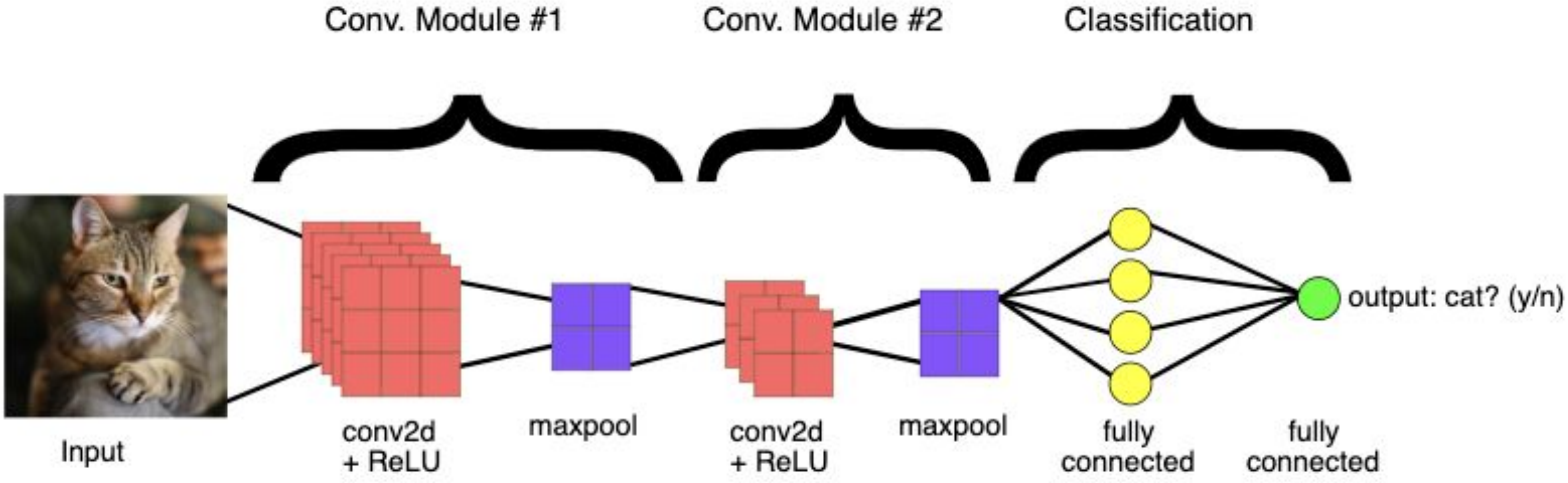
Convolutional Neural Network

HPC-Gateway Fall School

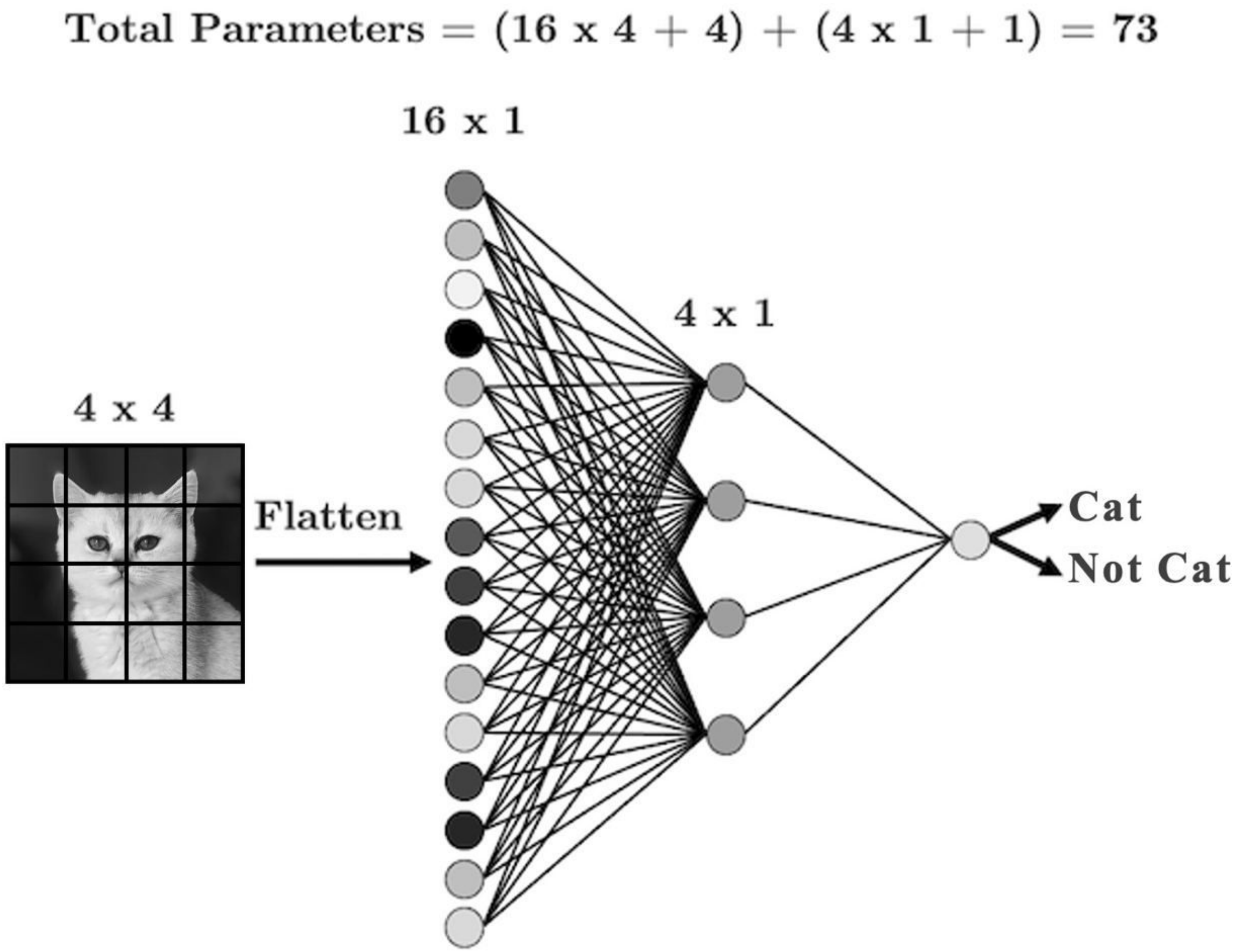
Israt Jahan Tulin, Scientific Researcher, AI Consultant Team, HZDR, Germany
