

# Introduction to Deep Learning for Computer Vision

# Outline

- Understanding convolutional neural networks (convnets)
- Using **data augmentation** to mitigate overfitting
- Using a **pretrained convnet** to do feature extraction
- **Fine-tuning** a pretrained convent

# Convolutional neural networks

- Computer vision is the problem domain that led to the initial rise of deep learning between 2011 and 2015
- *Convolutional neural networks* started getting remarkably good results on image classification
  - the ICDAR 2011 Chinese character recognition competition
  - Hinton's group winning the high-profile ImageNet large-scale visual recognition challenge 2012
- Convolutional neural networks (*convnets*), the type of deep learning model that is now used almost universally in computer vision applications.

# Convolutional neural networks

- <https://www.youtube.com/@codebasics>
- <https://www.upgrad.com/blog/basic-cnn-architecture/>

9



-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

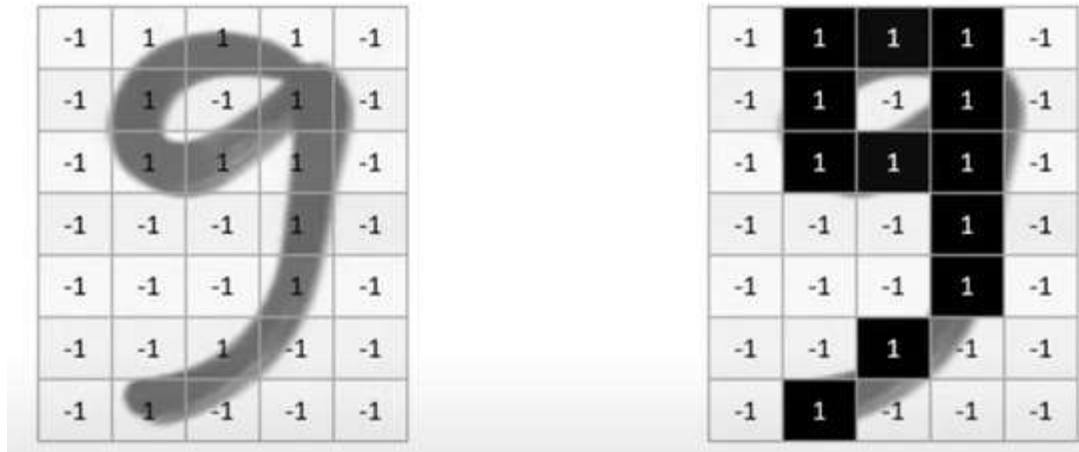


- RGB numbers 0-255

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1





- Too much hard coded.

# Location Shifted



# Variation 1

-1	-1	1	-1	-1
-1	1	-1	1	-1
-1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1



-1	-1	1	-1	-1
-1	1	-1	1	-1
-1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1



-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1
-1	1	-1	-1	-1

# Variation 2

-1	-1	1	1	-1
-1	1	-1	1	-1
-1	1	-1	1	-1
-1	-1	1	1	-1
-1	-1	-1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1



-1	-1	1	1	-1
-1	1	-1	1	-1
-1	1	-1	1	-1
-1	-1	1	1	-1
-1	-1	-1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1



-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1



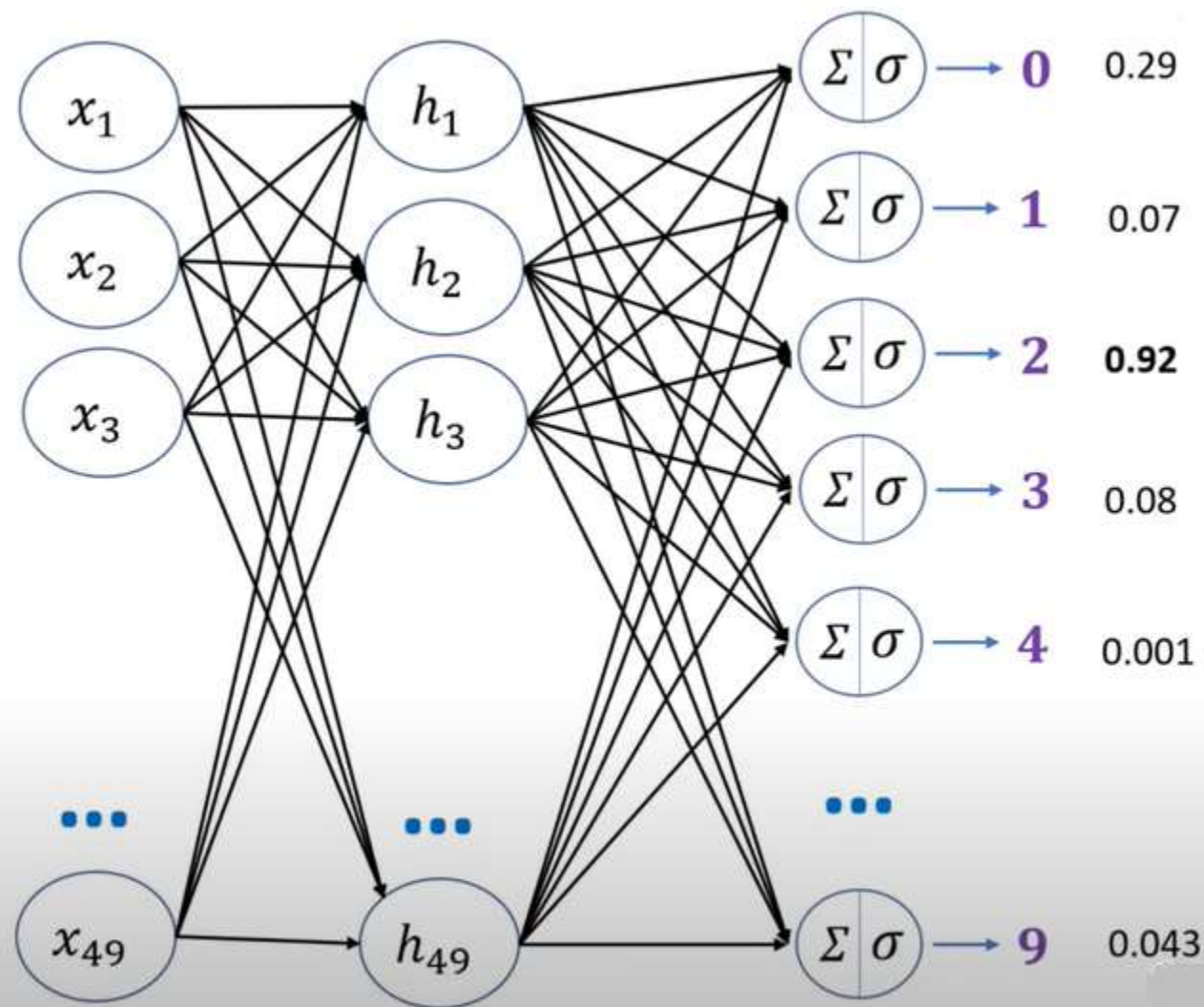
- To handle variety in digits we can use simple ANN.



0	0	0	0	0	0	0
0	87	240	210	24	0	0
0	13	0	101	195	0	0
0	35	167	99	210	0	0
0	145	230	240	201	189	140
0	0	102	67	17	13	0
0	0	0	0	0	0	0

7 by 7 grid

0  
0  
0  
0  
0  
0  
0  
87  
240  
210  
24  
0  
...  
0



$$49 \times 49 = 2401$$

$$49 \times 10 = 490$$



Image size = 1920 x 1080 X 3



Image size =  $1920 \times 1080 \times 3$

First layer neurons =  $1920 \times 1080 \times 3 \sim 6 \text{ million}$







Image size =  $1920 \times 1080 \times 3$

First layer neurons =  $1920 \times 1080 \times 3 \sim 6$  million



Hidden layer neurons = Let's say you keep it  $\sim 4$  million



Image size =  $1920 \times 1080 \times 3$

First layer neurons =  $1920 \times 1080 \times 3 \sim 6 \text{ million}$

↓

Hidden layer neurons = Let's say you keep it  $\sim 4 \text{ million}$

Weights between input and hidden layer =  $6 \text{ mil} * 4 \text{ mil}$   
 $= 24 \text{ million}$

# Disadvantages of using ANN for image classification

- Too much computation
- Treats local pixels same as pixels far apart
- Sensitive to location of an object in an image.



Koala's **eye**? = Y



Koala's **nose**? = Y



Koala's **ears**? = Y





Koala's **eye**? = Y



Koala's **nose**? = Y



Koala's **ears**? = Y



Koala's **head**? = Y





Koala's **eye**? = Y



Koala's **nose**? = Y



Koala's **ears**? = Y



Koala's **head**? = Y



Koala's **hands**? = Y



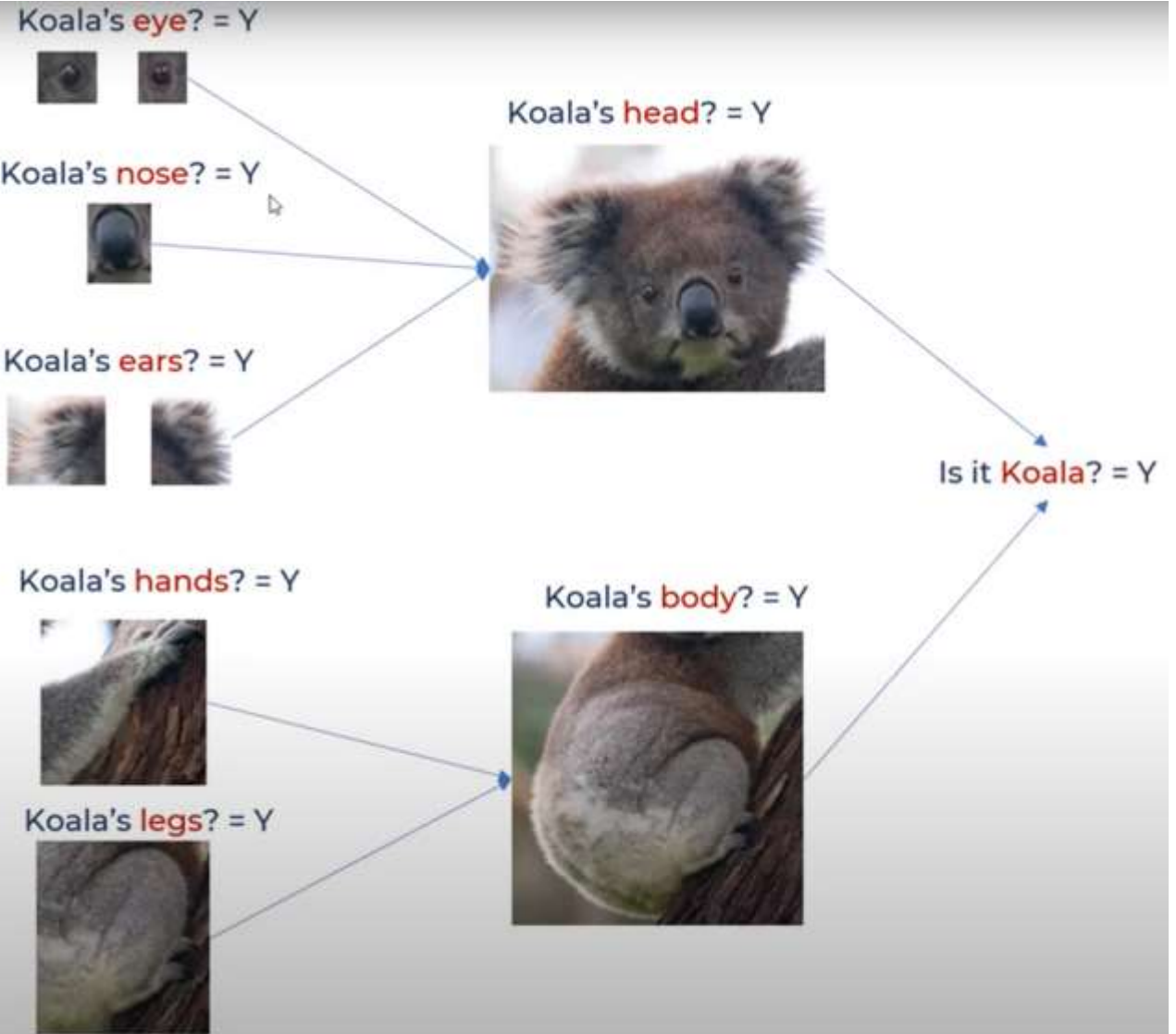
Koala's **legs**? = Y



Koala's **body**? = Y



Is it **Koala**? = Y





Loopy circle  
pattern



Vertical line



Diagonal line





-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Loopy pattern  
filter

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Vertical line  
filter

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Diagonal line  
filter



-1	<b>1</b>	<b>1</b>	<b>1</b>	-1
-1	<b>1</b>	-1	<b>1</b>	-1
-1	<b>1</b>	<b>1</b>	<b>1</b>	-1
-1	-1	-1	<b>1</b>	-1
-1	-1	-1	<b>1</b>	-1
-1	-1	<b>1</b>	-1	-1
-1	<b>1</b>	-1	-1	-1

