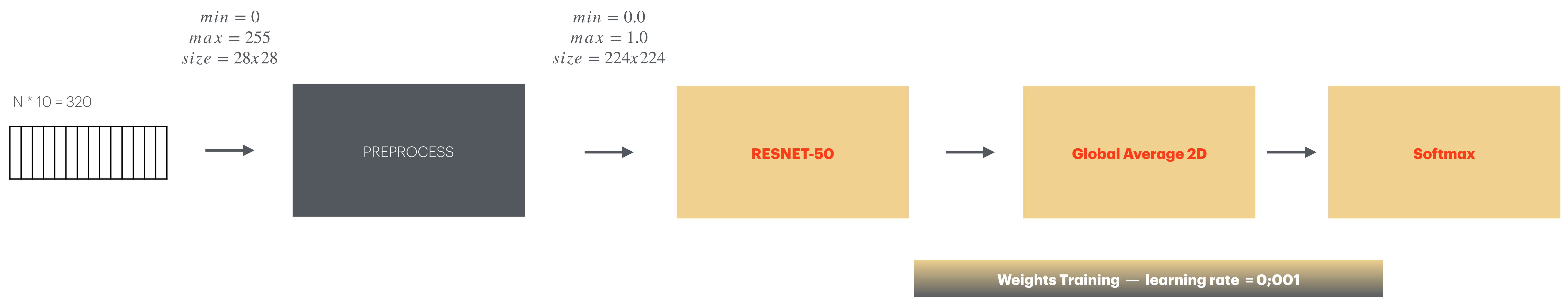
Preprocess_Augmentation



Pretrain



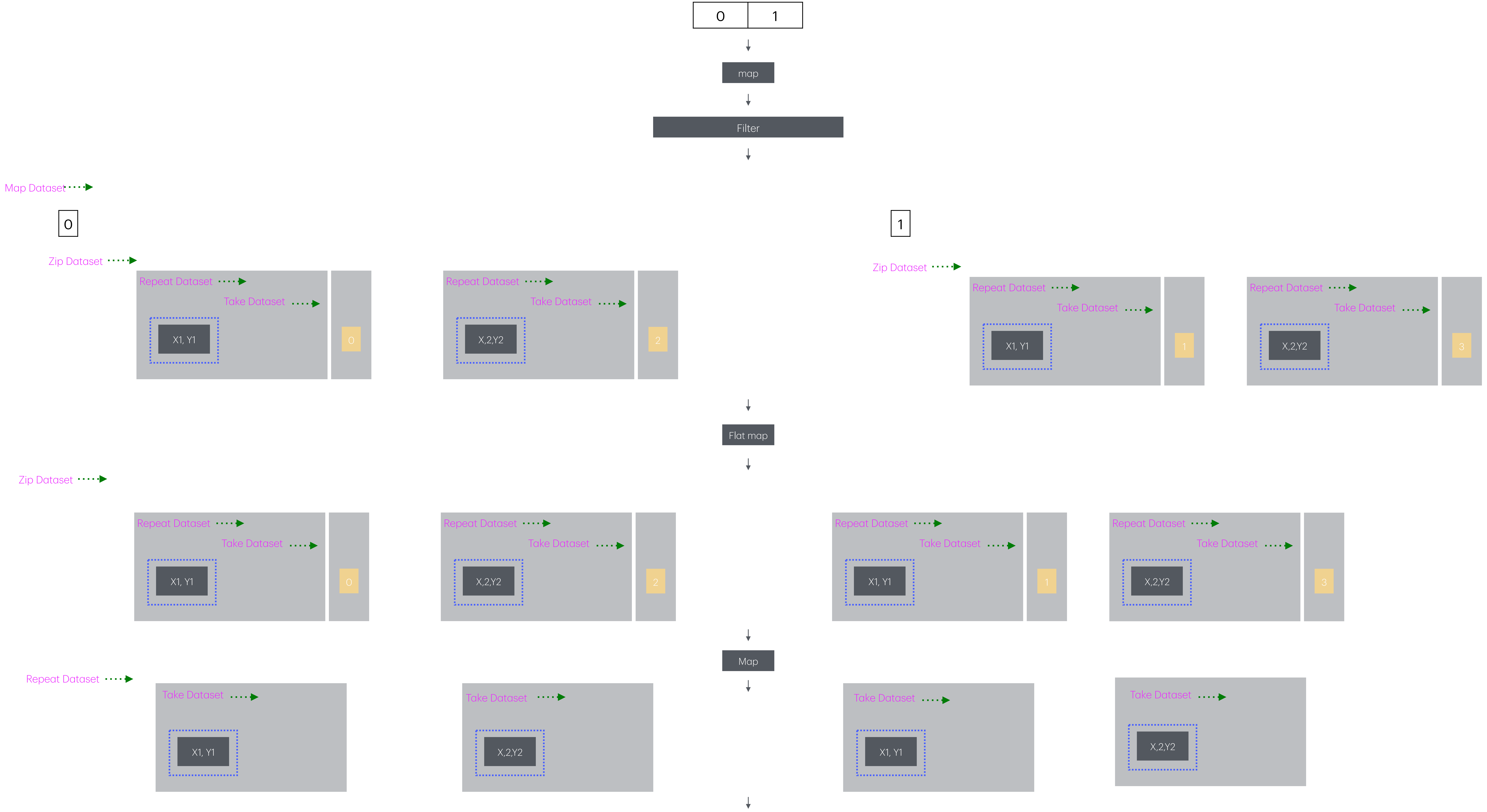
Trains upper two layers until search algorithm stops improving



Reducing learning rate after unlocking RESNET , ensures RESET Model will not diverge from its optimal model 'space'

Tensorflow Dataset Example

Dataset Example cont.



Dataset Example cont.