Date: November 04, 2025

CNN Architecture



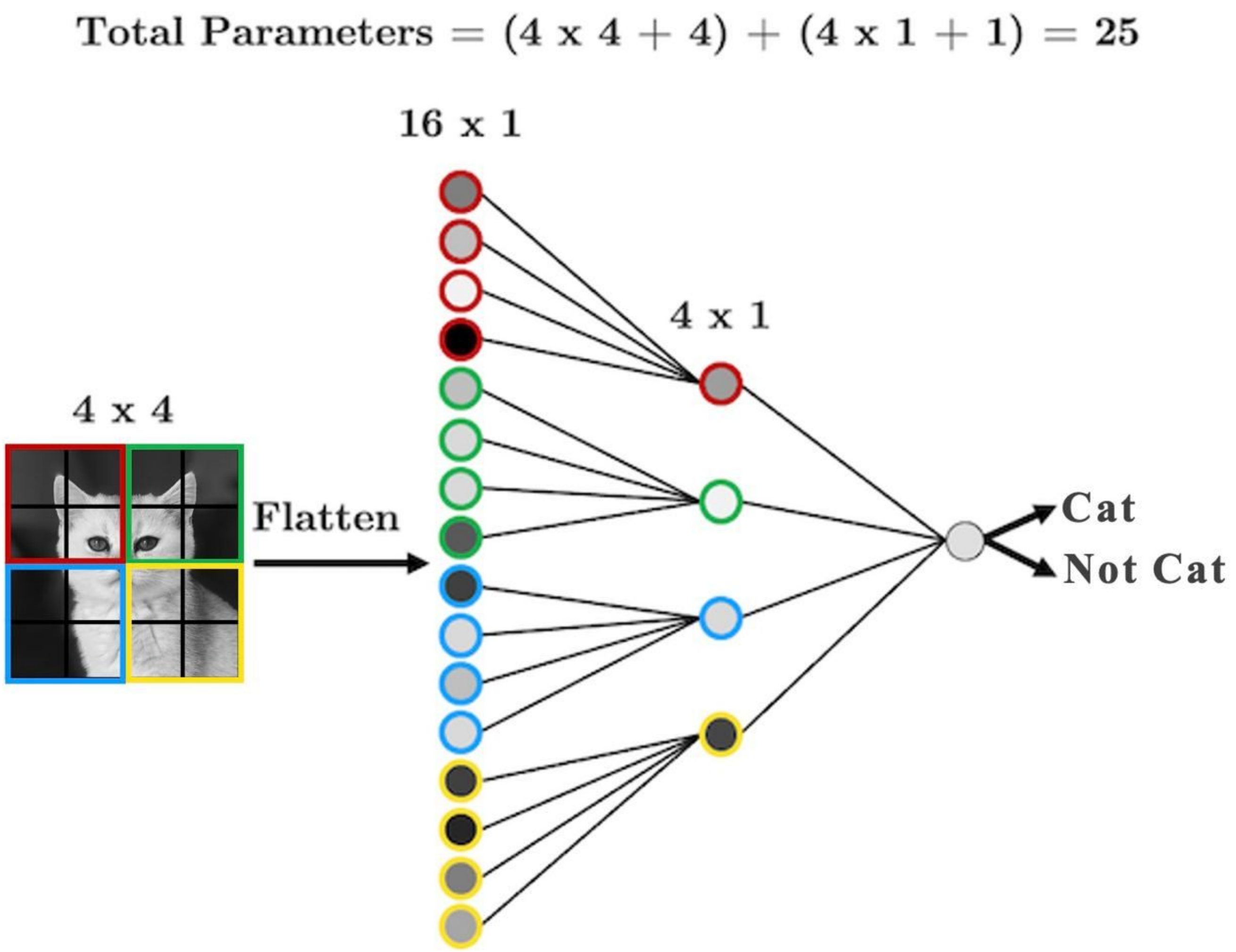
https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Fully Connected Network vs. CNN Idea