1	1	1
1	-1	1
1	1	1

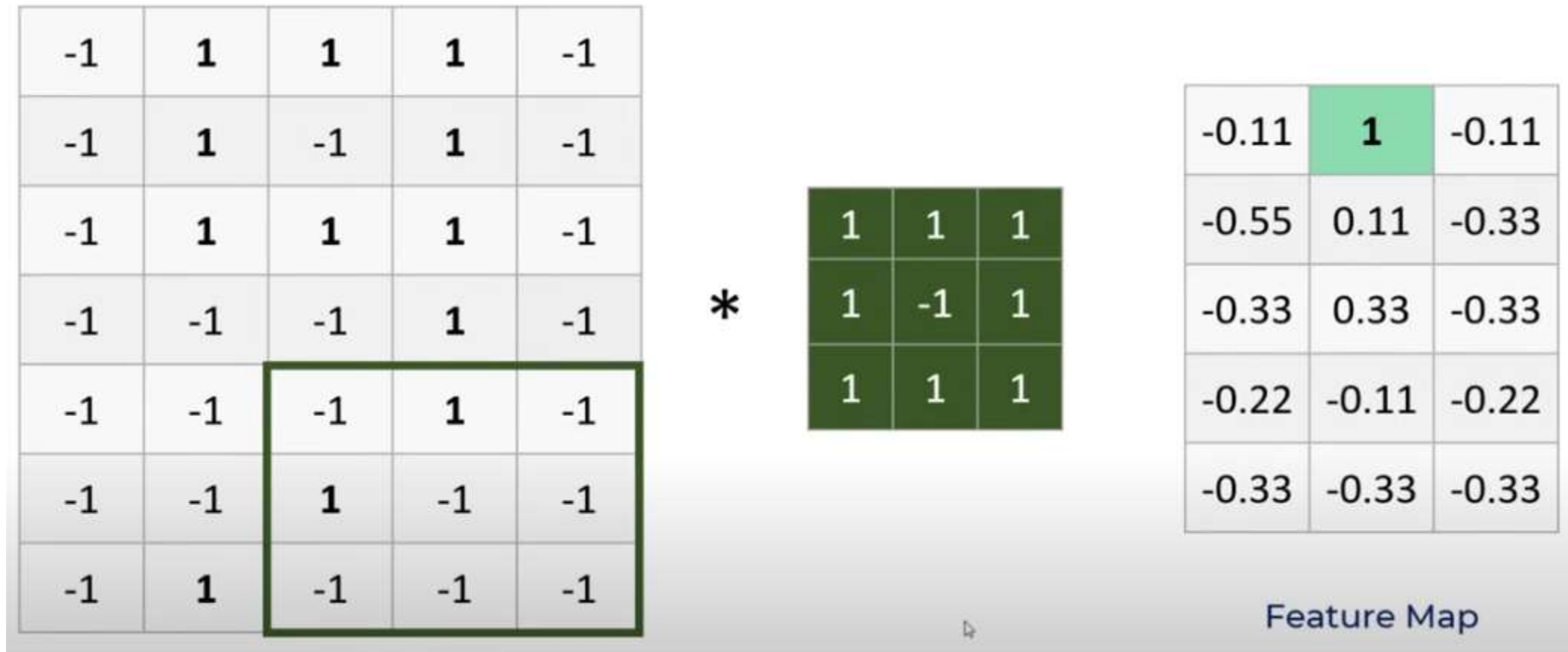
$$-1+1+1-1-1-1-1+1+1 = -1 \rightarrow -1/9 = -0.11$$

-1	<b>1</b>	<b>1</b>	<b>1</b>	-1
-1	<b>1</b>	-1	<b>1</b>	-1
-1	<b>1</b>	<b>1</b>	<b>1</b>	-1
-1	-1	-1	<b>1</b>	-1
-1	-1	-1	<b>1</b>	-1
-1	-1	<b>1</b>	-1	-1
-1	<b>1</b>	-1	-1	-1

\*


<b>1</b>	<b>1</b>	<b>1</b>
<b>1</b>	<b>-1</b>	<b>1</b>
<b>1</b>	<b>1</b>	<b>1</b>

-0.11		



- Stride: 1, 2
- Kernel size: 3x3, 5x5
- Padding: valid, same

Loopy pattern detector

 \* 

1	1	1
1	-1	1
1	1	1

 = 

	1	

Loopy pattern detector


Diagram illustrating a convolution operation:

- Input: A handwritten digit '6'.
- Operation: Convolution with a 3x3 kernel (Loopy pattern detector).
- Kernel Matrix:

1	1	1
1	-1	1
1	1	1

- Result: A 3x3 grid where the bottom-middle cell contains the value 1.

Loopy pattern detector

 \*

1	1	1
1	-1	1
1	1	1

=

	1	

[illegible]

Loopy pattern detector

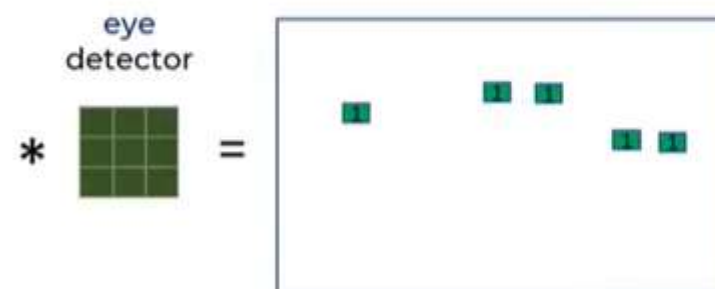
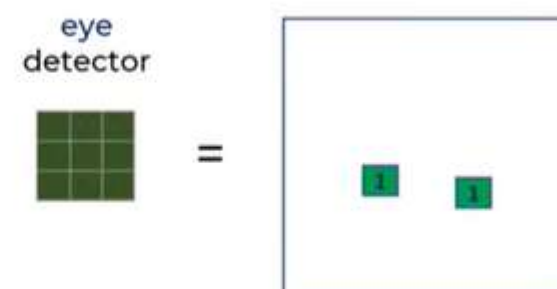
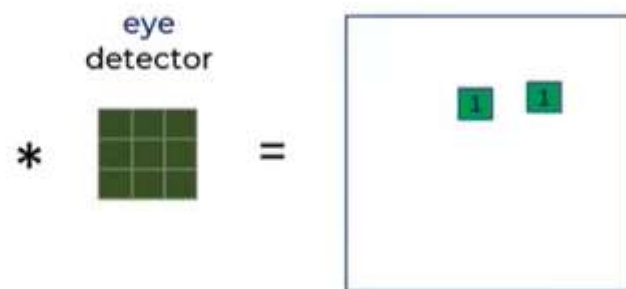
$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

[illegible]

# Introduction to convnets

- Filters are nothing but the feature detectors

Location invariant: It can detect eyes in any location of the image





\*

hands  
detector



=





Loopy pattern detector

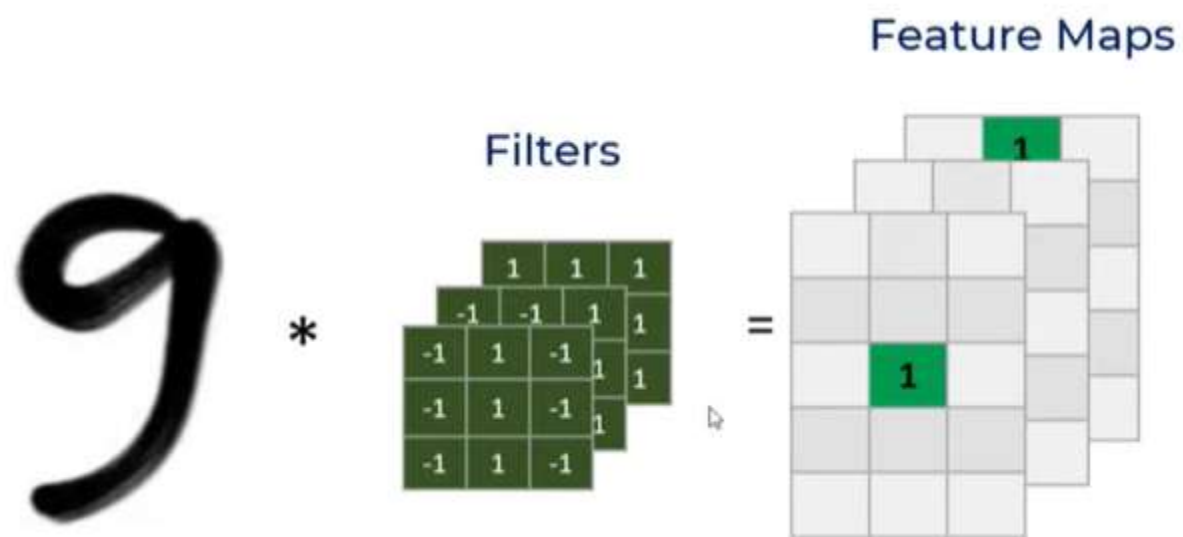
$$\begin{array}{c} \text{9} \end{array} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} & 1 & \\ & & \\ & & \\ & & \\ & & \\ & & \end{bmatrix}$$

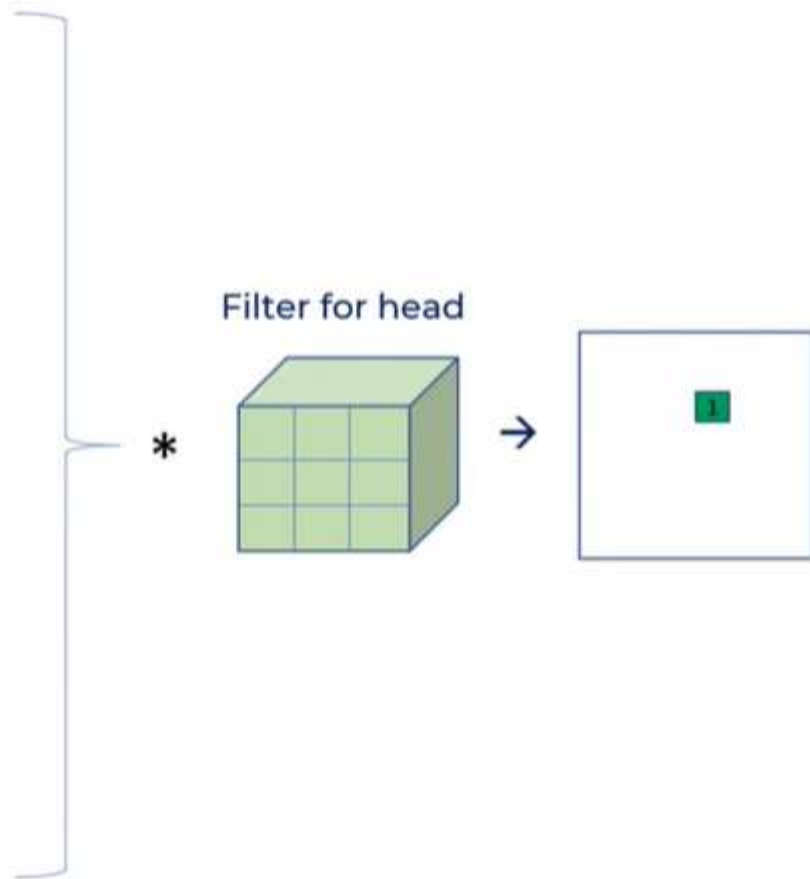
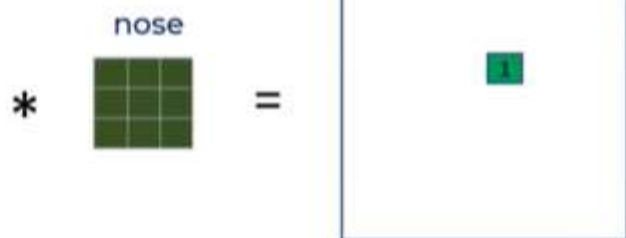
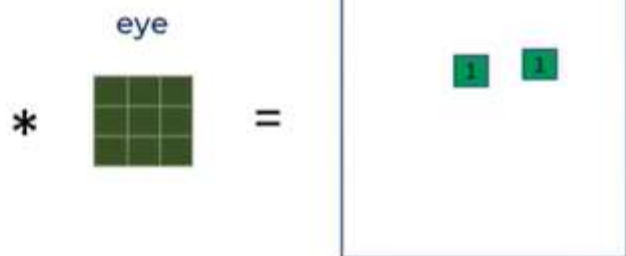
Vertical line detector

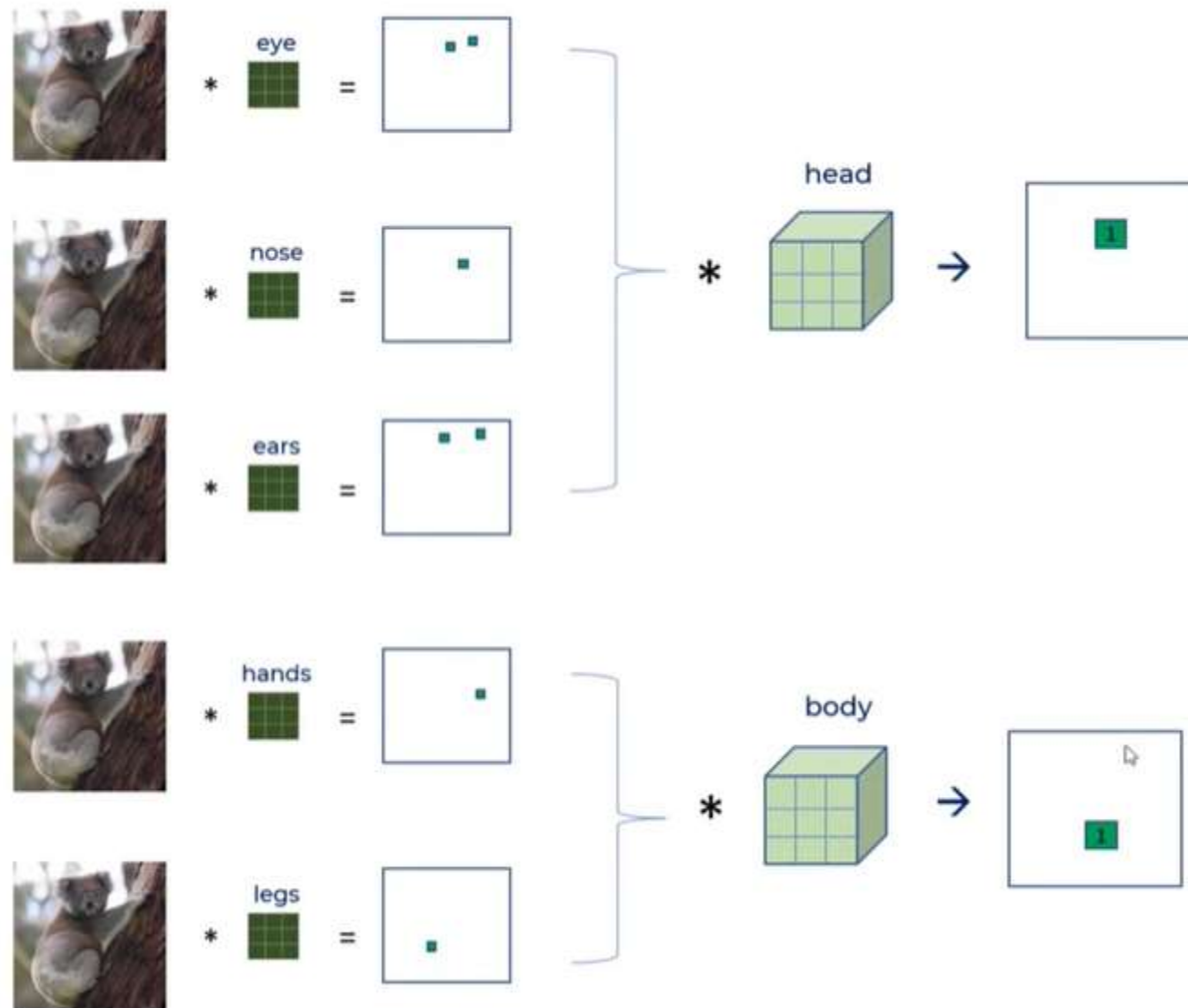
$$\begin{array}{c} \text{9} \end{array} * \begin{bmatrix} -1 & 1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ & 1 & \\ & & \\ & & \\ & & \end{bmatrix}$$

