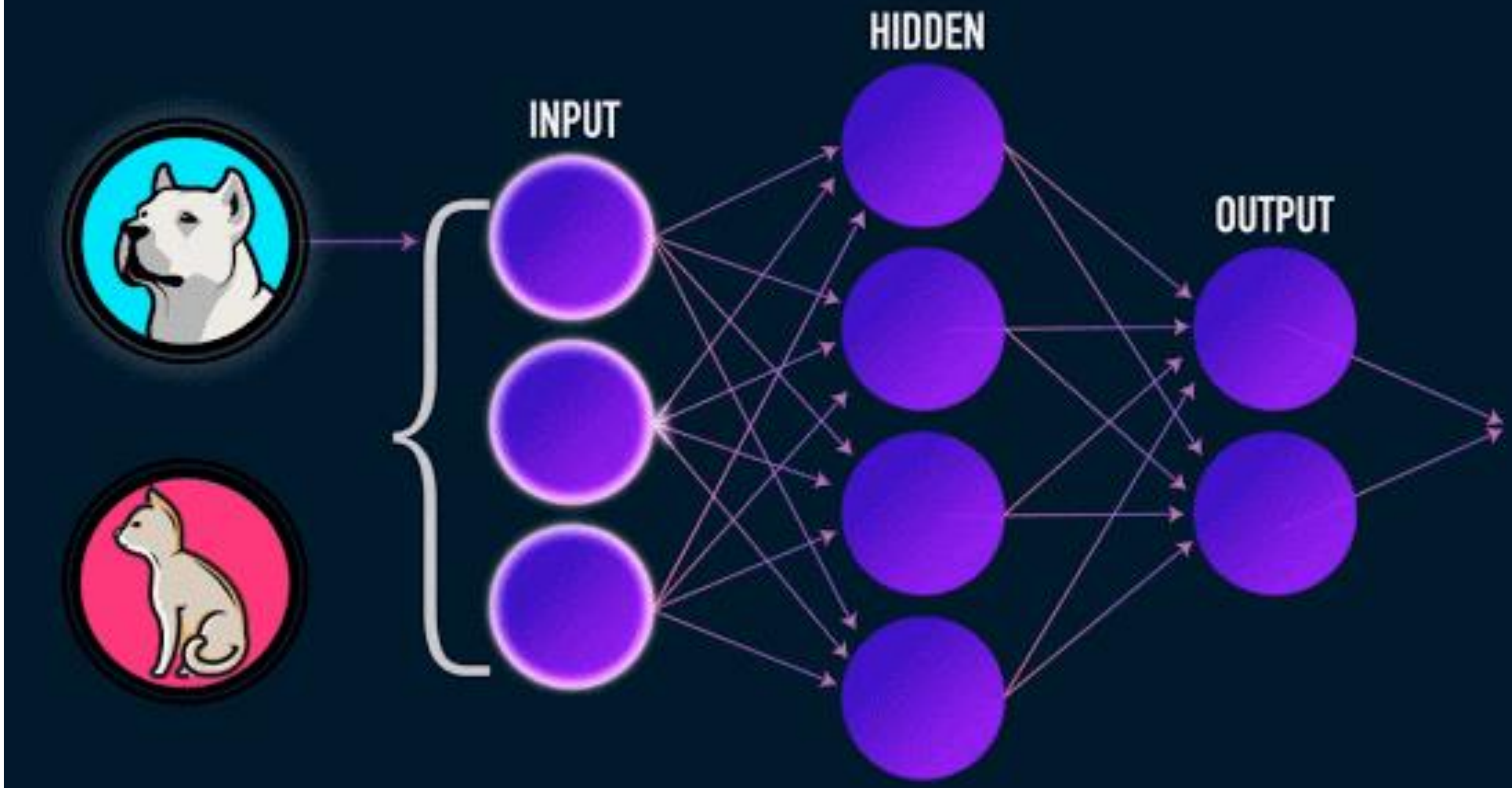
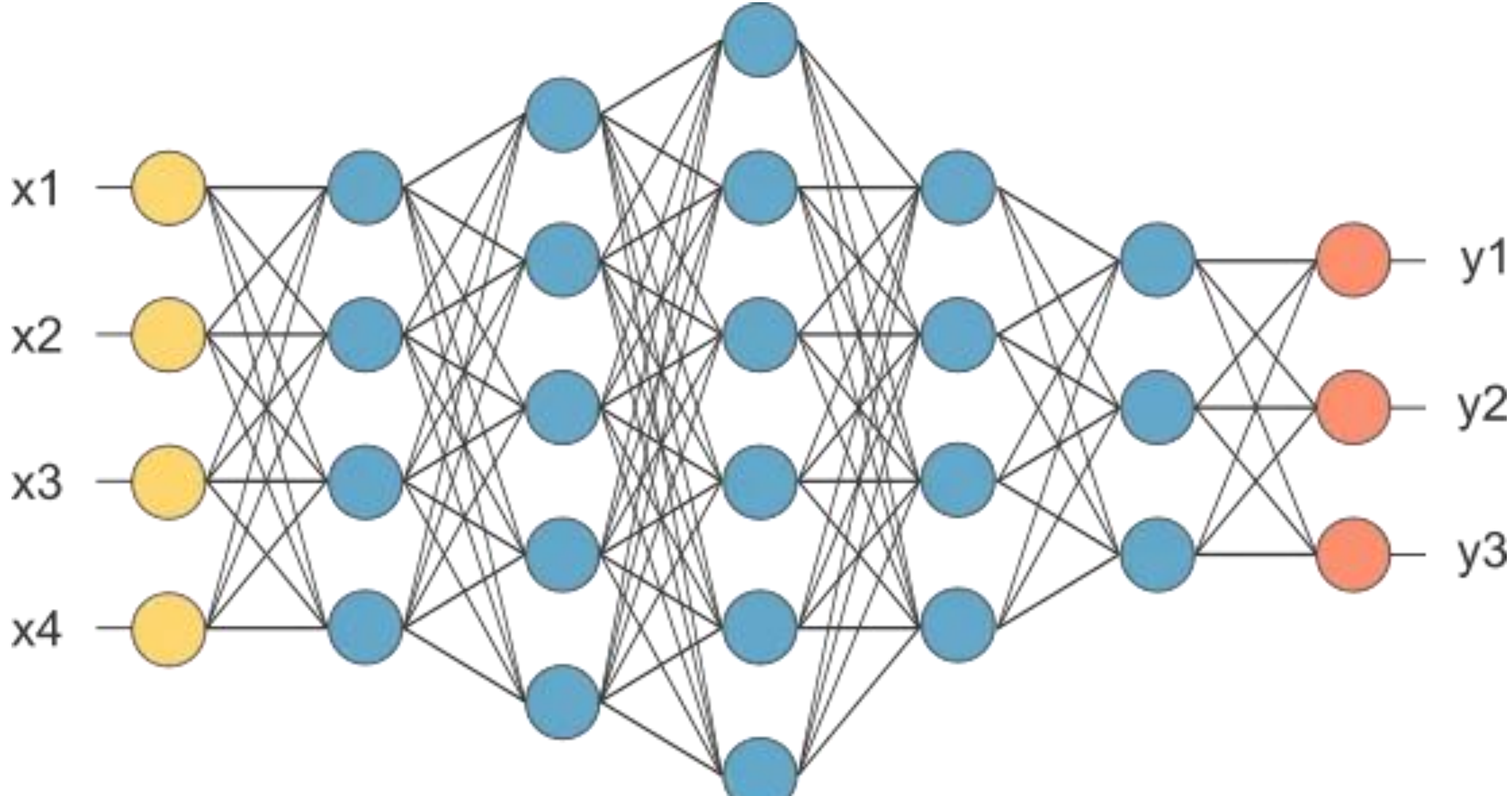
CNN


HIDDEN
INPUT
OUTPUT

Why Conv layer? What is the problem with Dense layers?



Convolution

5	2	3	1	2	4
2	4	1	0	3	1
5	1	0	2	8	3
0	2	1	5	2	4
2	7	0	0	2	1
1	3	2	8	7	0

Convolution

- *Kernel* = grid of weights
- Kernel is “applied” to the image
- Traditionally used in image processing