https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

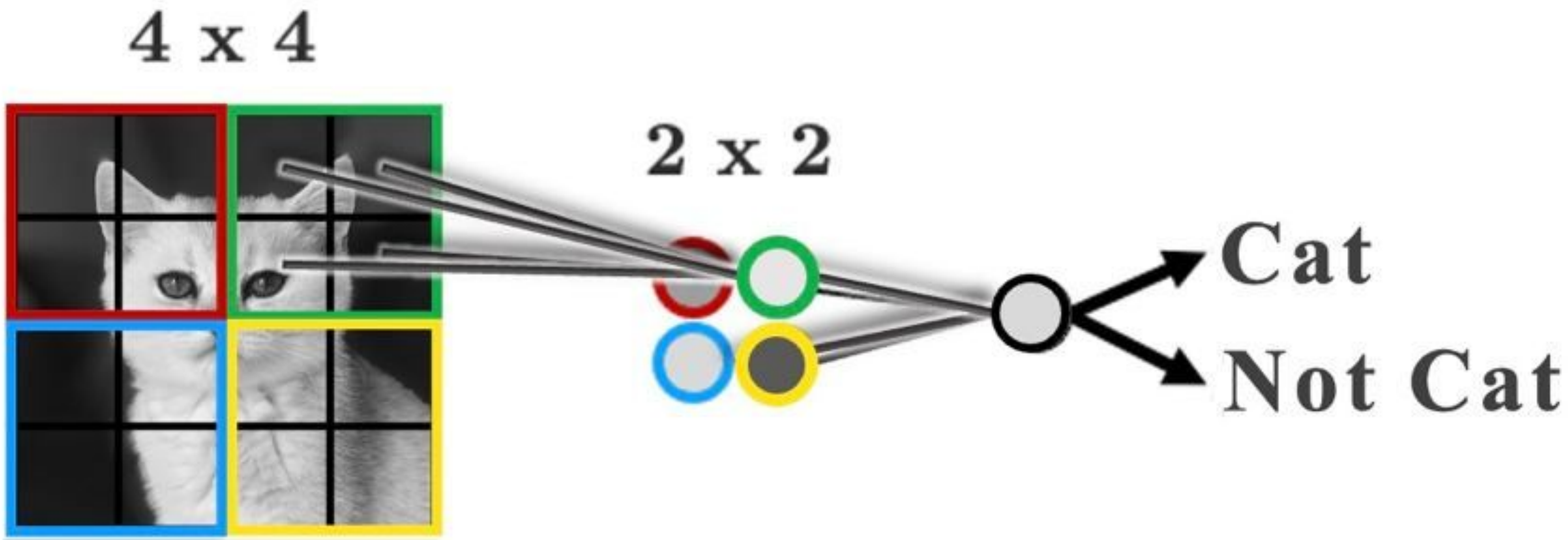
Local Connectivity Concept



4

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

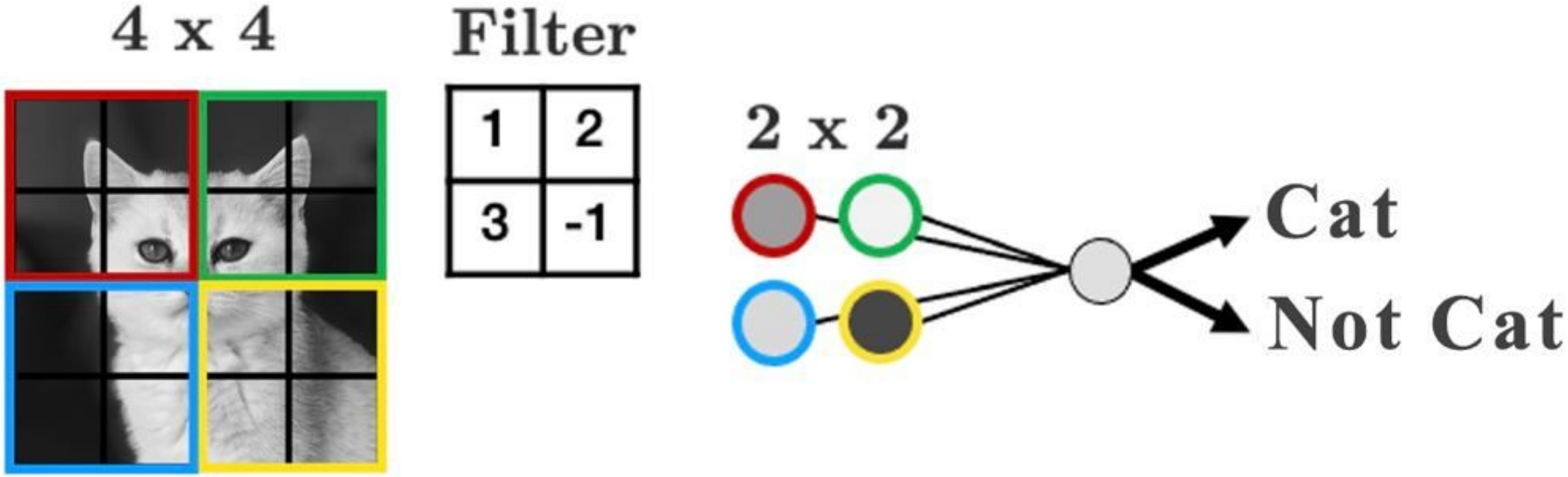