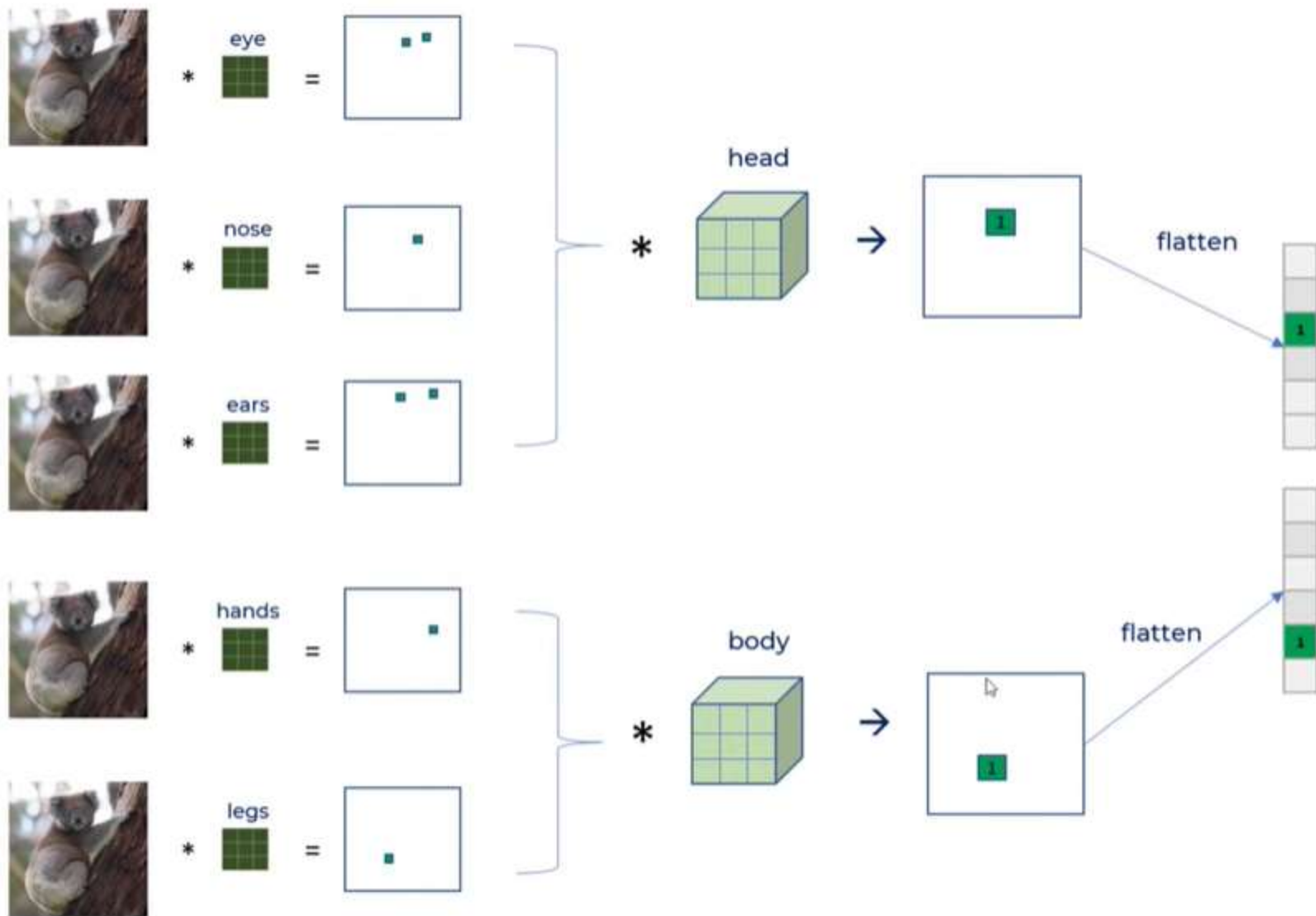
Diagonal line detector

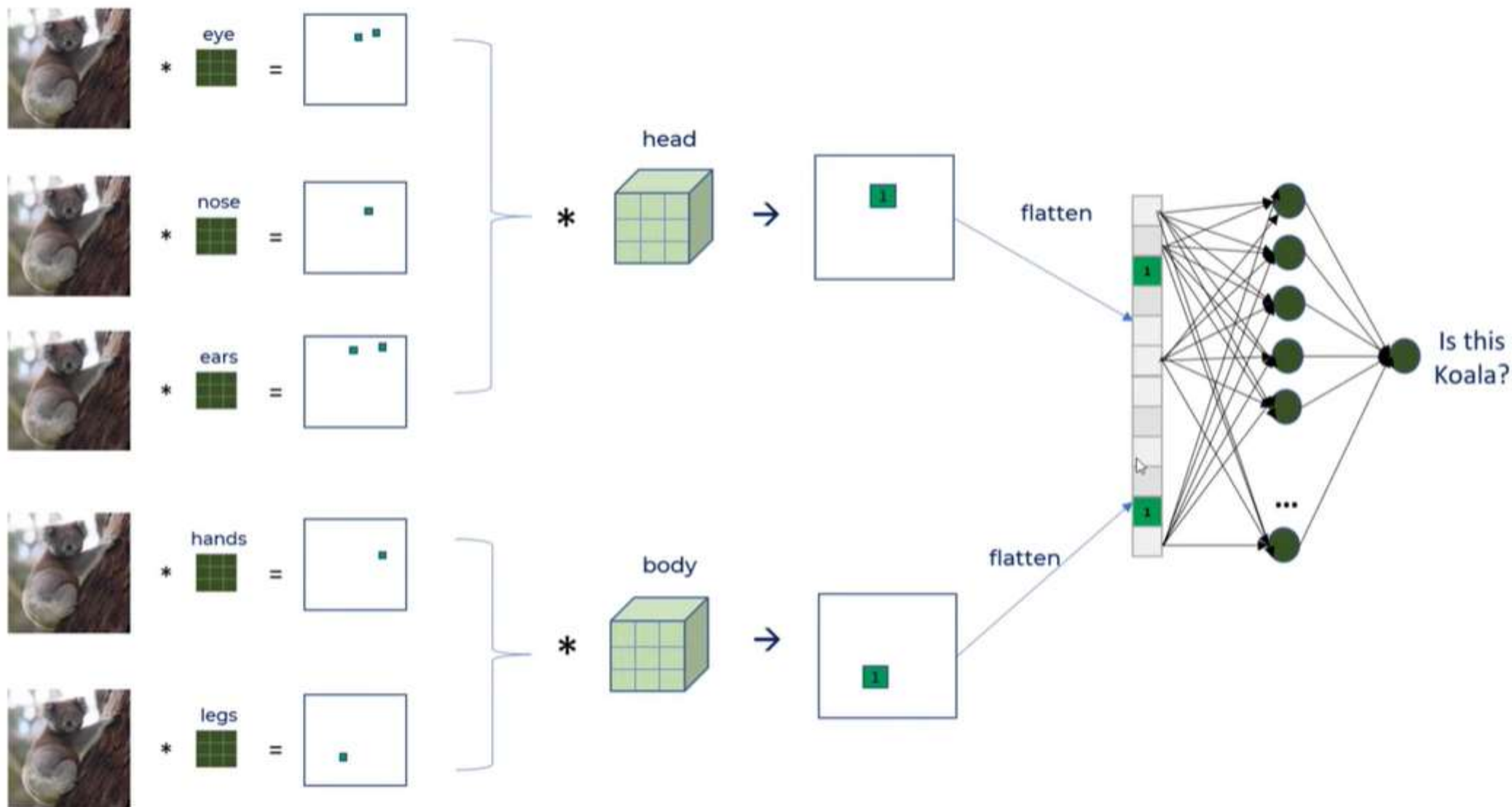
$$\begin{array}{c} \text{9} \end{array} * \begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ & & \\ & 1 & \\ & & \\ & & \end{bmatrix}$$