1	2	-1
0	1	2
-2	1	0

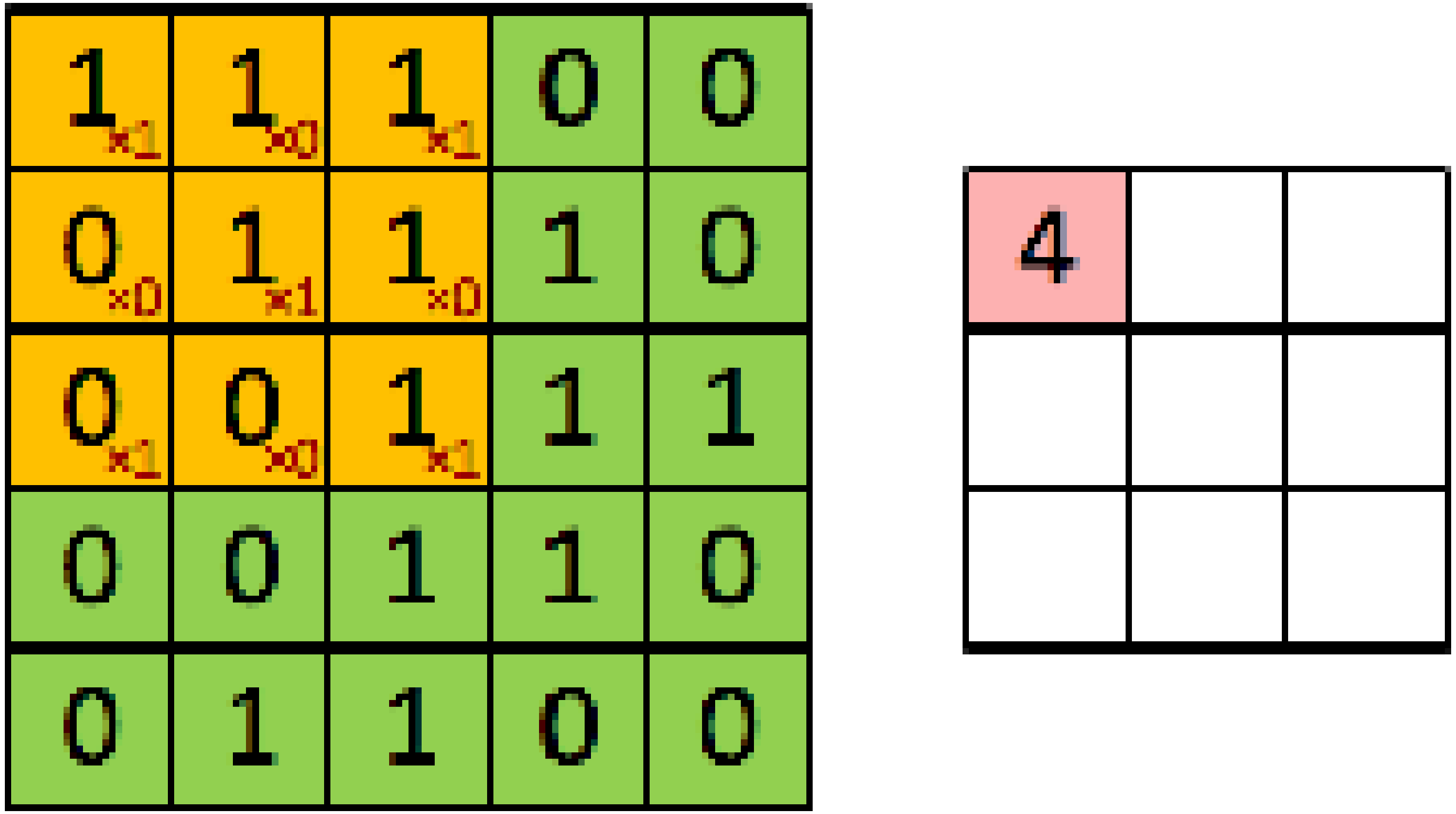
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

5 x 5 – Image Matrix

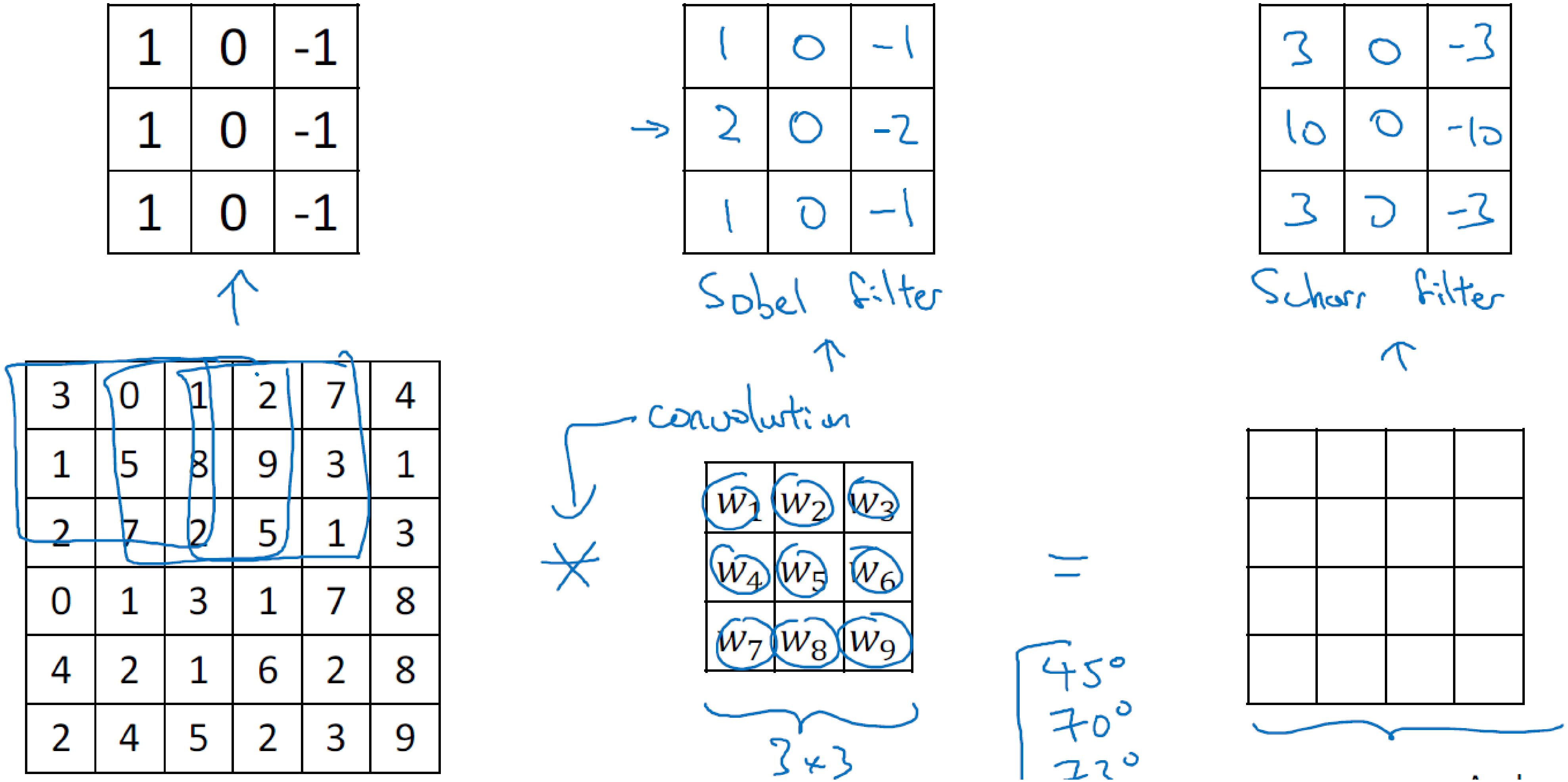
*

1	0	1
0	1	0
1	0	1

3 x 3 – Filter Matrix



Learning to detect edges



1	0	-1
1	0	-1
1	0	-1

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

convolution

$*$

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

3×3

=

45°
70°
72°

Kernels

- Feature detectors
- Kernels are learned

Oblique line detector

1	0	0
0	1	0
0	0	1

Vertical line detector

0	1	0
0	1	0
0	1	0

Architectural decisions for convolution

- Grid size
- Stride
- Depth
- Number of kernels

Grid size

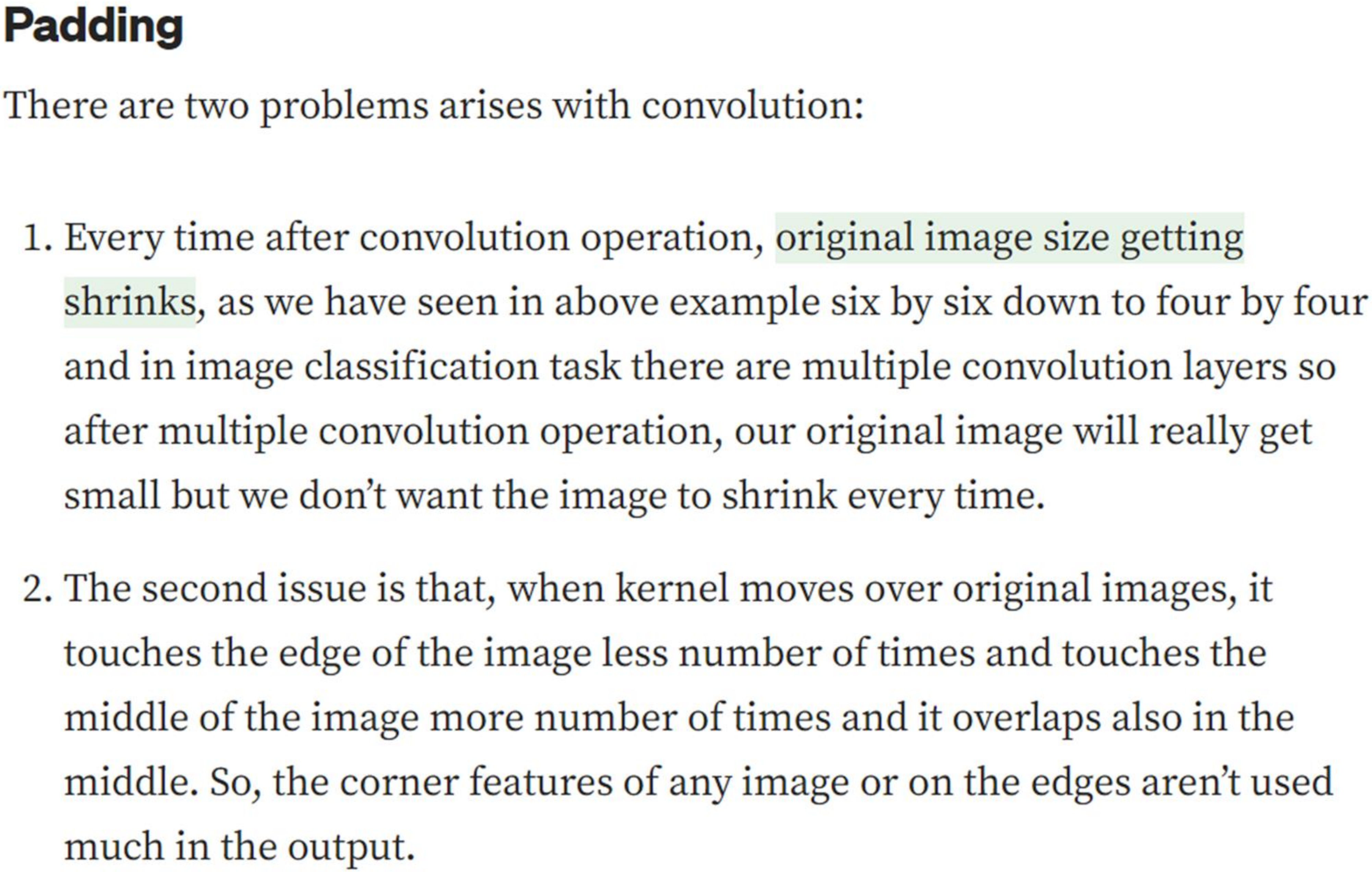
- # of pixels for height/width
- Odd numbers