Local Connectivity Concept



NB: only "green" connections shown for simplicity.

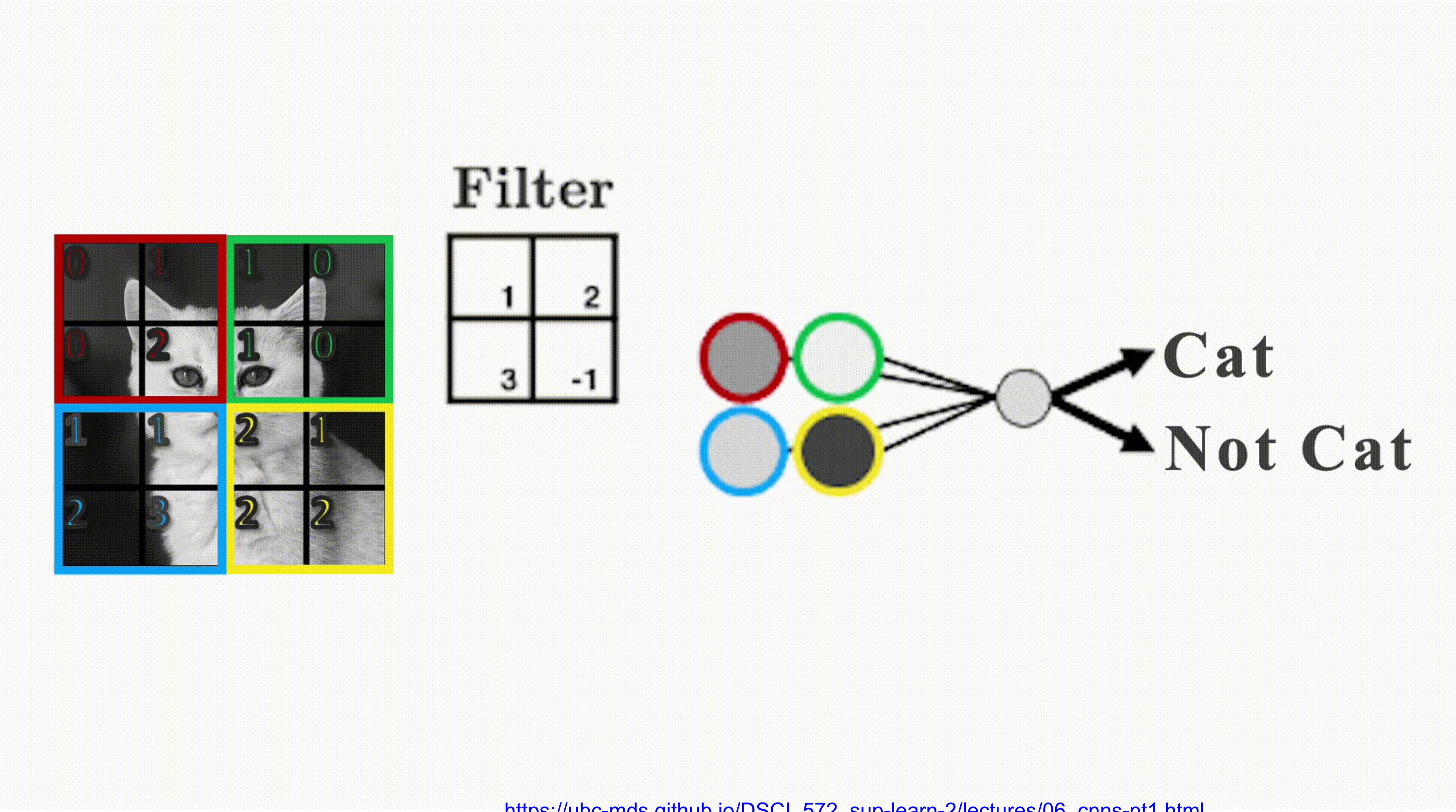
https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Filter Representation – introducing weight filter