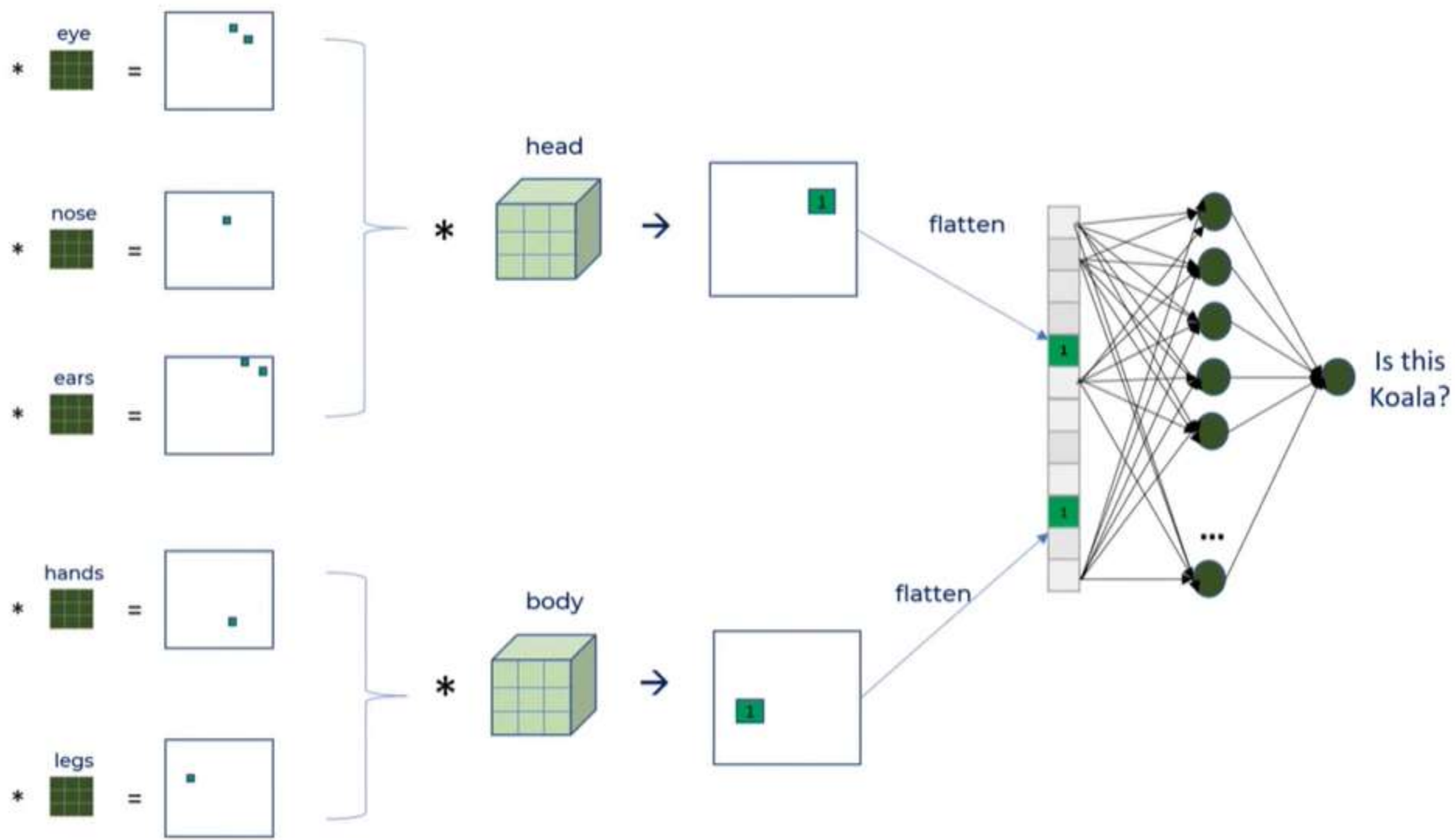


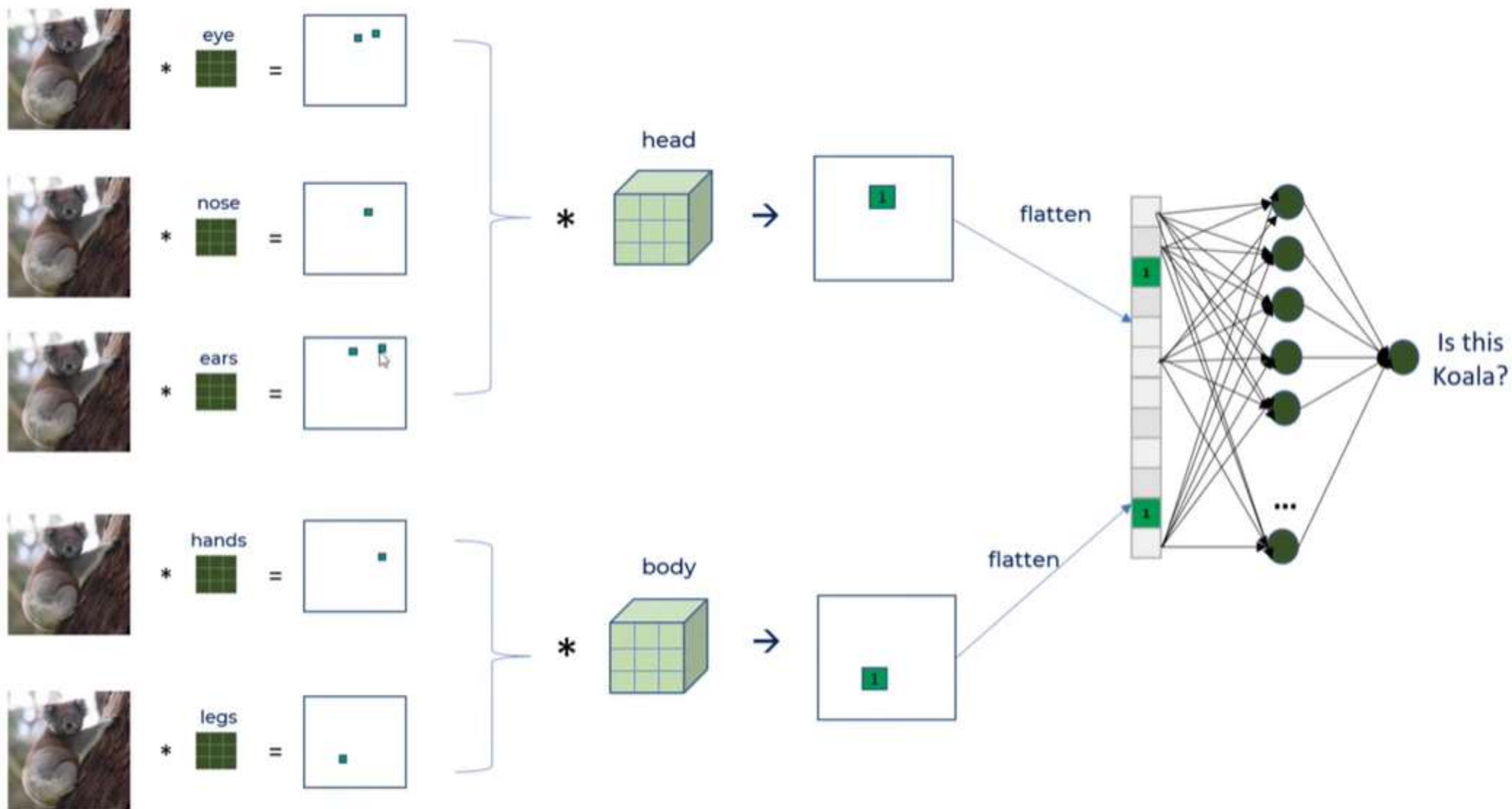




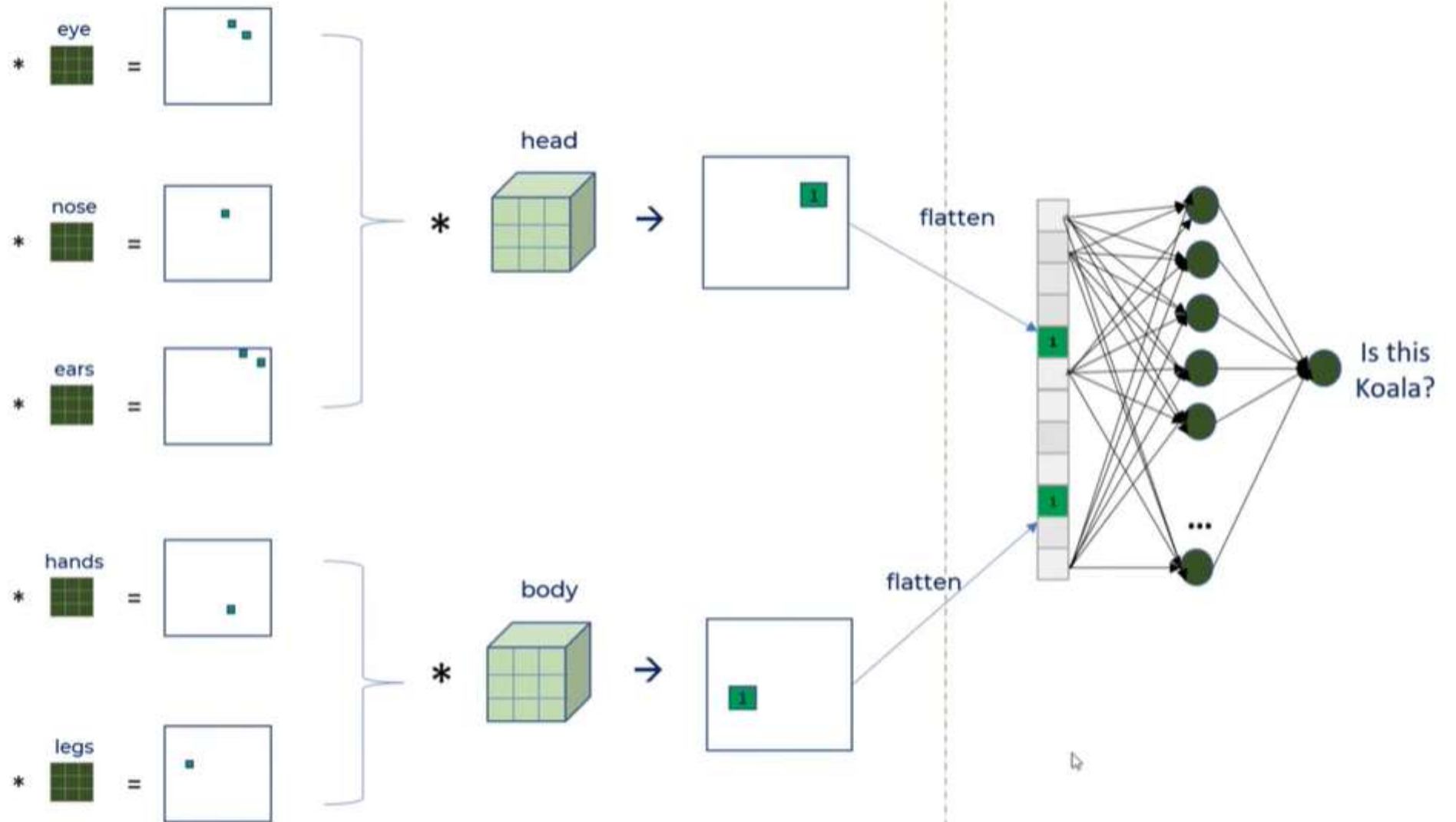




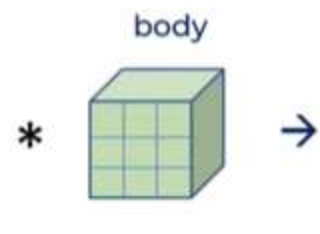
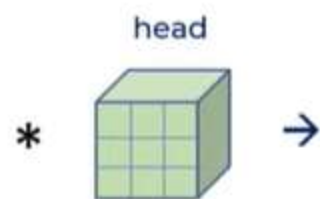






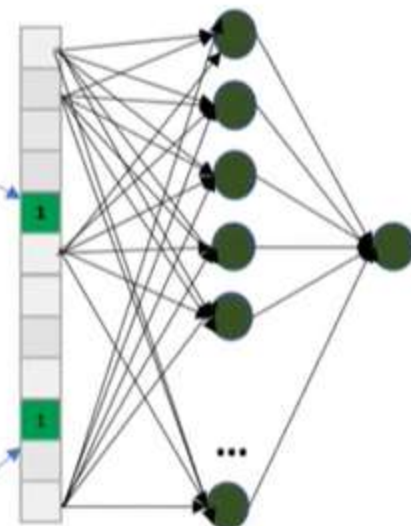


Feature Extraction



flatten

flatten



Is this  
Koala?

Feature Extraction

Classification

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

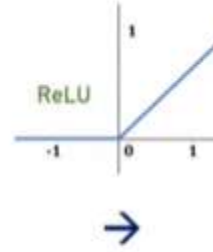
\*

Loopy pattern  
filter

1	1	1
1	-1	1
1	1	1



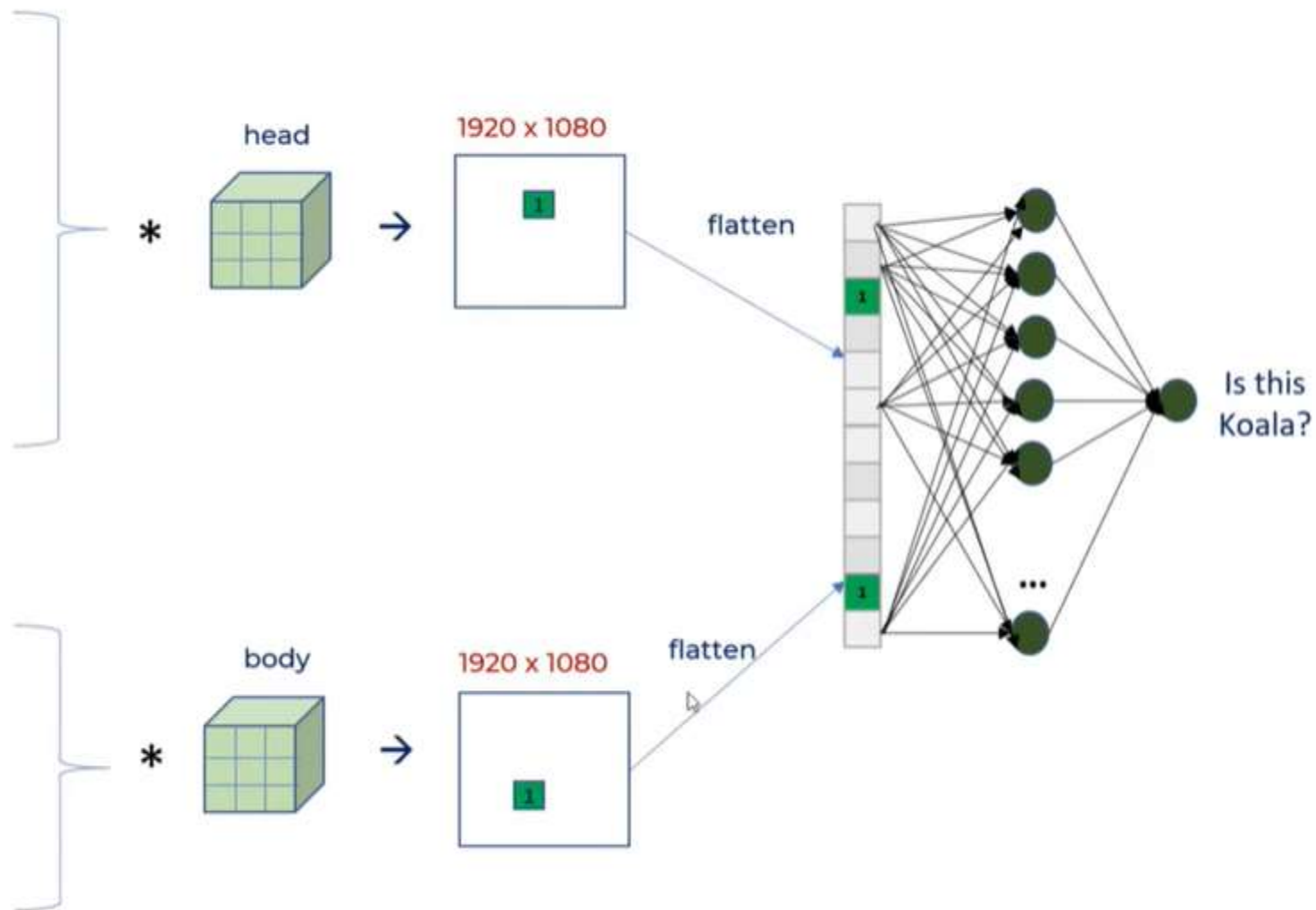
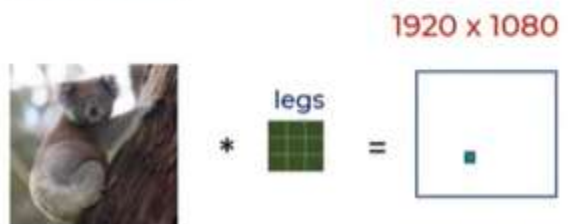
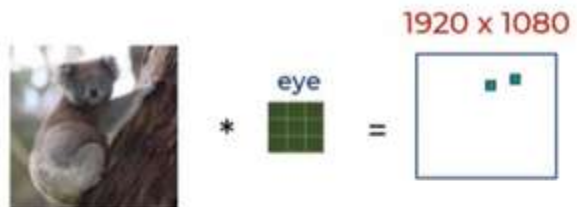
-0.11	1	-0.11
-0.55	0.11	-0.33
-0.33	0.33	-0.33
-0.22	-0.11	-0.22
-0.33	-0.33	-0.33



0	1	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0

# Introduction to convnets

- We didn't address the issue of too much computation.