5 by 5

1	2	9	8	7
1	6	5	0	0
2	2	3	1	0
1	1	-3	0	-1
1	-2	2	2	3

3 by 3

1	2	9
1	6	5
2	2	3



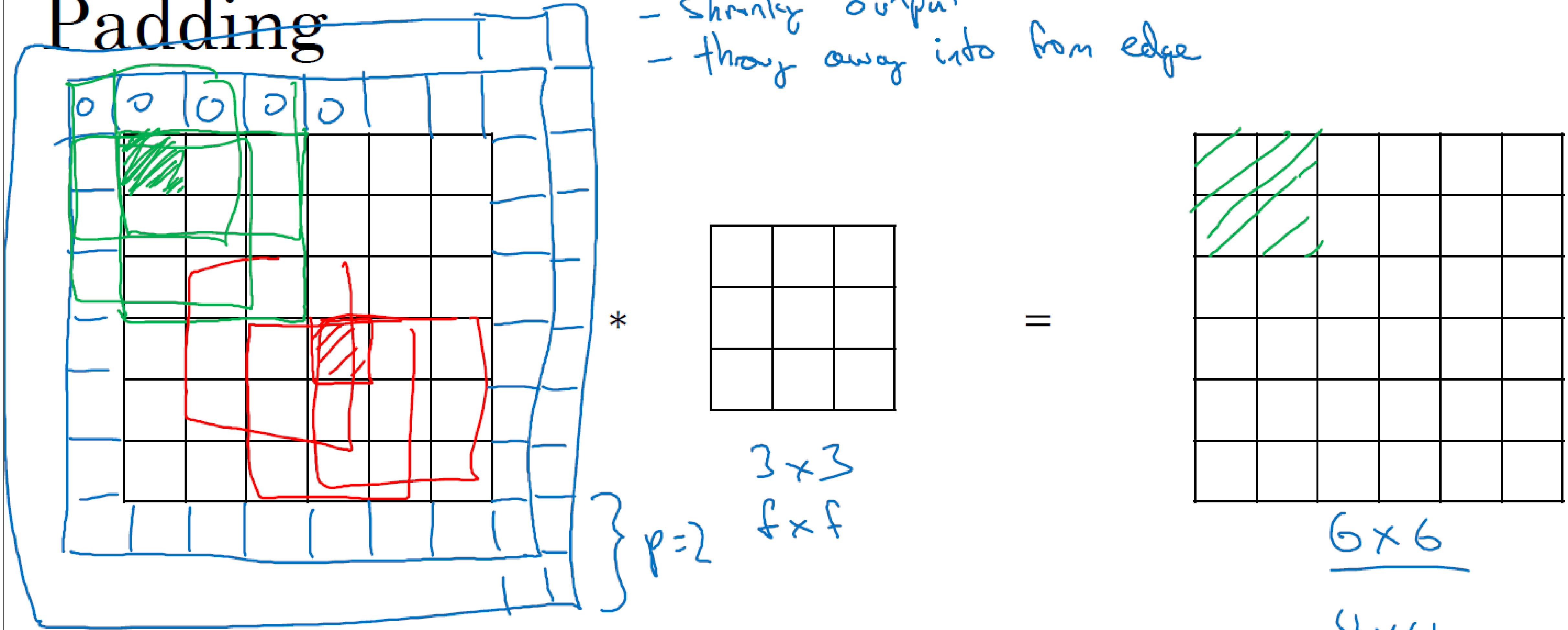
Padding

There are two problems arises with convolution:

1. Every time after convolution operation, original image size getting shrinks, as we have seen in above example six by six down to four by four and in image classification task there are multiple convolution layers so after multiple convolution operation, our original image will really get small but we don’t want the image to shrink every time.
2. The second issue is that, when kernel moves over original images, it touches the edge of the image less number of times and touches the middle of the image more number of times and it overlaps also in the middle. So, the corner features of any image or on the edges aren’t used much in the output.

Padding

- shrinky output
- throw away info from edge



$6 \times 6 \rightarrow 8 \times 8$

$n \times n$

$n-f+1 \times n-f+1$ → ~~4×4~~

$6-3+1=4$

$p = \text{padding} = 1$

$n+2p-f+1 \times n+2p-f+1$

$6+2-3+1 \times \_\_\_ = 6 \times 6$

Valid and Same convolutions

no padding

"Valid": $n \times n \ * \ f \times f \longrightarrow \underline{n-f+1} \times n-f+1$

$6 \times 6 \ * \ 3 \times 3 \longrightarrow 4 \times 4$

"Same": Pad so that output size is the same as the input size.

$n+2p-f+1 \times n+2p-f+1$

$\cancel{n}+2p-f+1 = \cancel{n} \Rightarrow \boxed{p = \frac{f-1}{2}}$

$3 \times 3 \quad p = \frac{3-1}{2} = 1 \quad \Big| \quad 5 \times 5 \quad p = 2$