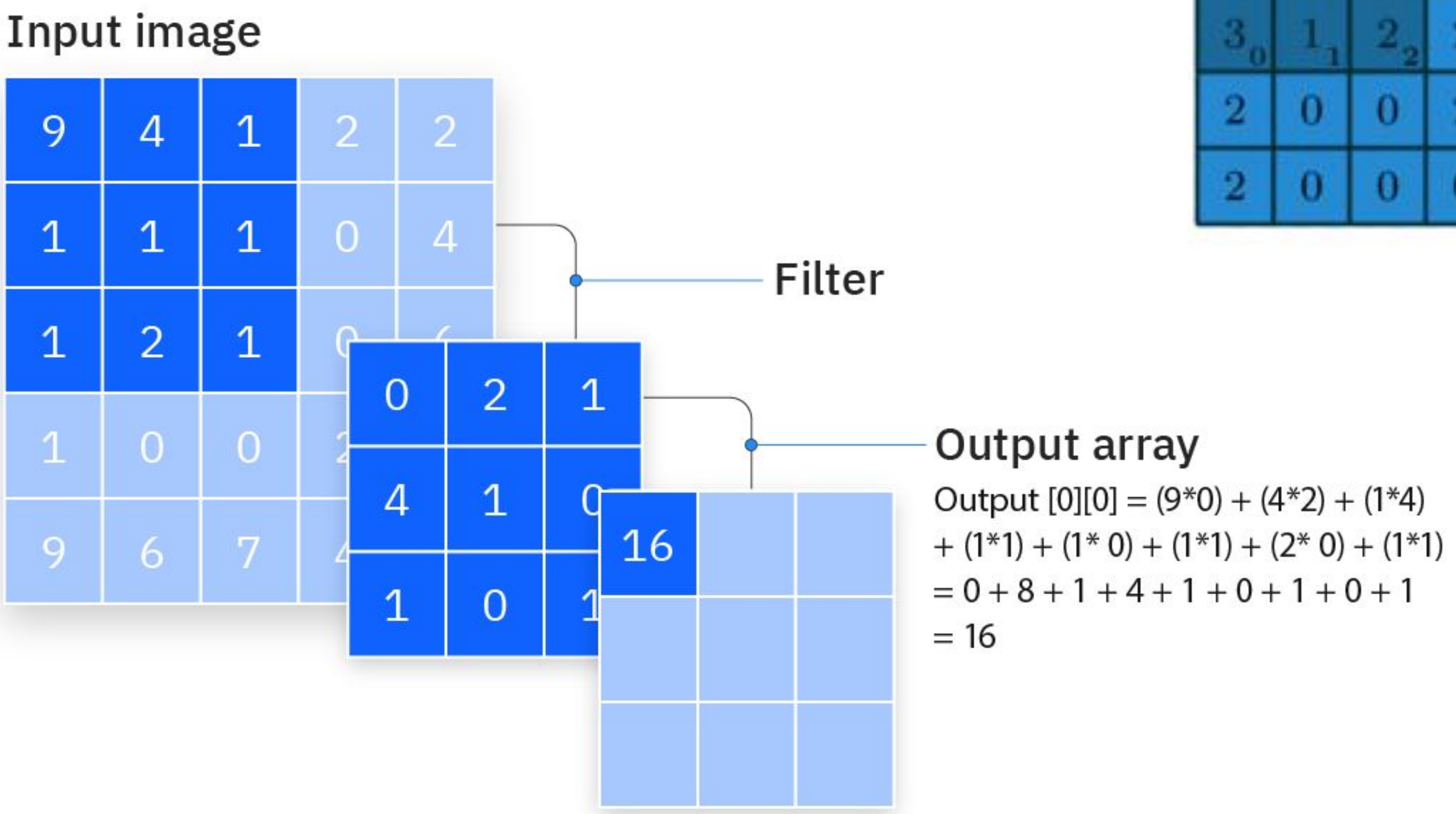
https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