# Max Pooling

5	1	3	4
8	2	9	2
1	3	0	1
2	2	2	0

8	<b>9</b>
3	2

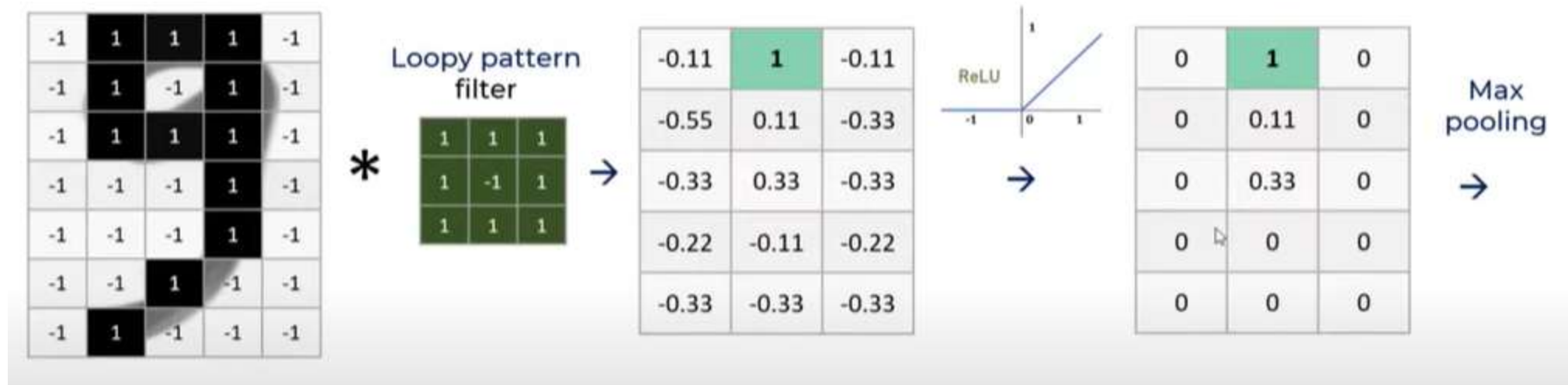
2 by 2 filter with stride = 2

# Average pooling

5	1	3	4
8	2	9	2
1	3	0	1
2	2	2	0

4	4.5
2	0.75

# Max Pooling





# Max Pooling

0	<b>1</b>	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0

<b>1</b>	

2 by 2 filter with stride = 1

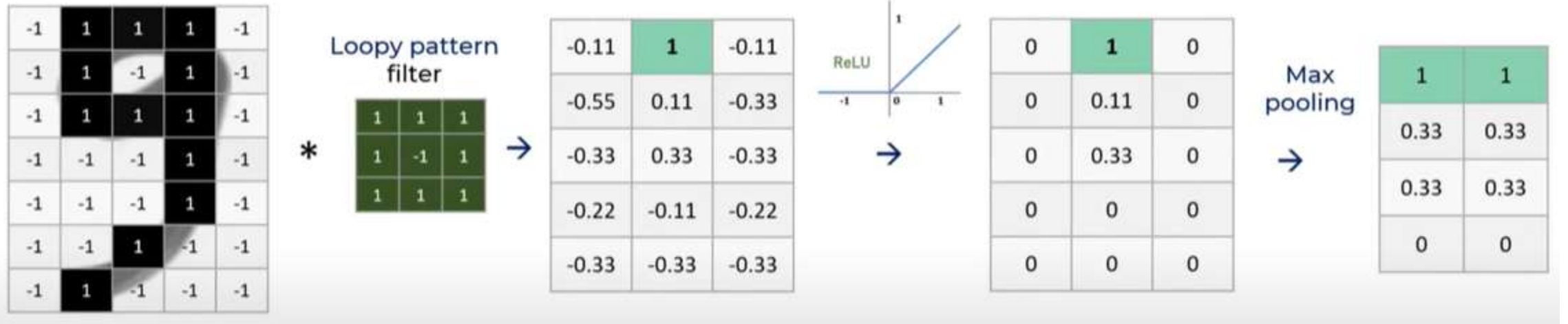
# Max Pooling

0	<b>1</b>	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0

<b>1</b>	<b>1</b>
0.33	0.33
0.33	0.33
0	0

2 by 2 filter with stride = 1

# Max Pooling



Shifted 9 at  
different position

1	1	1	-1	-1
1	-1	1	-1	-1
1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1
1	-1	-1	-1	-1

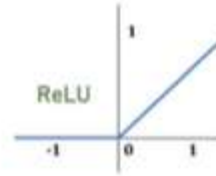
Loopy pattern  
filter

\*

1	1	1
1	-1	1
1	1	1



1	-0.11	-0.11
0.11	-0.33	0.33
0.33	-0.33	-0.33
-0.11	-0.55	-0.33
-0.55	-0.33	-0.55



1	0	0
0.11	0	0.33
0.33	0	0
0	0	0
0	0	0

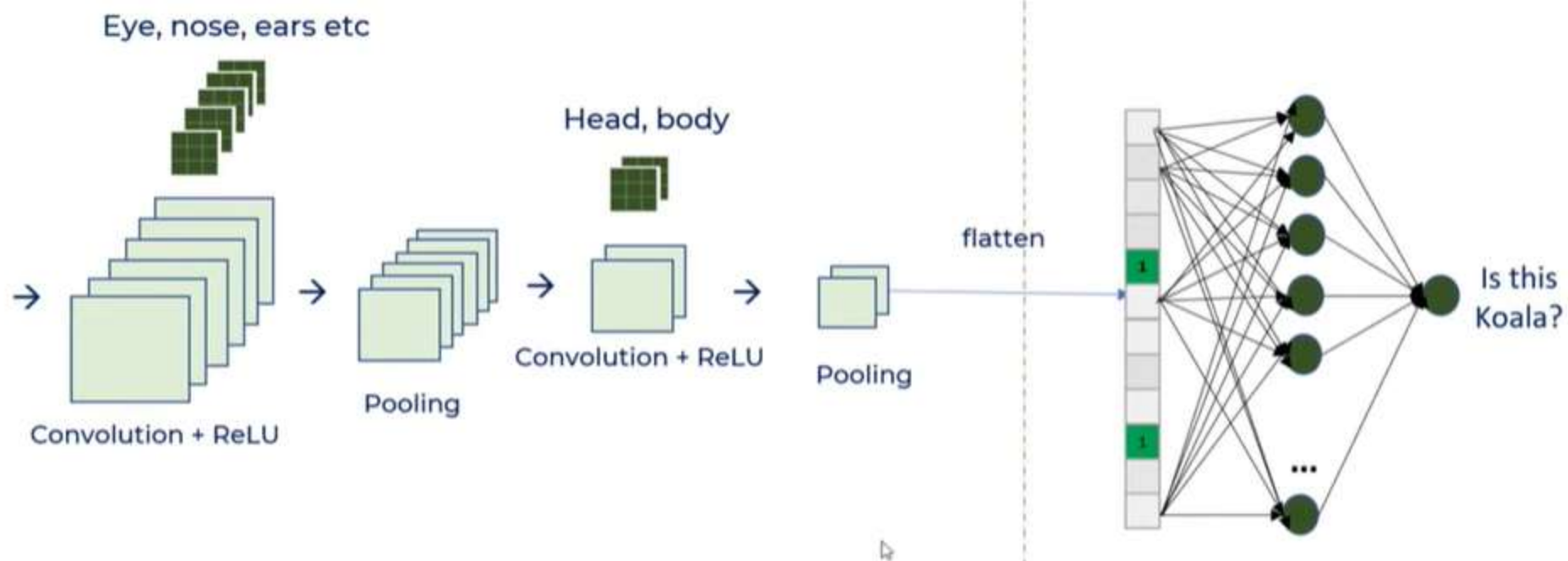
Max  
pooling



1	0.33
0.33	0.33
0.33	0
0	0

# Benefits of pooling

- Reduces dimensions and computation.
- Reduce overfitting as there are less parameters.
- Model is tolerant towards variations, distortions.



Feature Extraction

Classification

1	1	0
4	2	1
0	2	1

Pooled Feature Map

Flattening

1
1
0
4
2
1
0
2
1

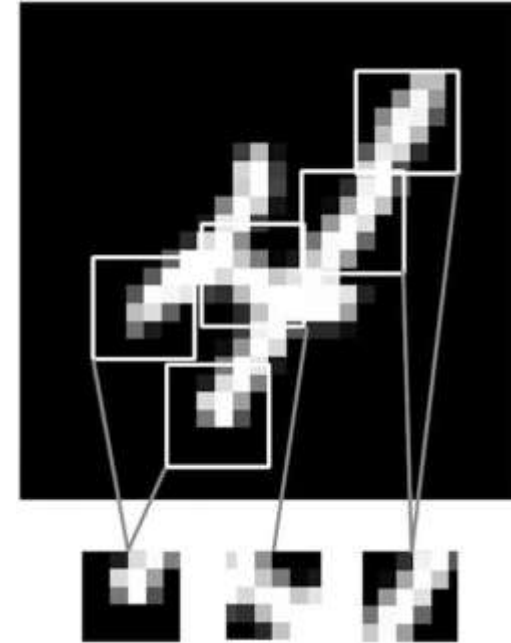
# Introduction to convnets

- These filters the network will learn on its own and part of the training.
- During training, it will figure out the values of the filters.
- Hyperparameters: how many filters and filter size.



# The convolution operation

- The fundamental difference between a densely connected layer and a convolution layer:
  - Dense layers learn global patterns in their input feature space (for example, for a MNIST digit, patterns involving all pixels),
  - Convolution layers learn local patterns—in the case of images, patterns found in small 2D windows of the inputs



# The convolution operation...

This key characteristic gives CNN (convnets) **two** interesting properties:

- *The patterns they learn are **translation-invariant**.*

After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere: for example, in the upper-left corner.

- A densely connected model would have to **learn the pattern anew** if it appeared at a new location. This makes convnets data-efficient: they need fewer training samples to learn representations that have generalization power.

# The convolution operation...

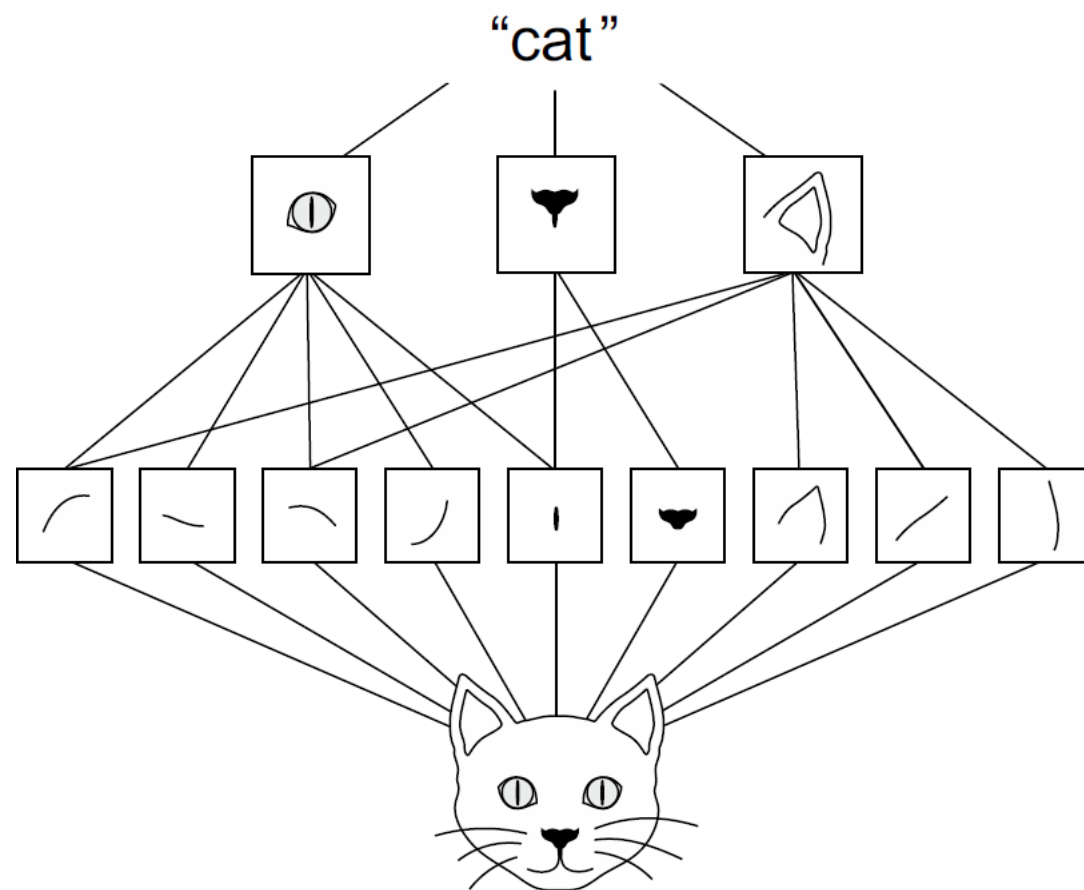
This key characteristic gives CNN (convnets) **two** interesting properties:

- *They can learn **spatial hierarchies** of patterns.*

A **first convolution layer** will learn small local patterns such as edges, a **second convolution layer** will learn larger patterns made of the features of the first layers, and so on.

- This allows convnets to efficiently learn increasingly complex and abstract visual concepts, because *the **visual world is fundamentally spatially hierarchical**.*

# The convolution operation...



# The convolution operation...

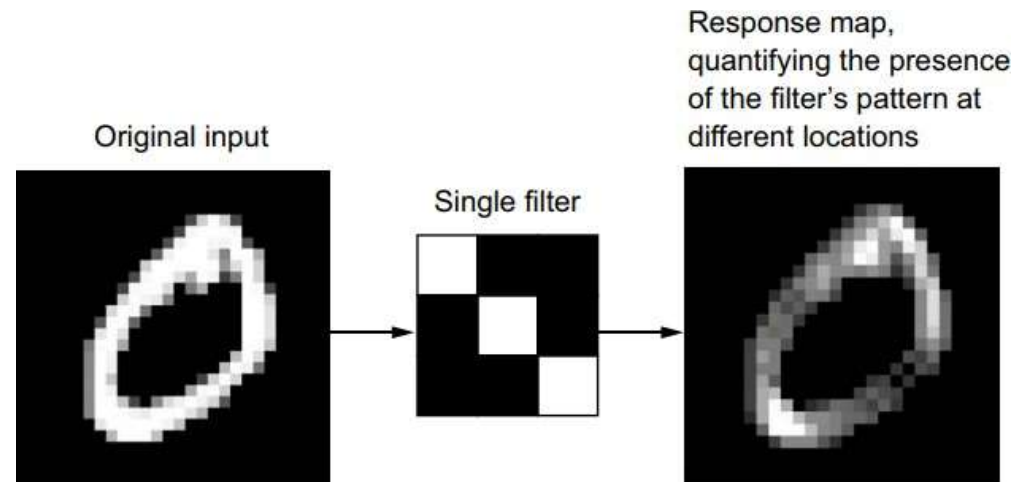
- Convolutions operate over rank-3 tensors called **feature maps**, with two spatial axes (**height and width**) as well as a depth axis (also called the **channels axis**).
- For an **RGB image**, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. For a **black-and-white picture**, like the MNIST digits, the depth is 1 (levels of gray).

# The convolution operation...

- The convolution operation extracts patches from its **input feature map** and **applies the same transformation to all of these patches**, producing an **output feature map**. This output feature map is still a rank-3 tensor: it has a width and a height.
- Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, stand for filters. Filters encode specific aspects of the input data: at a high level, **a single filter could encode the concept “presence of a face in the input,”** for instance.