$f=5$

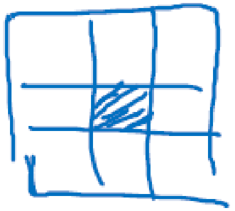
f is usually odd

1×1

3×3

5×5

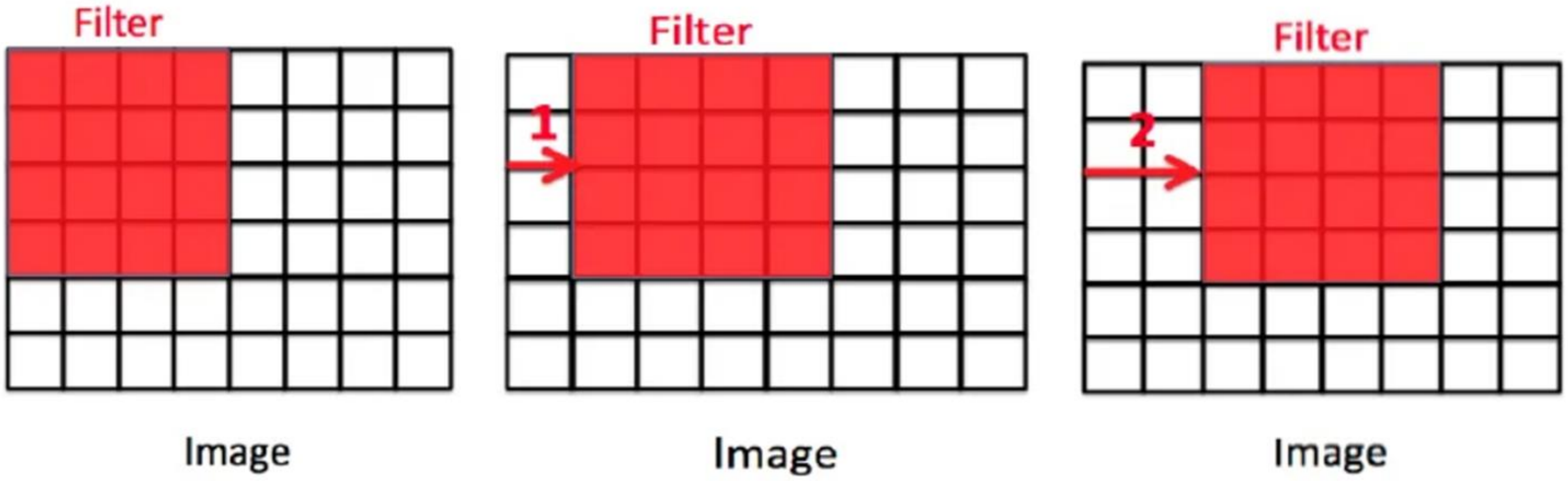
7×7



Stride

- Step size used for sliding kernel on image
- Indicated in pixels

Stride



left image: stride =0, middle image: stride = 1, right image: stride =2

Summary of convolutions

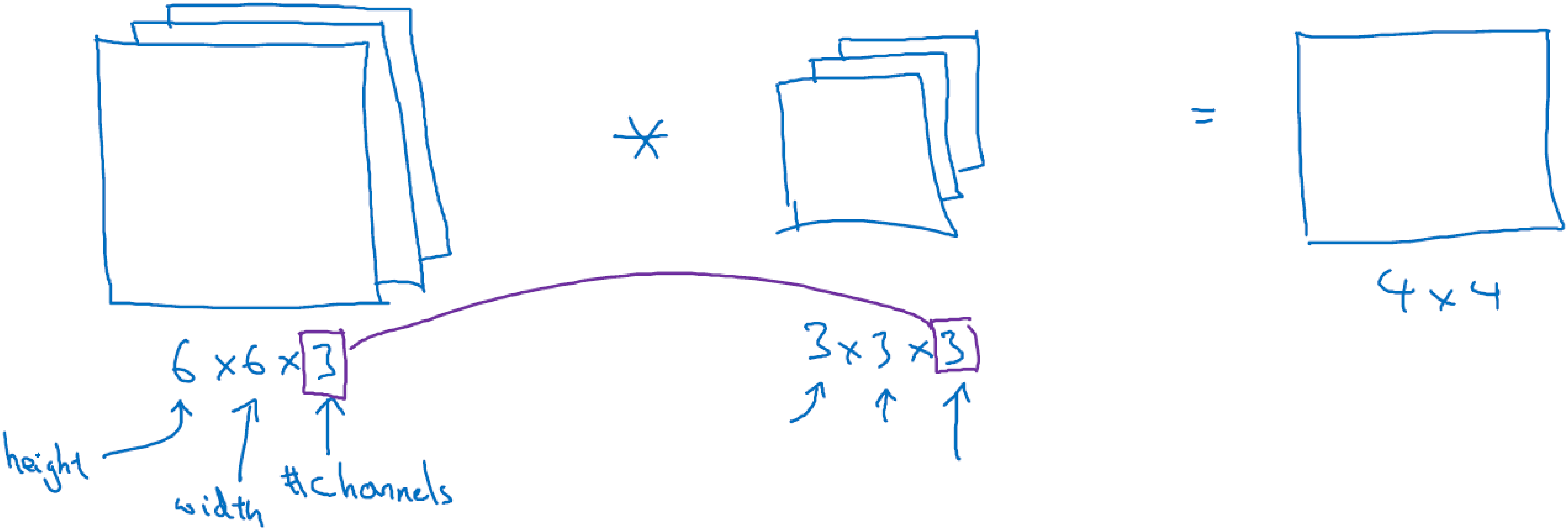
$n \times n$ image $\qquad f \times f$ filter

padding p $\qquad$ stride s

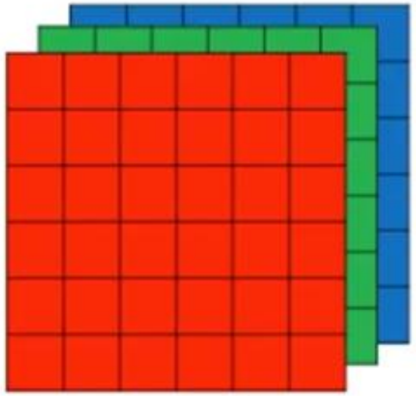
Output size:

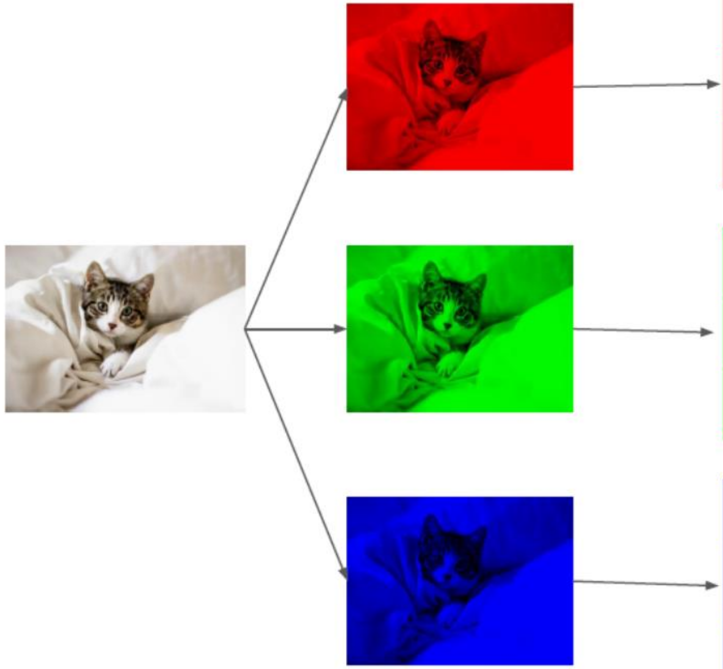
$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \quad \times \quad \left\lfloor \underbrace{\frac{n+2p-f}{s}}_{} + 1 \right\rfloor$$