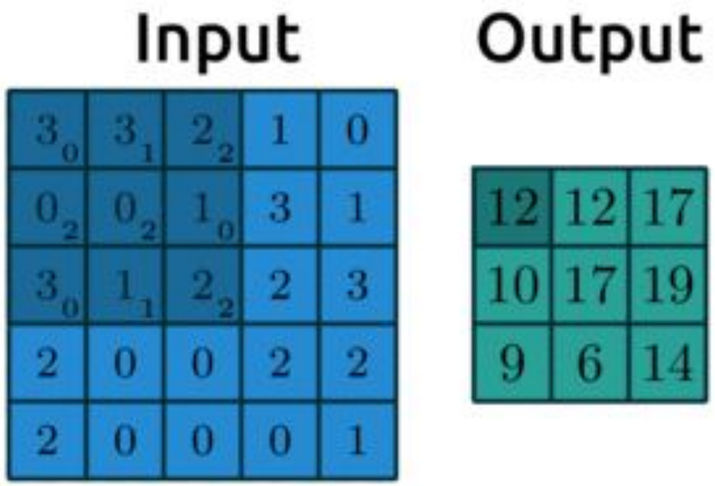
Example 1: Convolution Operation



https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

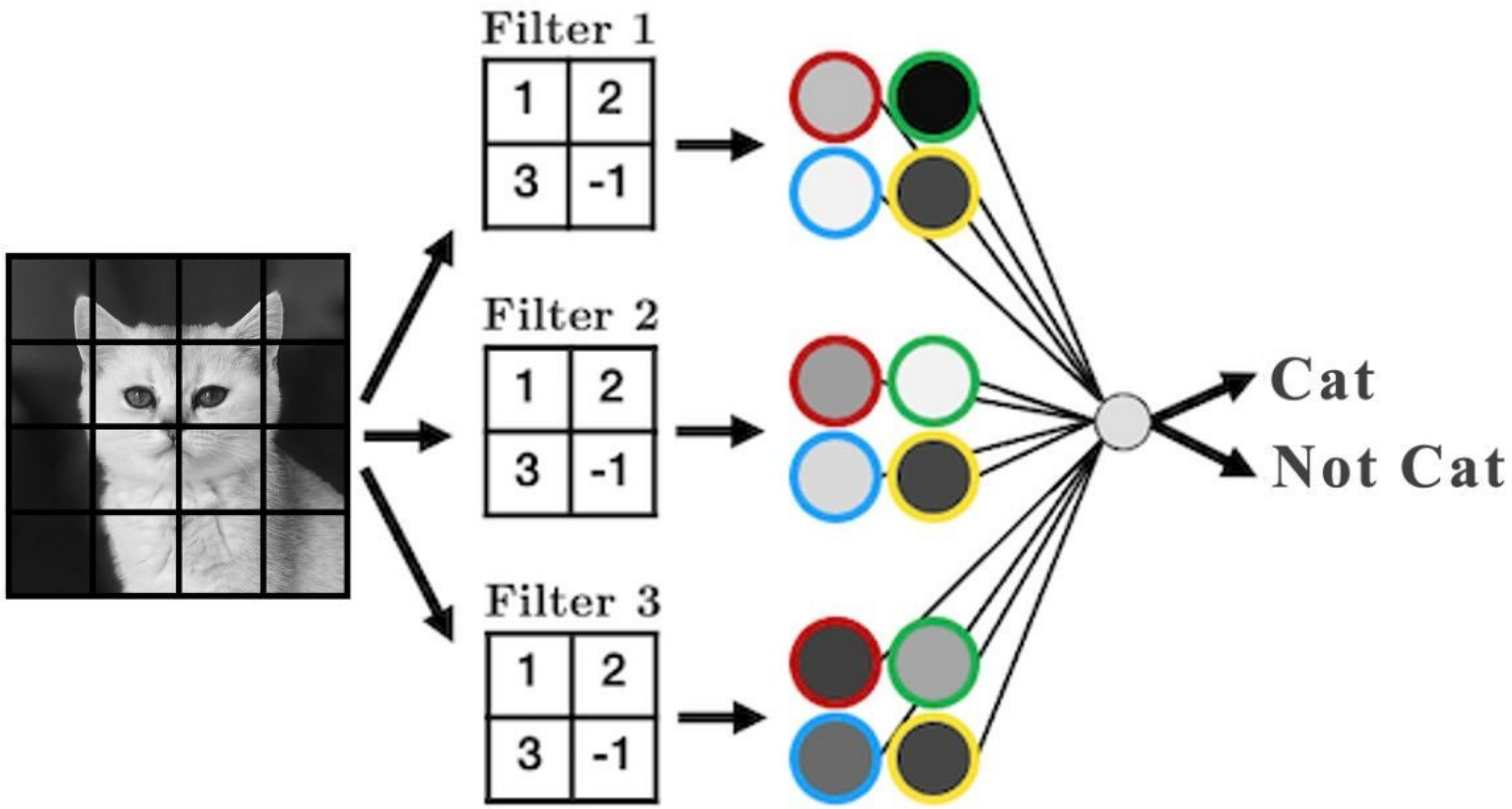