# The convolution operation...

- In the MNIST example, the first convolution layer takes a feature map of size (28, 28, 1) and outputs a feature map of size (26, 26, 32): it **computes 32 filters** over its input.
- Each of these 32 output channels contains a  $26 \times 26$  grid of values, which is a response map of the filter over the input, indicating the response of that filter pattern at different locations in the input



# The convolution operation...

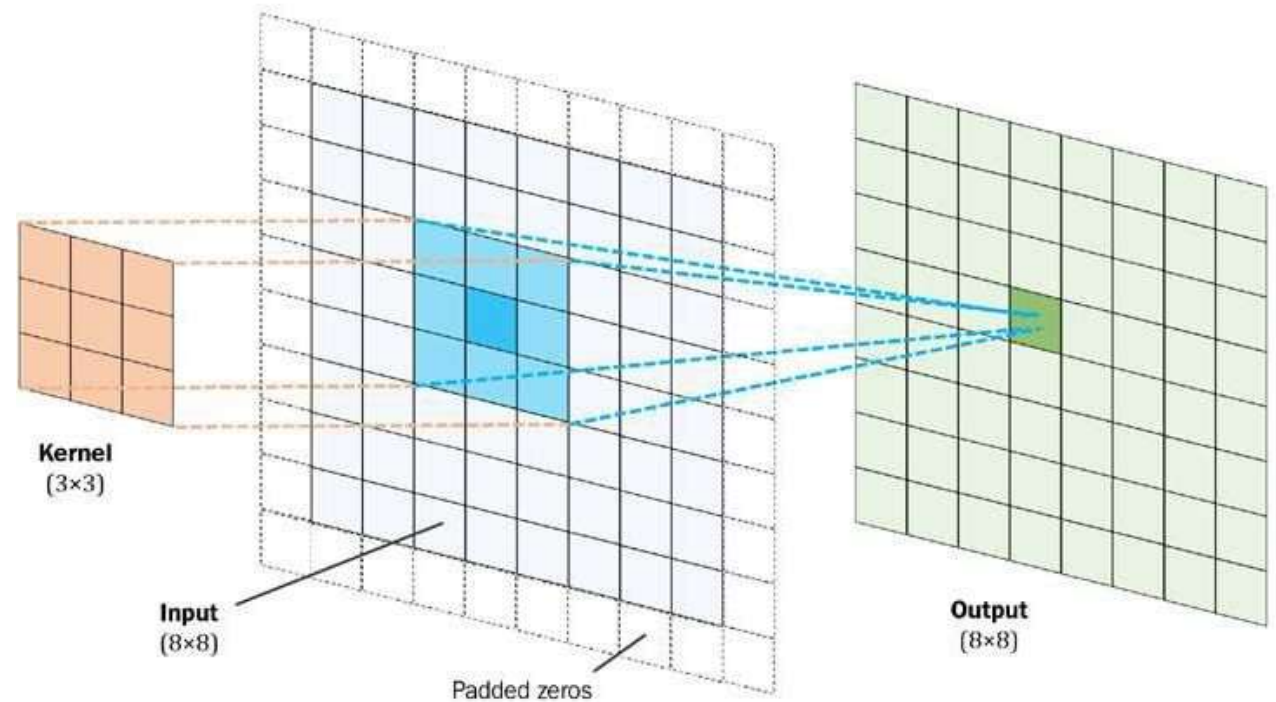
Convolutions are defined by two key parameters:

- Size of the patches extracted from the inputs—These are typically  $3 \times 3$  or  $5 \times 5$ . In the example, they were  $3 \times 3$ , which is a common choice.
- Depth of the output feature map—This is the number of filters computed by the convolution.



# How convolution works (Input depth = 1)

- A convolution **works by sliding these windows** of size  $3 \times 3$  or  $5 \times 5$  over the input feature map, stopping at every possible location, and **extracting the patch** of surrounding features.



## 2D Convolution (Input Depth > 1)

- A convolution **works by sliding these windows** of size  $3 \times 3$  or  $5 \times 5$  over the 3D input feature map, stopping at every possible location, and **extracting the 3D patch** of surrounding features (shape (window\_height, window\_width, input\_depth)).
- Each such 3D patch is **then transformed into a 1D vector of shape** (output\_depth), which is done **via a tensor product** with a **learned** weight matrix, called the convolution kernel— the same kernel is reused across every patch.

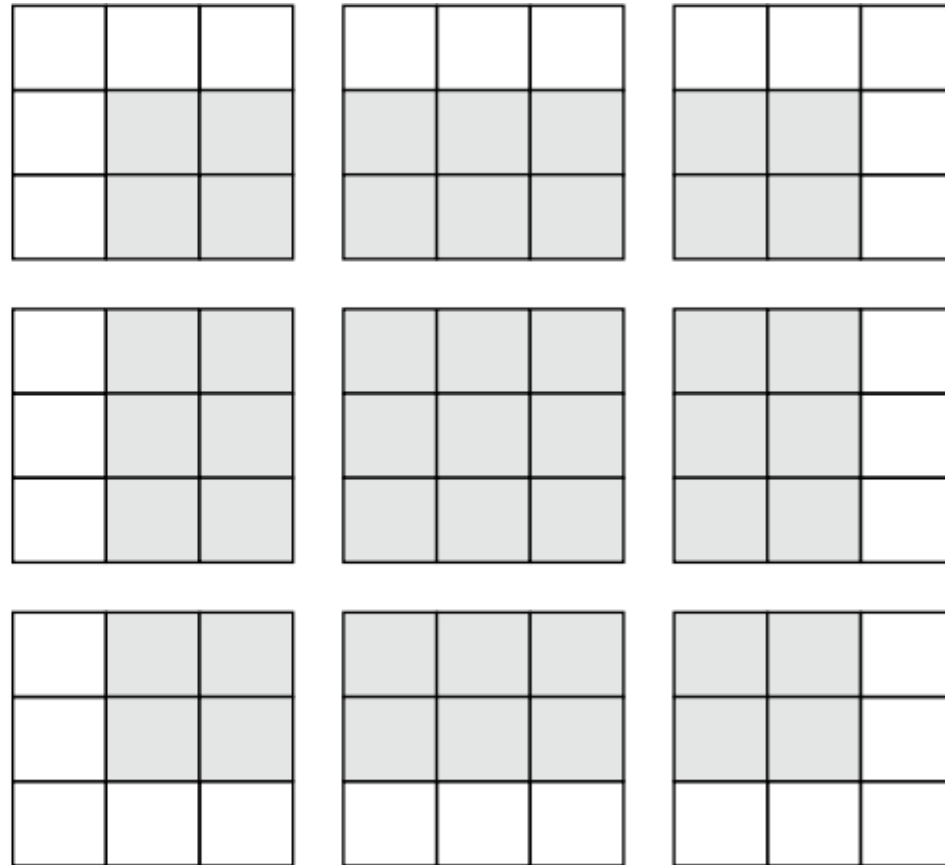
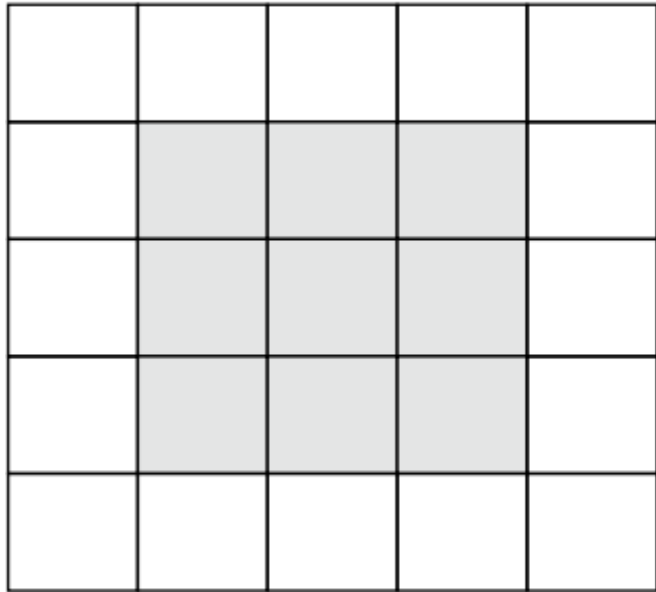
## 2D Convolution (Input Depth $> 1$ )

- All of these vectors (one per patch) are then spatially reassembled into a 3D output map of shape (height, width, output\_depth).
- Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input).

# Understanding Border Effects

- The output width and height may differ from the input width and height for two reasons:
  - Border effects, which can be countered by padding the input feature map
  - The use of strides
- Consider a  $5 \times 5$  feature map (25 tiles total). There are only 9 tiles around which you can center a  $3 \times 3$  window, forming a  $3 \times 3$  grid. Hence, the output feature map will be  $3 \times 3$ .
- It shrinks by exactly two tiles alongside each dimension, in this case. You can see this border effect in action in the earlier example: you start with  $28 \times 28$  inputs, which become  $26 \times 26$  after the first convolution layer.

# Valid locations of $3 \times 3$ patches in a $5 \times 5$ input feature map



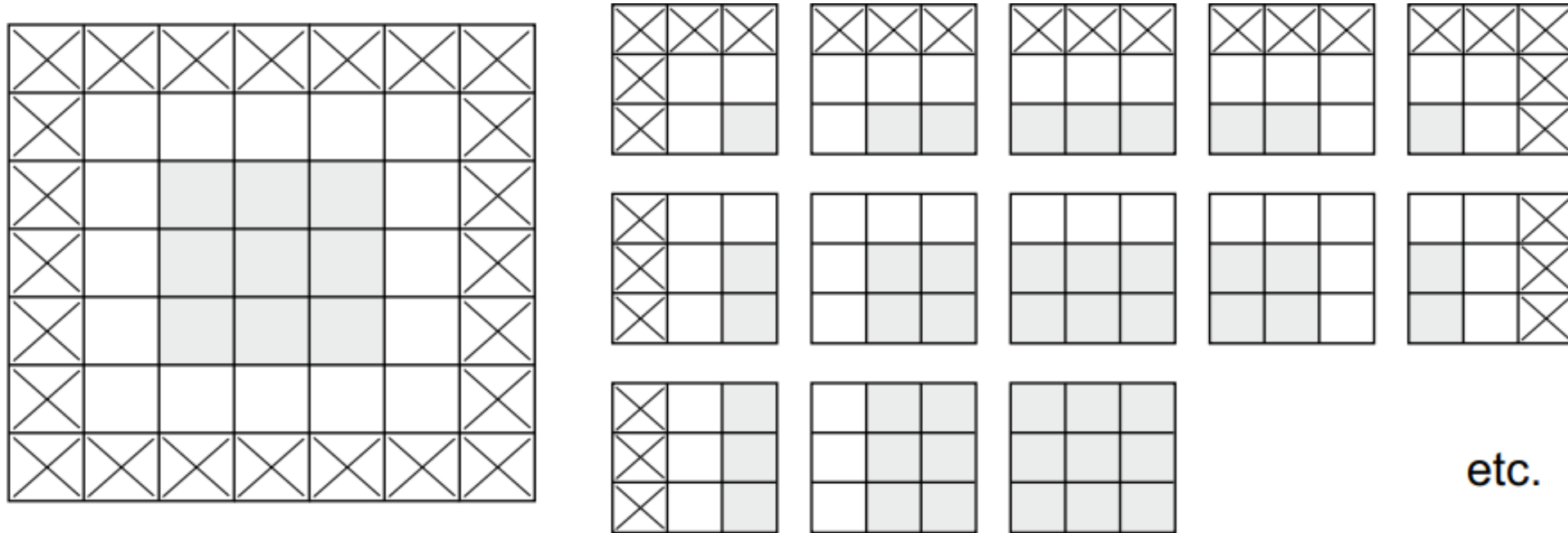
# Understanding Border affects and Padding

- If you want to get an output feature map with the same spatial dimensions as the input, you can use **padding**.
  - Padding consists of adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit center convolution windows around every input tile.
  - For a  $3 \times 3$  window, you add one column on the right, one column on the left, one row at the top, and one row at the bottom.

# Understanding Border affects and Padding

- In Conv2D layers, padding is configurable via the padding argument, which takes two values:
  - "valid", which means no padding (only valid window locations will be used), and
  - "same", which means “pad in such a way as to have an output with the same width and height as the input.” The padding argument defaults to "valid".

Padding a  $5 \times 5$  input in order to be able to extract 25  $3 \times 3$  patches





# Understanding Convolution Strides

- The other factor that can influence output size is the notion of *strides*.
- The **distance between two successive windows** is a parameter of the convolution, called its *stride*, which defaults to 1. It's possible to have *strided convolutions*: convolutions with a stride higher than 1.
  - The patches extracted by a  $3 \times 3$  convolution with stride 2 over a  $5 \times 5$  input (without padding) is shown here.
- Strided convolutions are rarely used in classification models, but they come in handy for some types of models (discussed later)
- In classification models, instead of strides, we tend to **use the *max-pooling*** operation to downsample feature maps.