Convolutions on RGB images



Depth



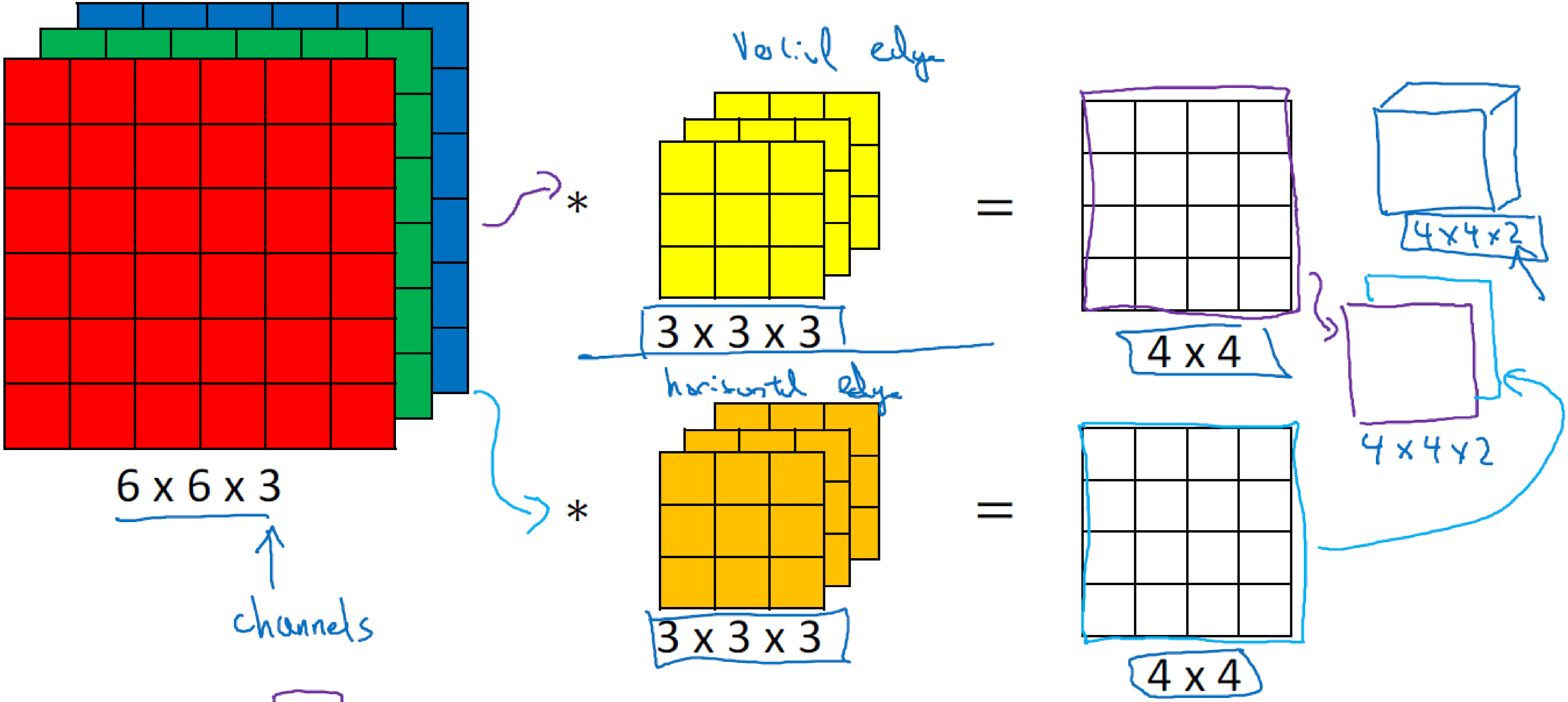


1	2	-1
0	1	2
-2	1	0

1	2	-1
0	1	2
-2	1	0

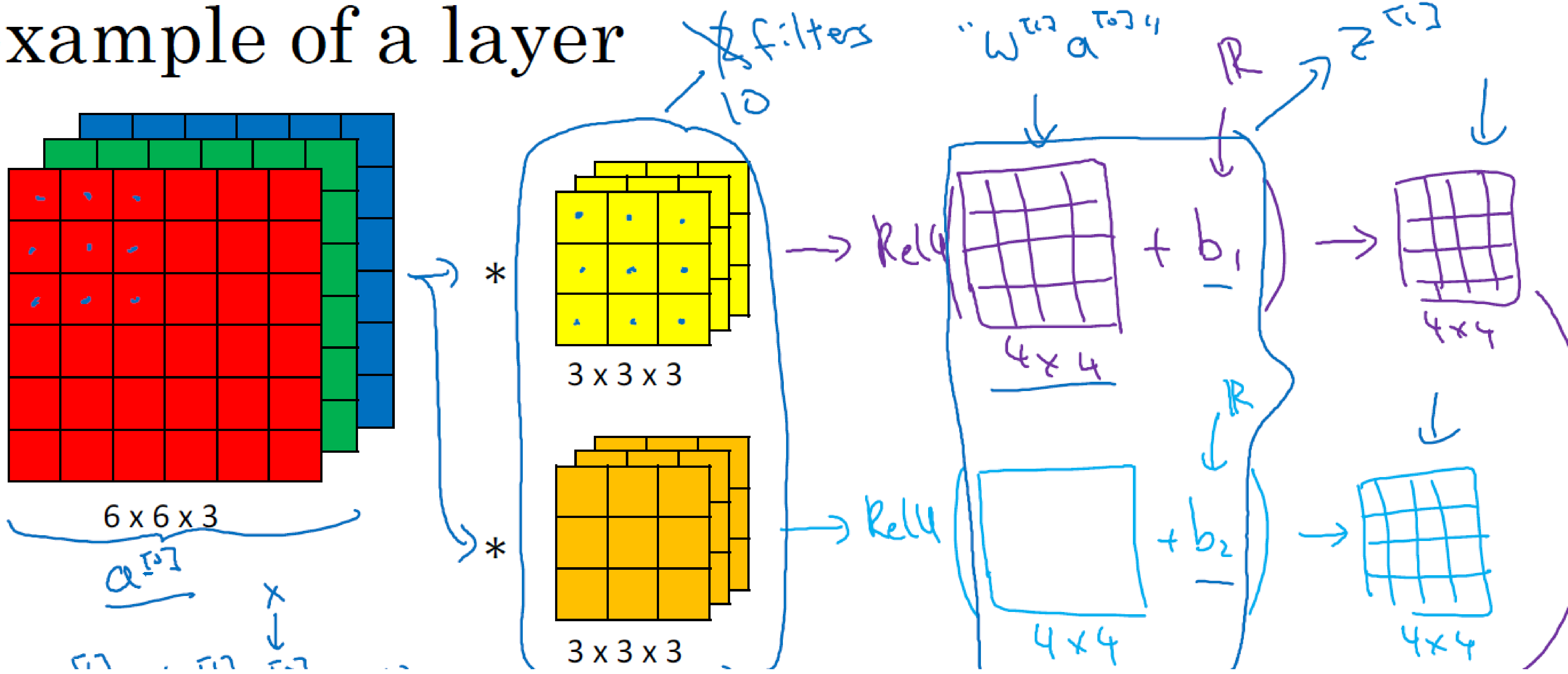
1	2	-1
0	1	2
-2	1	0

Multiple filters



Summary: $$\underset{6\times 6\times 3}{n\times n\times \boxed{n_c}} \; * \; \underset{3\times 3\times 3}{f\times f\times \boxed{n_c}} \longrightarrow \underset{4}{\underline{n-f+1}} \times \underset{4}{\underline{n-f+1}} \times \underset{2}{n_c'}$$ ← #filters

Example of a layer



Number of parameters in one layer

If you have 10 filters that are 3 x 3 x 3 in one layer of a neural network, how many parameters does that layer have?

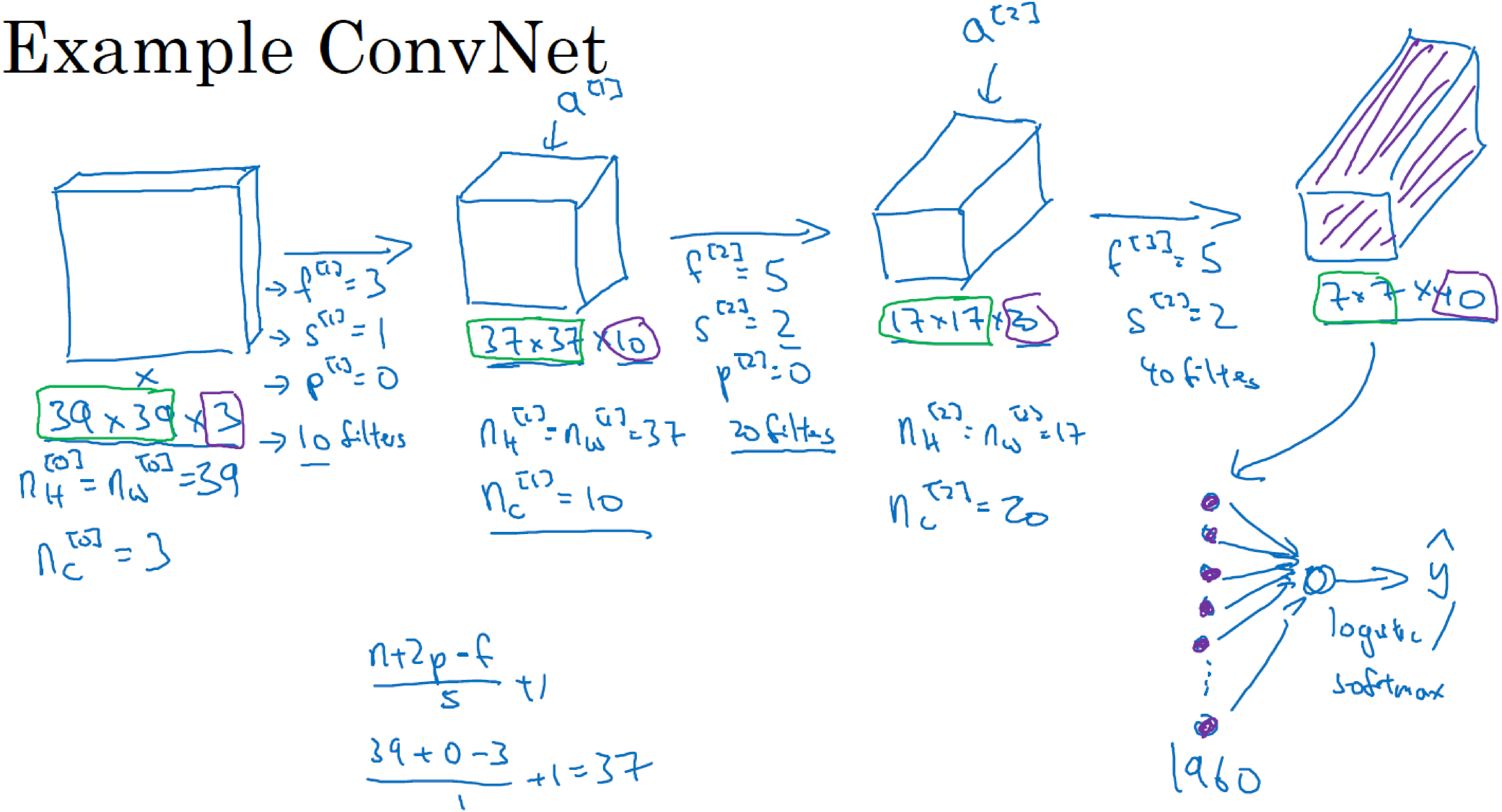
General rule of parameter computation

- Filter size, stride, padding, and number of filters are given

- Input shape is provided

- #parameters = ???

Example ConvNet



$$\frac{n+2p-f}{s}+1$$

$$\frac{39+0-3}{1}+1=37$$

Types of layer in a convolutional network:

- Convolution (CONV) ←
- Pooling (POOL) ←
- Fully connected (FC) ←

Pooling

- Downsample the image
- Overlaying grid on image
- Max/average pooling
- No parameters

Why pooling?