Example 2: Convolution Operation





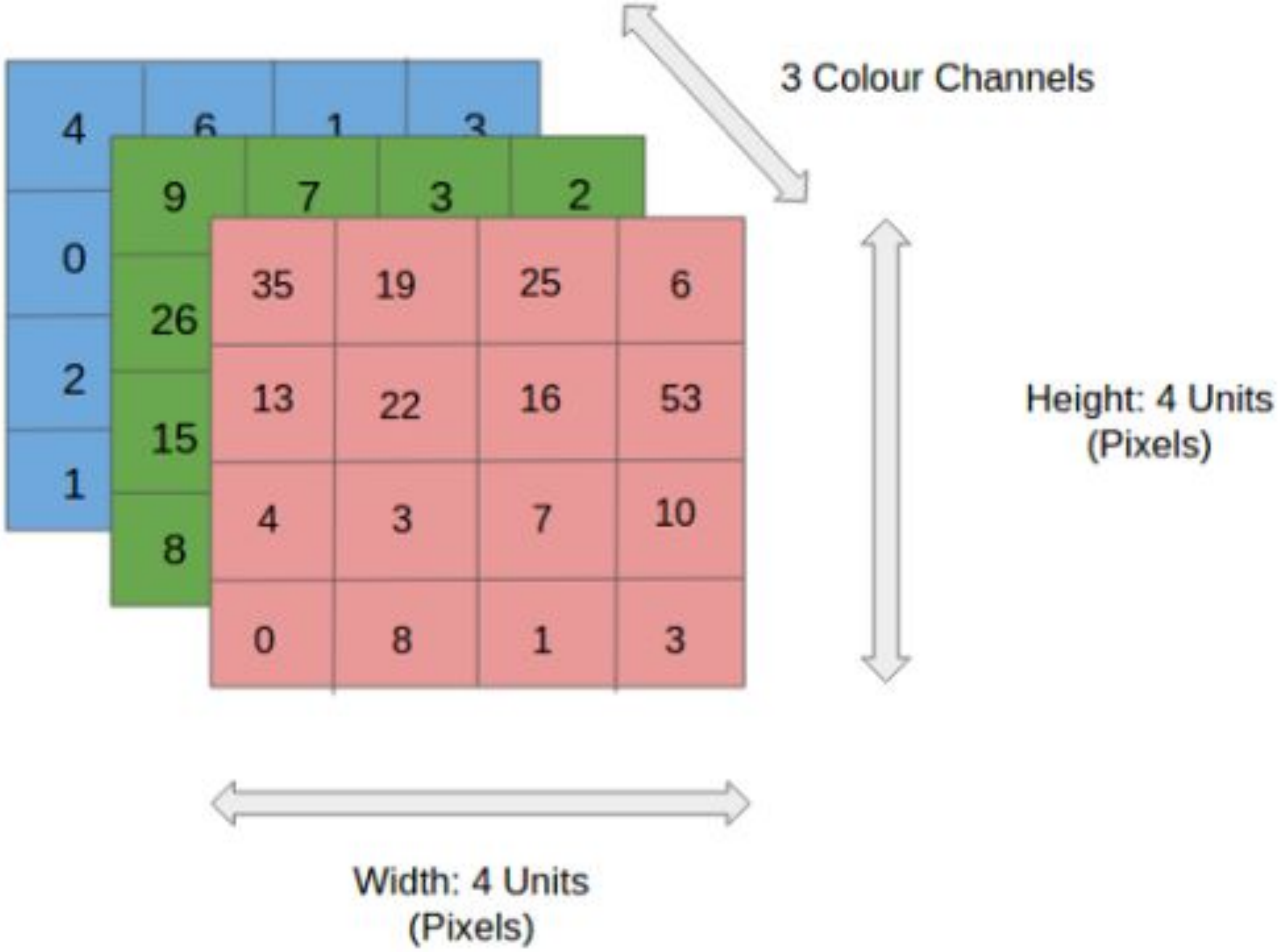
https://www.ibm.com/think/topics/convolutional-neural-networks