# $3 \times 3$ convolution patches with $2 \times 2$ strides

	1		2	
	3		4	

	1	

	2	

	3	

	4	

# The max-pooling operation

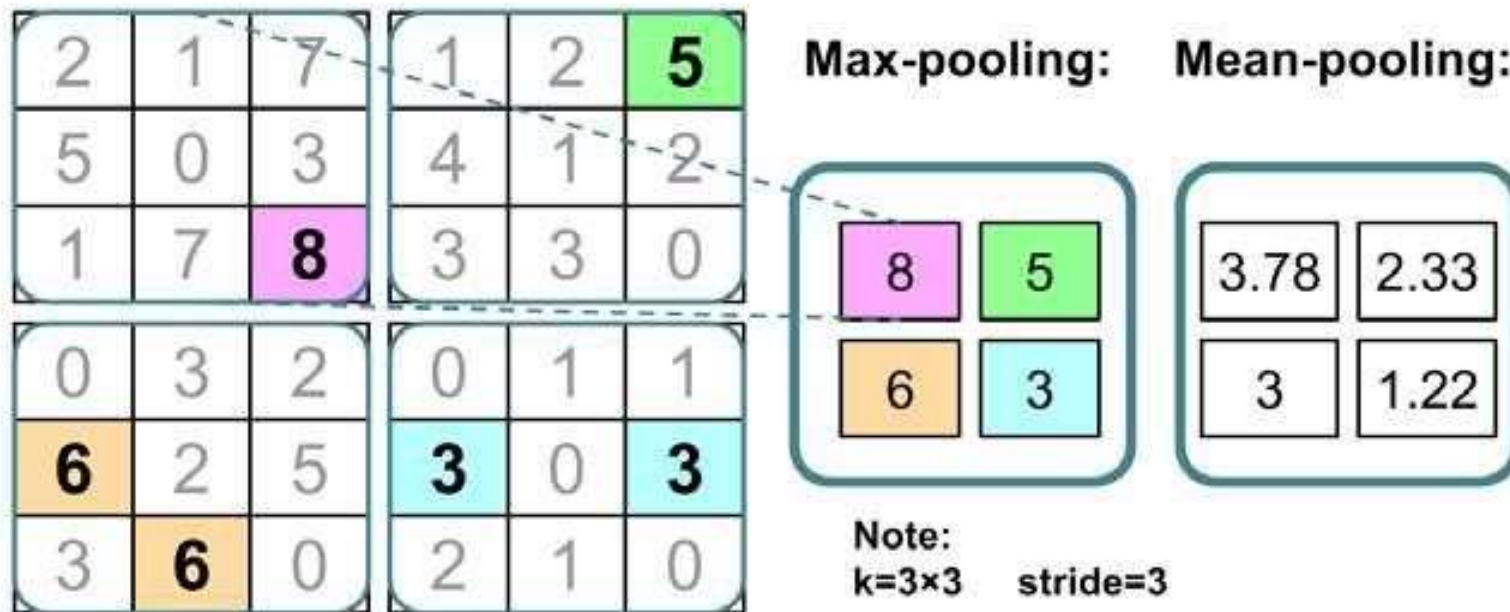
- Max pooling consists of extracting windows from the input feature maps **and outputting the max value** of each channel.
- A big difference from convolution is that max pooling is **usually done with  $2 \times 2$  windows** and stride 2, in order to downsample the feature maps by a factor of 2.
- On the other hand, convolution is **typically done with  $3 \times 3$  windows** and no stride (stride 1).

# The max-pooling operation

- Max pooling isn't the only way to downsampling. Other than strides, you can use **average pooling**, where each local input patch is transformed by taking the average value of each channel over the patch.
- Max pooling is **more informative to look at the maximal presence** of different features than at their average presence.
- The **most reasonable subsampling strategy** is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

# Max/ Mean Pooling

## Pooling ( $P_{3 \times 3}$ )



# A small convnet for digits classification

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

# A small convnet for digits classification

```
>>> model.summary()
```

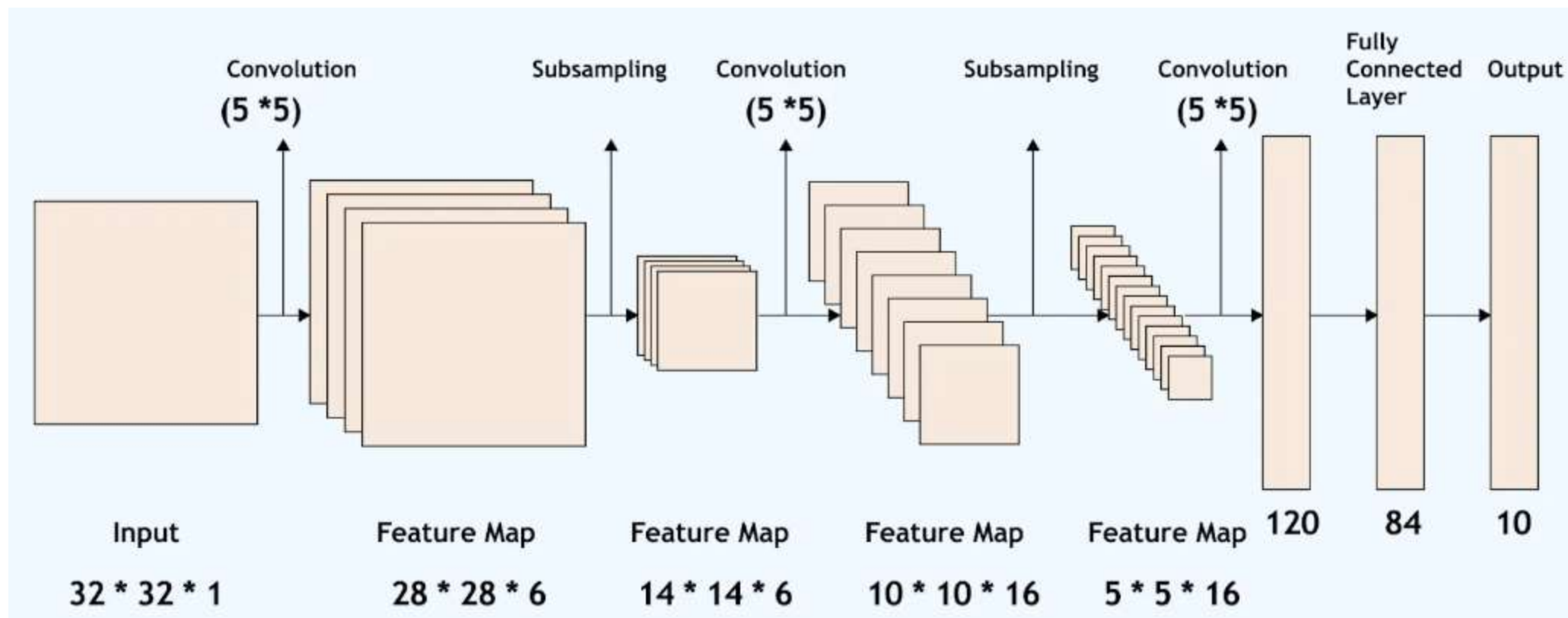
```
Model: "model"
```

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
<hr/>		
conv2d (Conv2D)	(None, 26, 26, 32)	320
<hr/>		
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
<hr/>		
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
<hr/>		
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73856
<hr/>		
flatten (Flatten)	(None, 1152)	0
<hr/>		
dense (Dense)	(None, 10)	11530
=====		

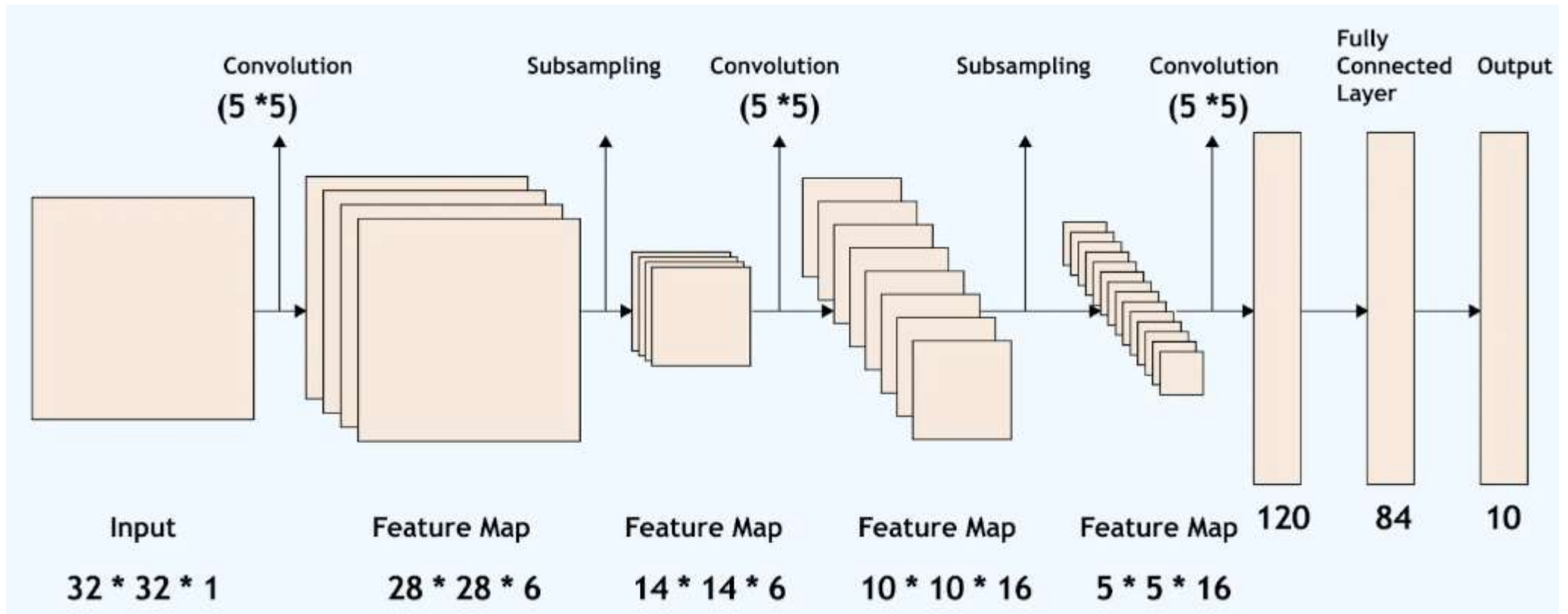
```
Total params: 104,202
```

```
Trainable params: 104,202
```

```
Non-trainable params: 0
```