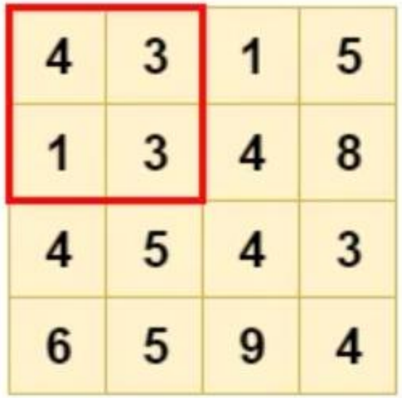
- Reduce the size of the representation
- Speed up computation

Pooling settings

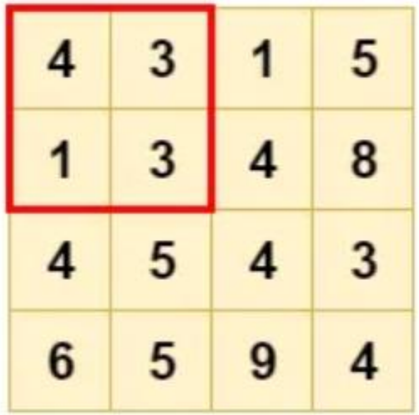
- Grid size
- Stride
- Type (e.g., max, average)

Max Pooling

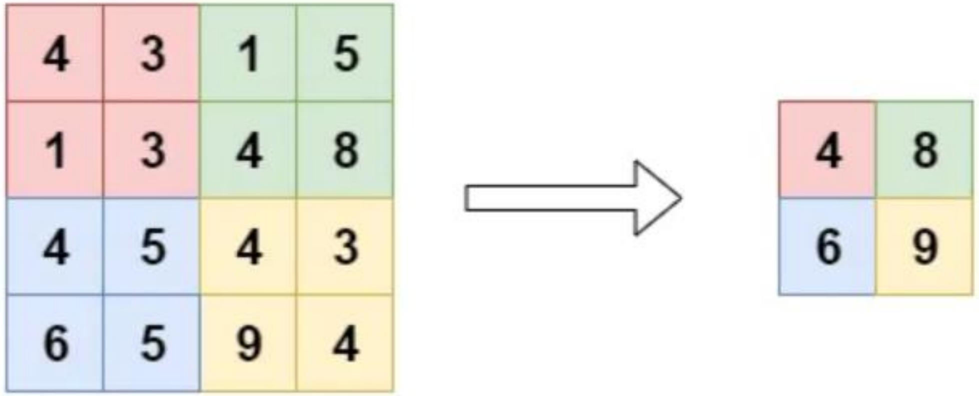
Average Pooling

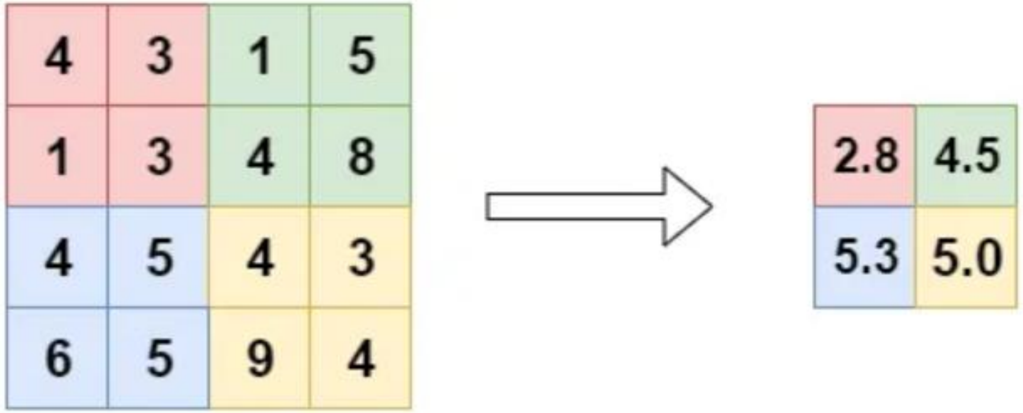


Max([4, 3, 1, 3]) = 4



Avg([4, 3, 1, 3]) = 2.75





Summary of pooling

Hyperparameters:

- f : filter size
- s : stride
- Max or average pooling

$f=2, s=2$

$f=3, s=2$

No parameters to learn!

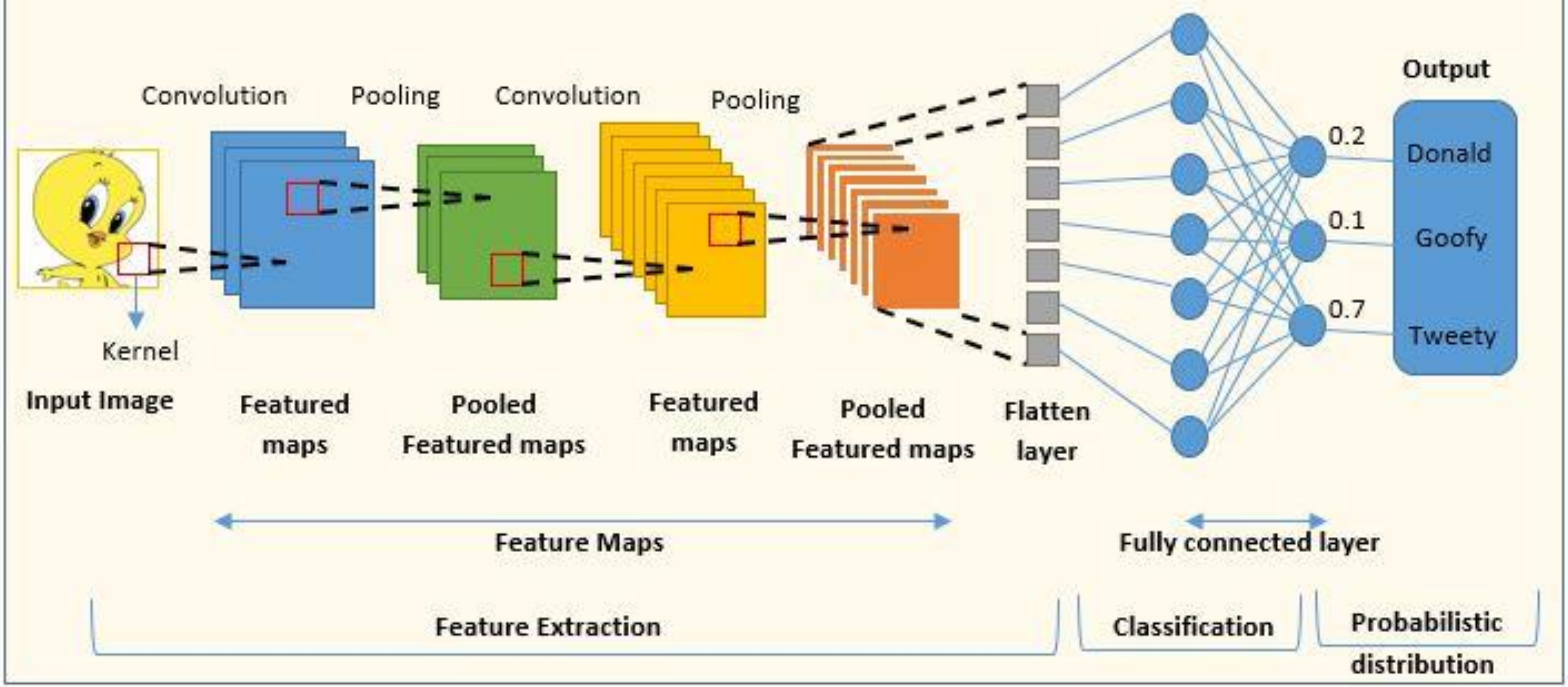
$$n_H \times n_W \times \underline{n_C}$$

$$\downarrow$$

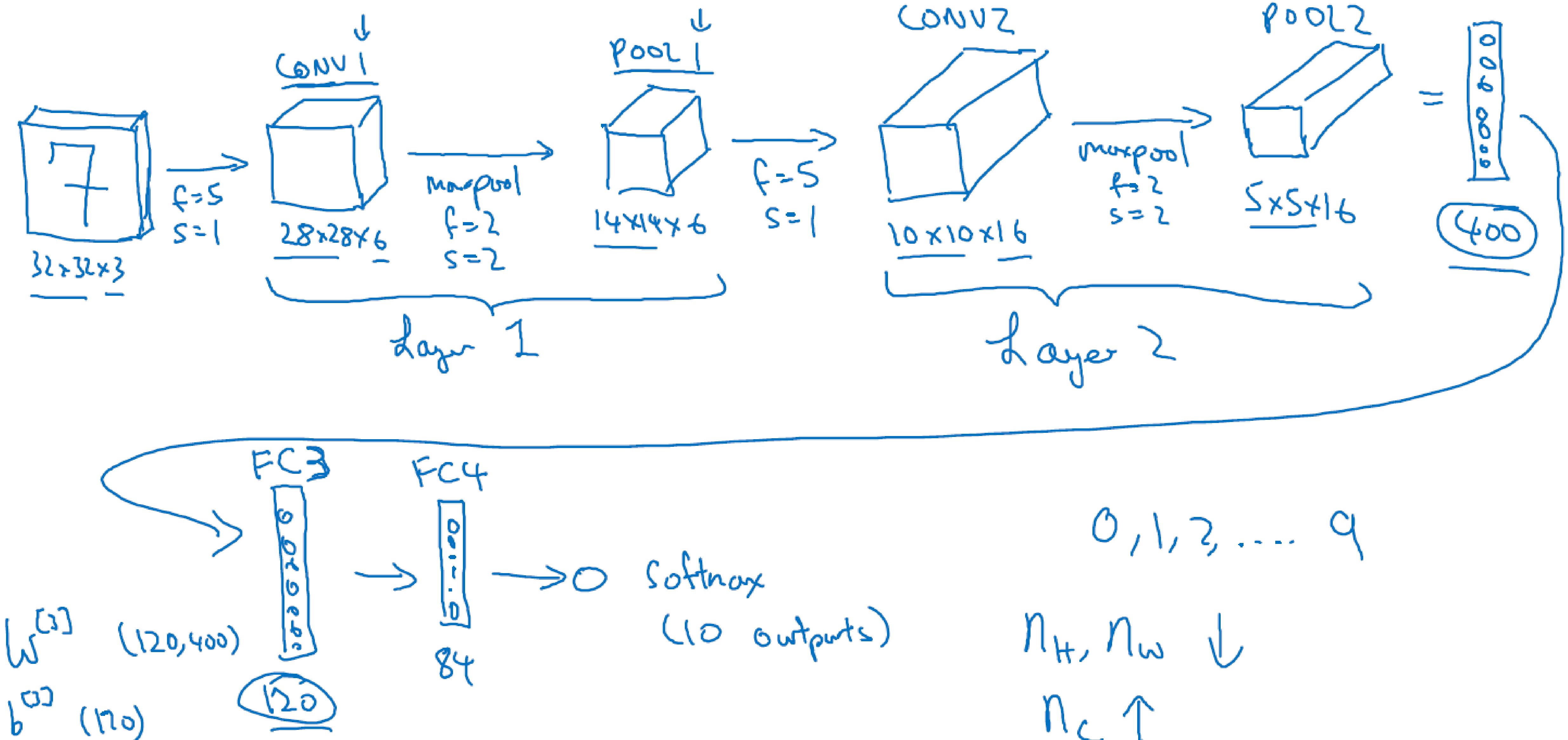
$$\left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor$$

$$\times \underline{n_C}$$

A Typical Convolutional Neural Network (CNN)