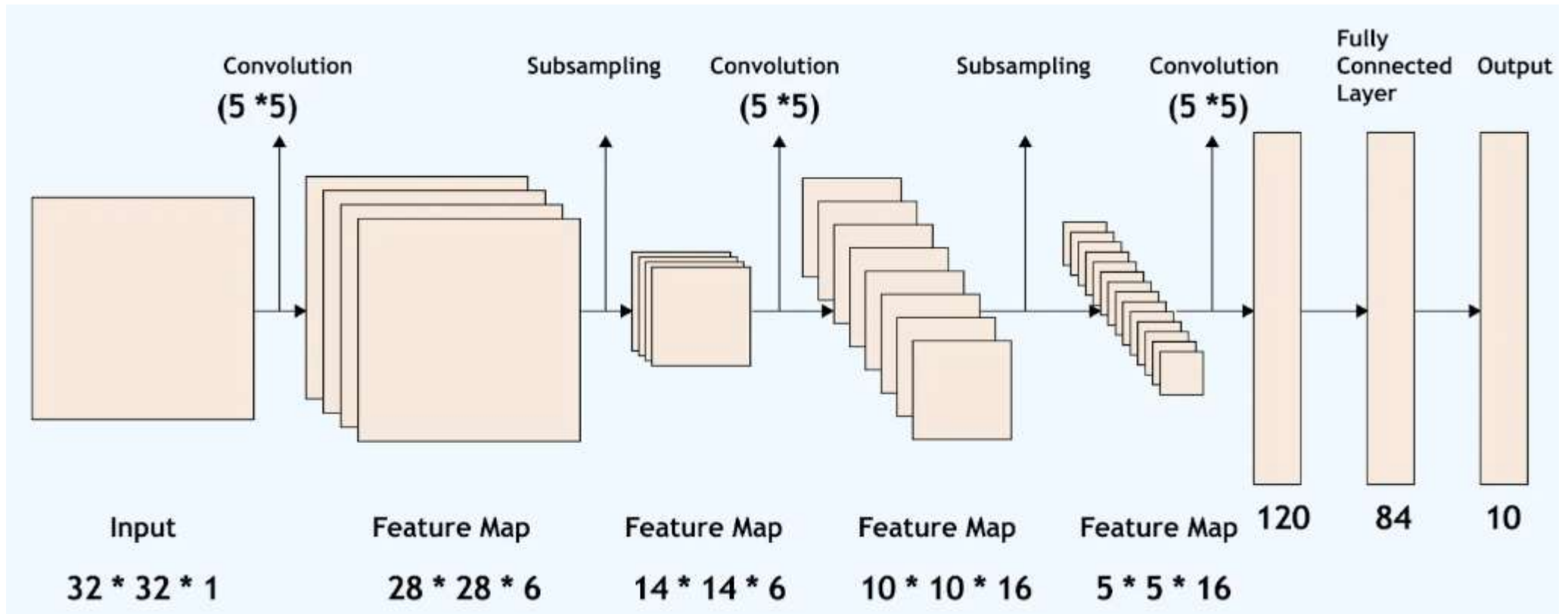






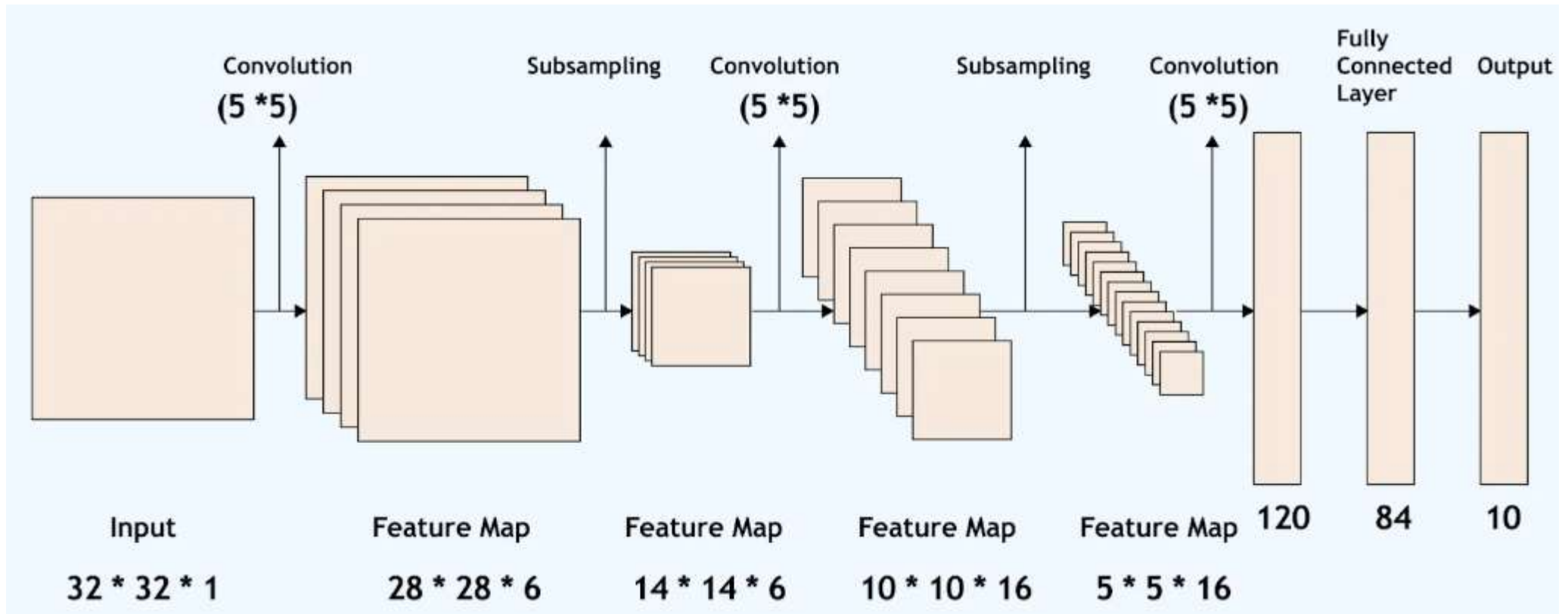
ReLU

$$\begin{aligned} &5 * 5 * 6 + 6 \\ &= 150 + 6 \\ &= 156 \end{aligned}$$



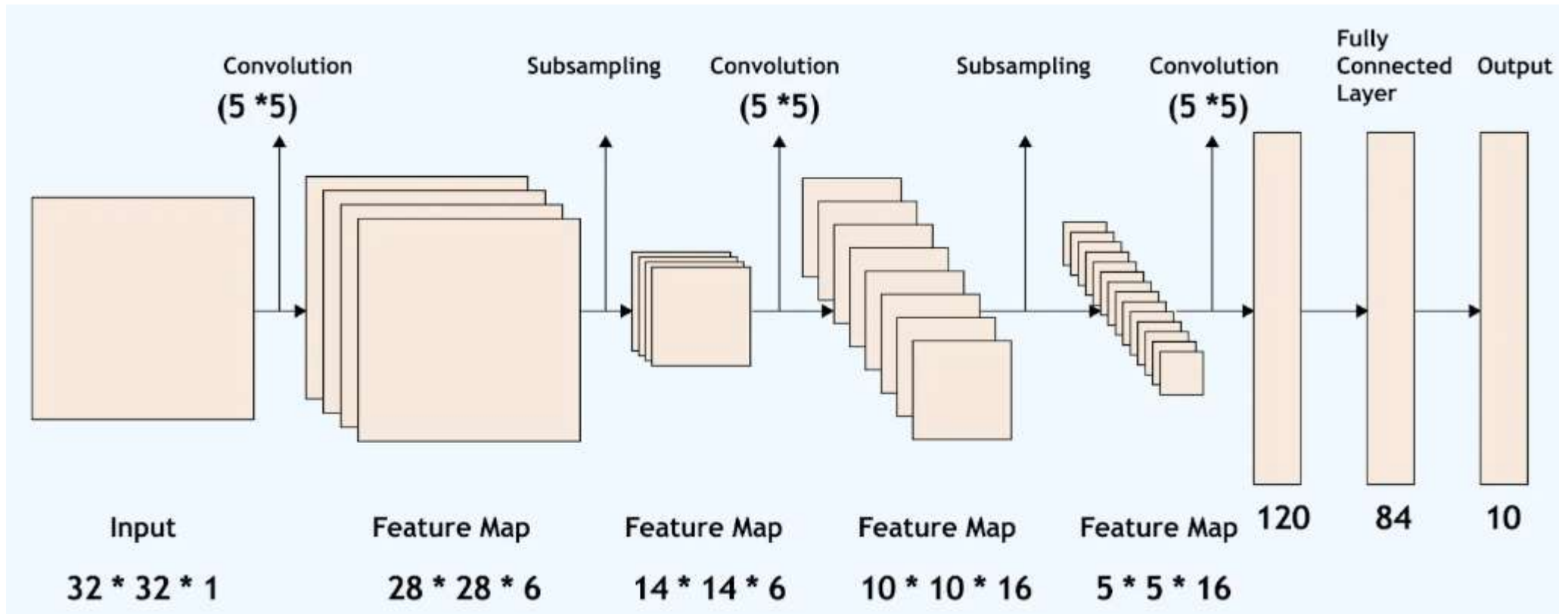
ReLU

$$\begin{aligned} &5 * 5 * 6 * 16 + 16 \\ &= 150 * 16 + 16 \\ &= 2400 + 16 \\ &= 2416 \end{aligned}$$



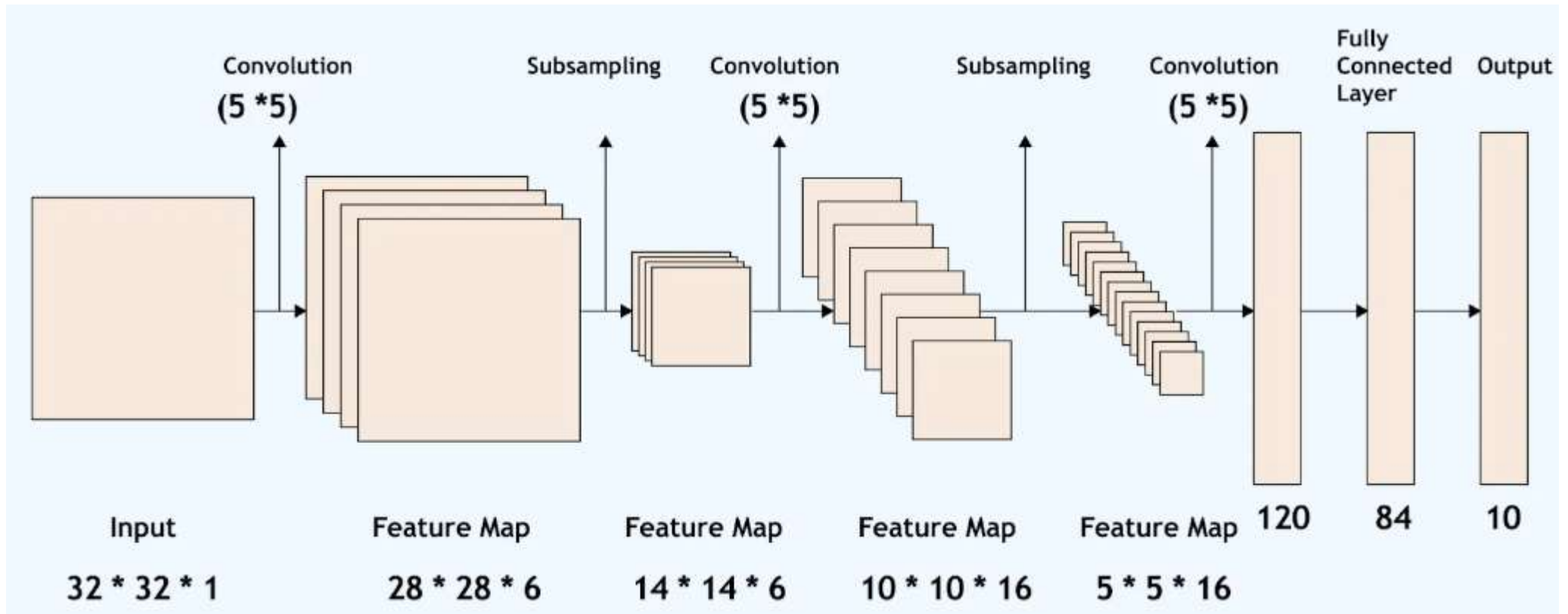
ReLU

$$\begin{aligned} &5 * 5 * 16 \\ &= 25 * 16 \\ &= 400 \end{aligned}$$



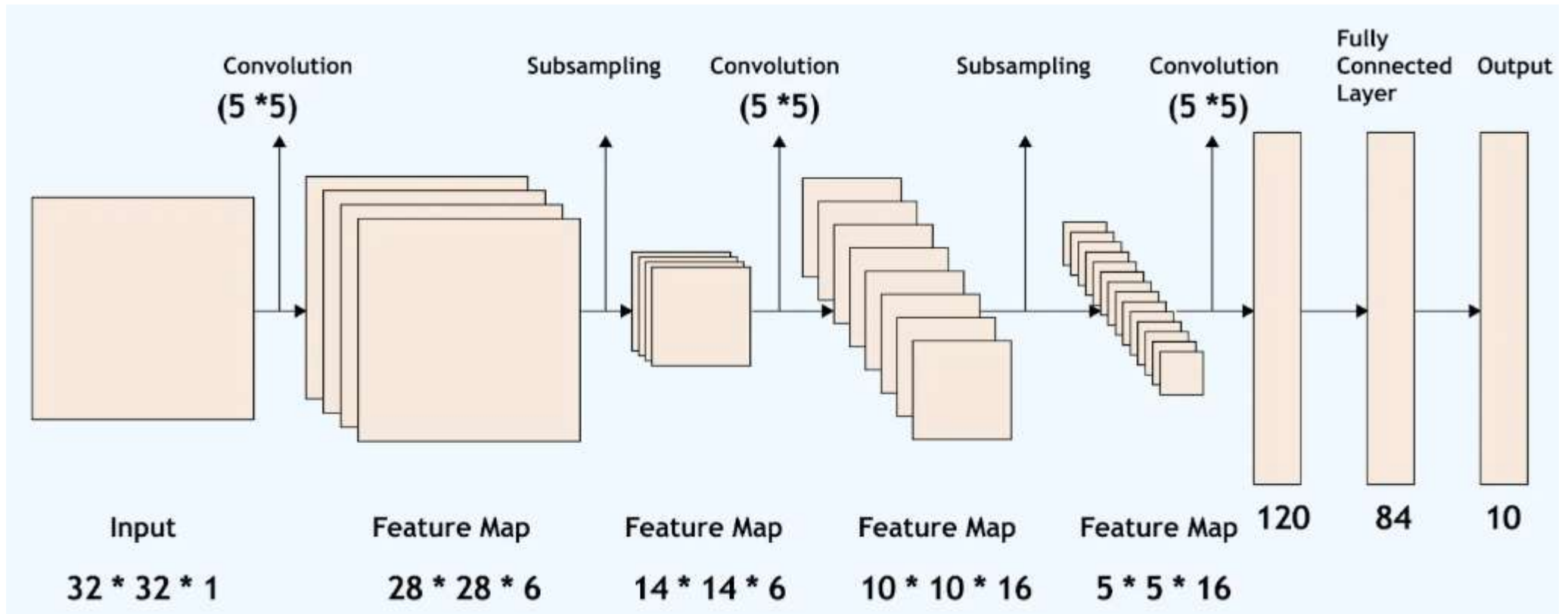
ReLU

$$400 * 120 + 120 \\ = 48120$$



ReLU

$$\begin{aligned} &48120 * 84 + 84 \\ &= 4042080 + 84 \\ &= 4042164 \end{aligned}$$



Softmax

$$\begin{aligned}
 &4042164 * 10 + 10 \\
 &= 40421640 + 10 \\
 &= 40421650
 \end{aligned}$$

# Class Test

- CNN architecture:
  - Input: 8x8 gray scale image
  - Two conv layers:
    - 1<sup>st</sup>: a 2x2 kernel, 'valid' padding, stride 2
    - 2<sup>nd</sup>: two 2x2 kernels, 'same' padding, no stride
  - Then, max pooling layer: 2x2 filters, stride 2
  - Two hidden dense layers:
    - 1<sup>st</sup>: 10 units
    - 2<sup>nd</sup>: 10 units
  - Three output classes.
- What activation function(s) are used?
- Number of learning parameters in each layer of the entire network.

# Data Augmentation



# Using data augmentation

- Data augmentation takes the approach of generating more training data from existing training samples by *augmenting* the samples via a number of random transformations that yield believable-looking images.
- The goal is that, at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data so it can generalize better.
- In Keras, this can be done by adding a number of *data augmentation layers* at the start of your model (before the rescaling layer).

# Define a data augmentation stage to add to an image model

```
data_augmentation = keras.Sequential(  
    [  
        layers.RandomFlip("horizontal"),  
        layers.RandomRotation(0.1),  
        layers.RandomZoom(0.2),  
    ]  
)
```

- `RandomFlip("horizontal")`—Applies horizontal flipping to a random 50% of the images that go through it
- `RandomRotation(0.1)`—Rotates the input images by a random value in the range  $[-10\%, +10\%]$  (these are fractions of a full circle—in degrees, the range would be  $[-36 \text{ degrees}, +36 \text{ degrees}]$ )
- `RandomZoom(0.2)`—Zooms in or out of the image by a random factor in the range  $[-20\%, +20\%]$

Generating variations of a dog  
via random data augmentation



Pretrained Model

# Leveraging a pretrained model

- A common and highly effective approach to deep learning on small image datasets is to use a pretrained model.
- A *pretrained model* is a model that was previously trained on a large dataset, typically on a large-scale image-classification task
- If this original dataset is **large enough and general enough**, the spatial hierarchy of features learned by the pretrained model can effectively act as a generic model of the visual world.

# Leveraging a pretrained model

- A large convnet trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes) containing many animal classes, including different species of cats and dogs, and you can thus expect it to perform well on the dogs-versus-cats classification problem.
- There are two ways to use a pretrained model: *feature extraction* and *fine-tuning*.

# Feature extraction with a pretrained model

- Feature extraction consists of **using the representations** learned by a previously trained model to extract interesting features from new samples. These features are then **run through a new classifier**, which is trained from scratch.
- Previous convnet start with a series of pooling and convolution layers, and they end with a densely connected classifier. The first part is called the *convolutional base* of the model.
- In case of convnets, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output.

# Fine-tuning a pretrained model

- Fine-tuning consists of **unfreezing a few of the top layers of a frozen model** base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers.
- This is called *fine-tuning* because it slightly adjusts the more abstract representations of the model being reused in order to make them more relevant for the problem at hand.



# Fine-tuning a pretrained model...

The steps for fine-tuning a network are as follows:

- Add our custom network on top of an already-trained base network.
- Freeze the base network.
- Train the part we added.
- Unfreeze some layers in the base network. (Note that you should not unfreeze “batch normalization” layers.)
- Jointly train both these layers and the part we added.