Neural network example (LeNet-5)



CONV - POOL - CONV - POOL - FC - FC - FC - SOFTMAX

Why convolutions

translation invariance

Parameter sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

→ **Sparsity of connections:** In each layer, each output value depends only on a small number of inputs.

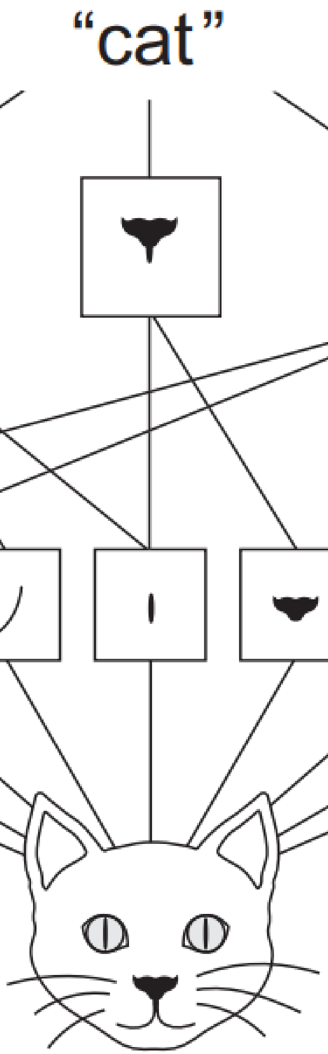
Advantage of convnet

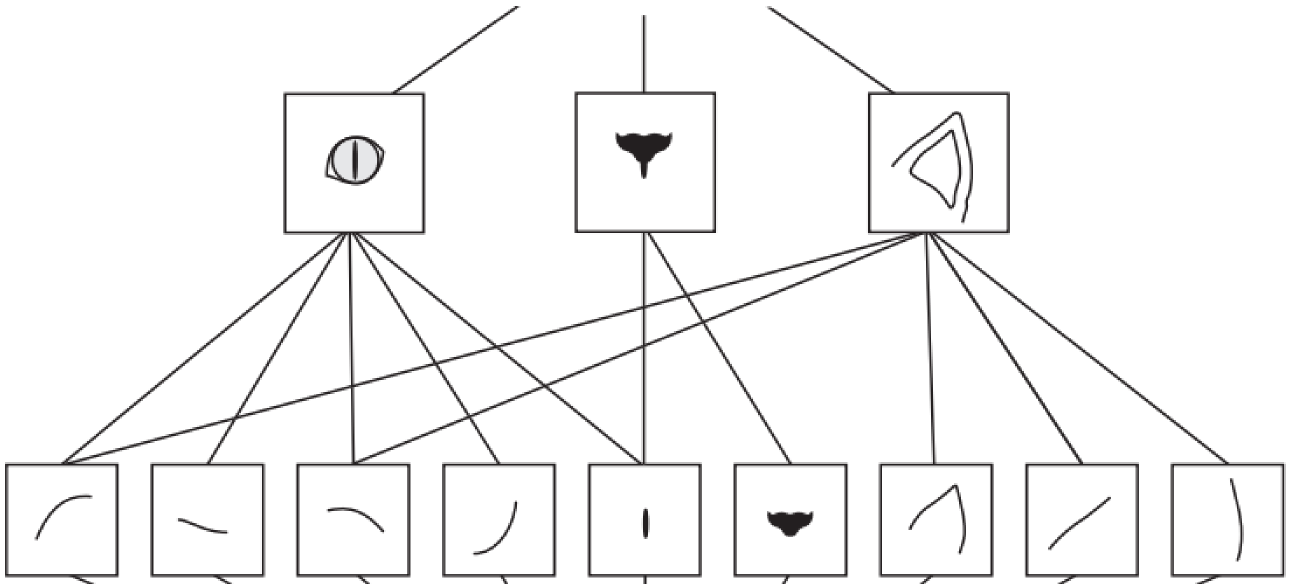
- Spatial Hierarchies and Local Receptive Fields:

CNNs are specifically designed for tasks involving grid-like data, such as images. They use spatial hierarchies and local receptive fields, meaning they capture patterns in small local regions of the input data.

- Parameter Sharing
- Translation Invariance: CNNs are inherently translation-invariant, meaning they can recognize patterns regardless of their position in the input space.
- Feature Hierarchy: CNNs automatically learn hierarchical representations of features. Lower layers capture simple features like edges and textures, while deeper layers capture more complex and abstract features.

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).





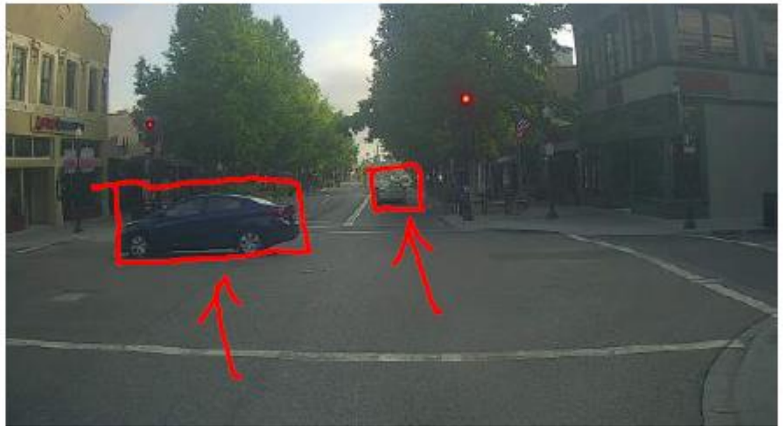
Computer Vision Problems

Image Classification

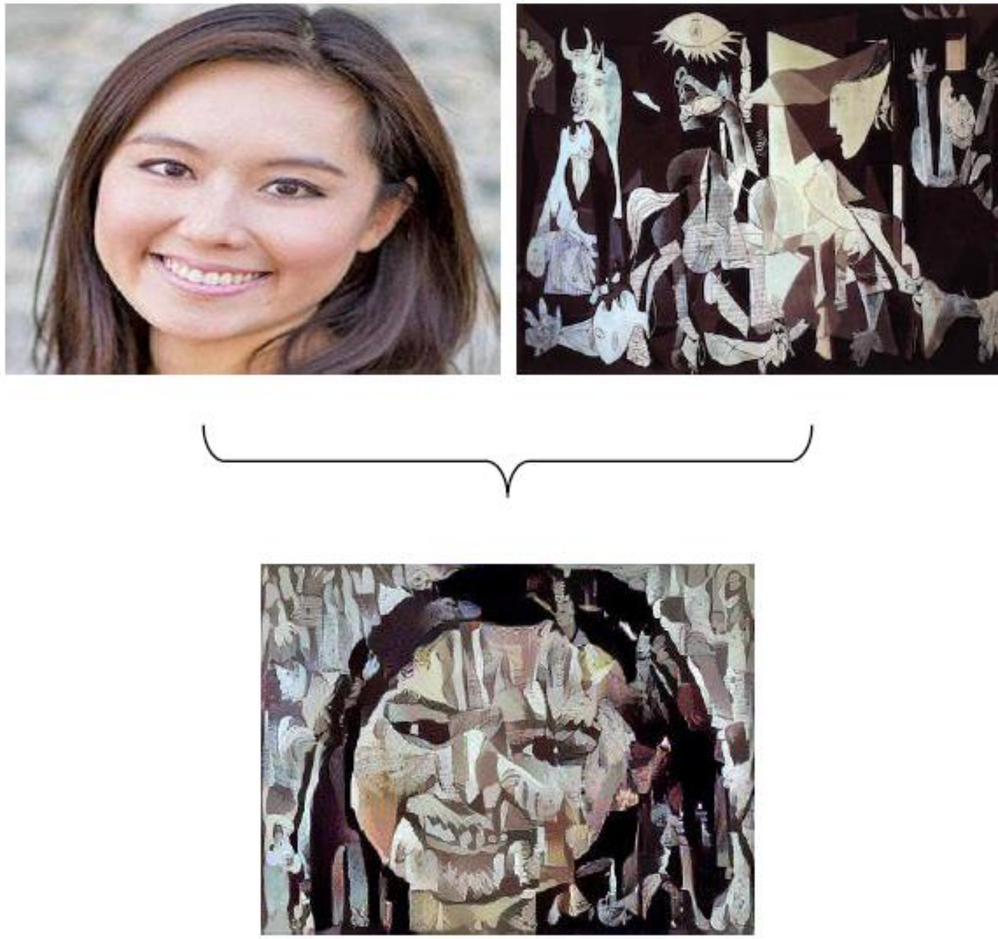
64x64

→ Cat? (0/1)

Object detection

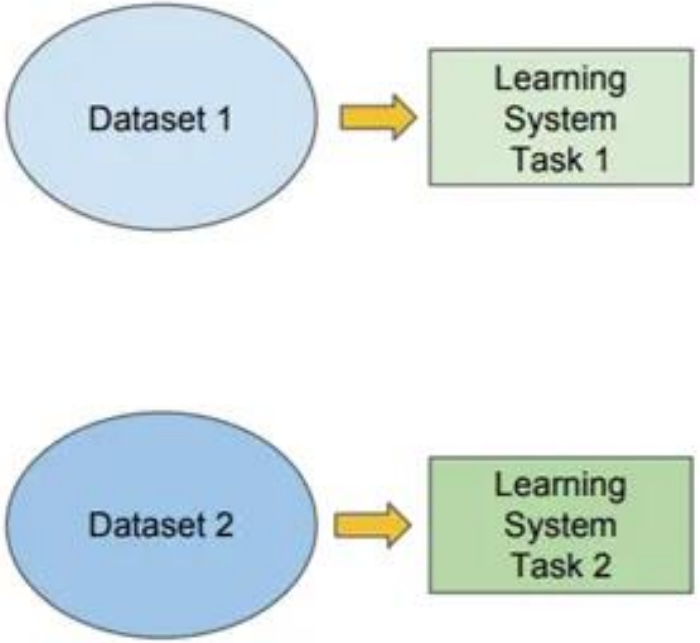


Neural Style Transfer

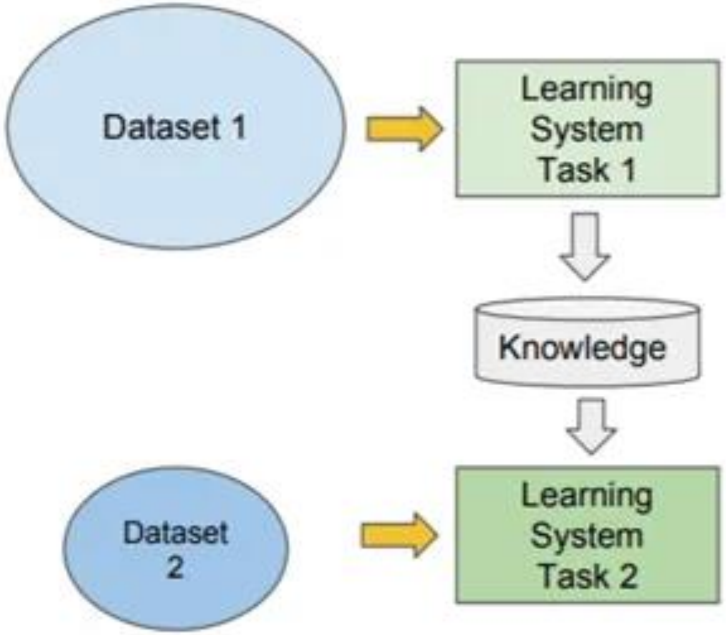


Traditional ML vs Transfer Learning

- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



- Learning of a new tasks relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data



Transfer Learning

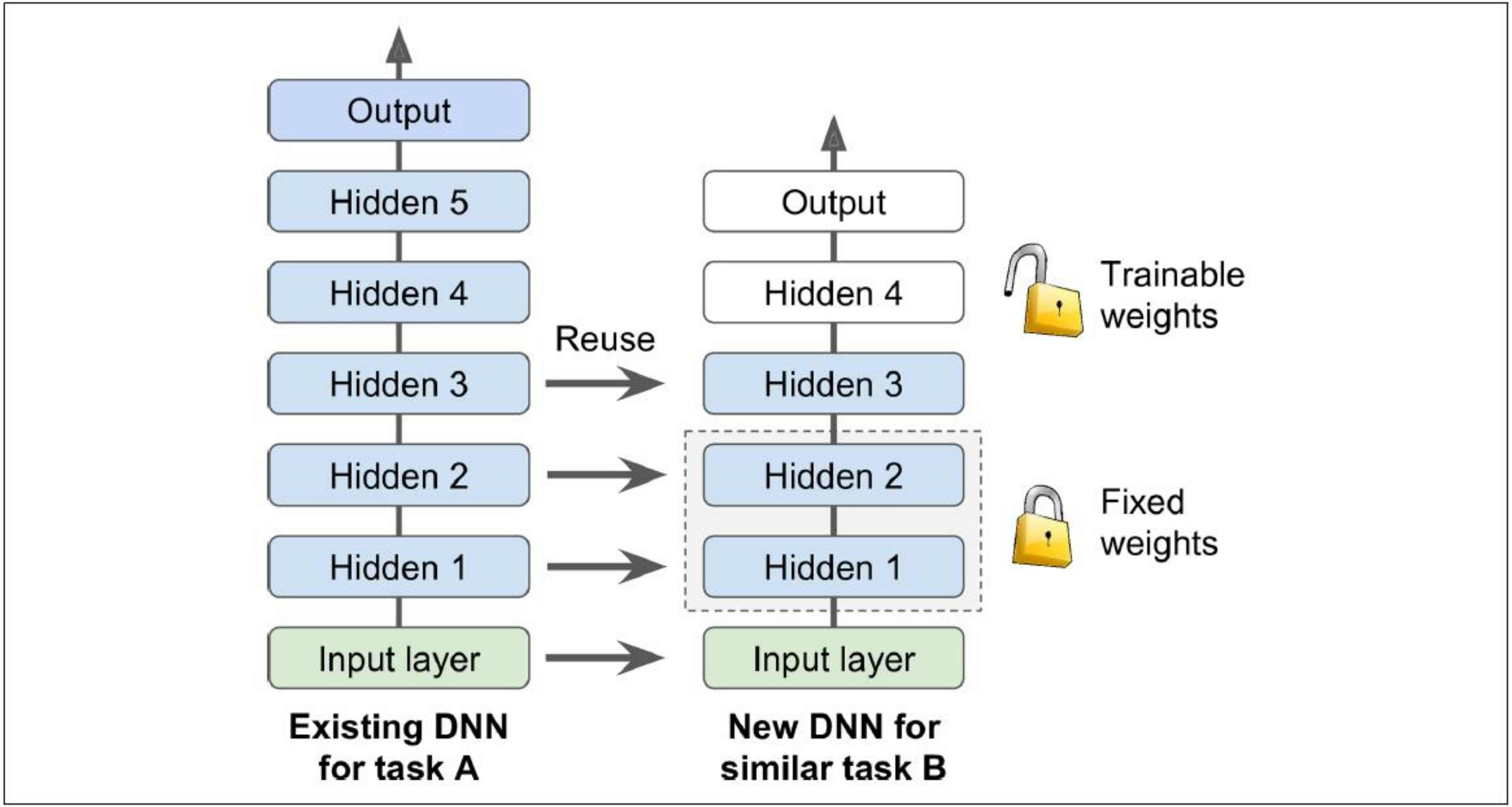
- Transfer learning is probably where neural networks have a unique advantage over the shallow models.
- In transfer learning, you pick an existing model trained on some dataset, and you adapt this model to predict examples from another dataset, different from the one the model was built on. For example, imagine you have trained your model to recognize (and label) wild animals on a big labelled dataset. After some time, you have another problem to solve: you need to build a model that would recognize domestic animals.
- With shallow learning algorithms, you do not have many options: you have to build another big labelled dataset, now for domestic animals.

With neural networks, the situation is much more favourable. Transfer learning in neural networks work like this.

1. You build a deep model on the original big dataset (wild animals).
2. You compile a much smaller labelled dataset for your second model (domestic animals).
3. You remove the last one or several layers from the first model. Usually, these are layers responsible for the classification or regression;
4. You replace the removed layers with new layers adapted for your new problem.
5. You “freeze” the parameters of the layers remaining from the first model.
6. You use your smaller labelled dataset and gradient descent to train the parameters of only the new layers.

Why upper-level layers are removed??

- The output layer of the original model should usually be replaced since it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.
- Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task.



Reusing pretrained layers

Transfer learning

- Apply the knowledge you took in a task A and apply it in another task B.
- For example, you have trained a cat classifier with a lot of data, you can use the part of the trained NN it to solve x-ray classification problem.
- To do transfer learning, delete the last layer of NN and it's weights and:
    - i. Option 1: if you have a small data set - keep all the other weights as a fixed weights. Add a new last layer(-s) and initialize the new layer weights and feed the new data to the NN and learn the new weights.
    - ii. Option 2: if you have enough data you can retrain all the weights.
- Option 1 and 2 are called **fine-tuning** and training on task A called **pretraining**.

- When transfer learning make sense:
    - Task A and B have the same input X (e.g. image, audio).
    - You have a lot of data for the task A you are transferring from and relatively less data for the task B your transferring to.
    - Low level features from task A could be helpful for learning task B.