Multiple Filters / Building CNNs

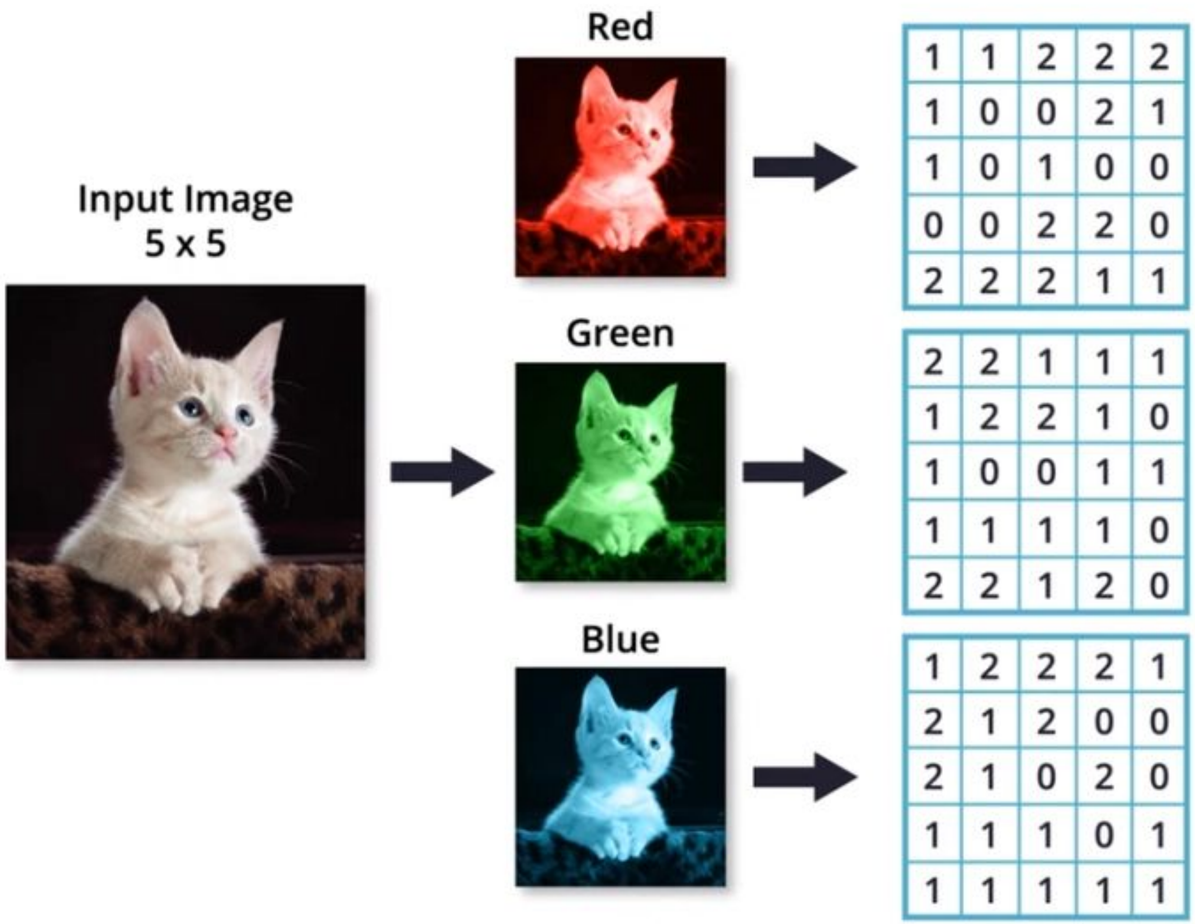


https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

RGB image - 3 color channel

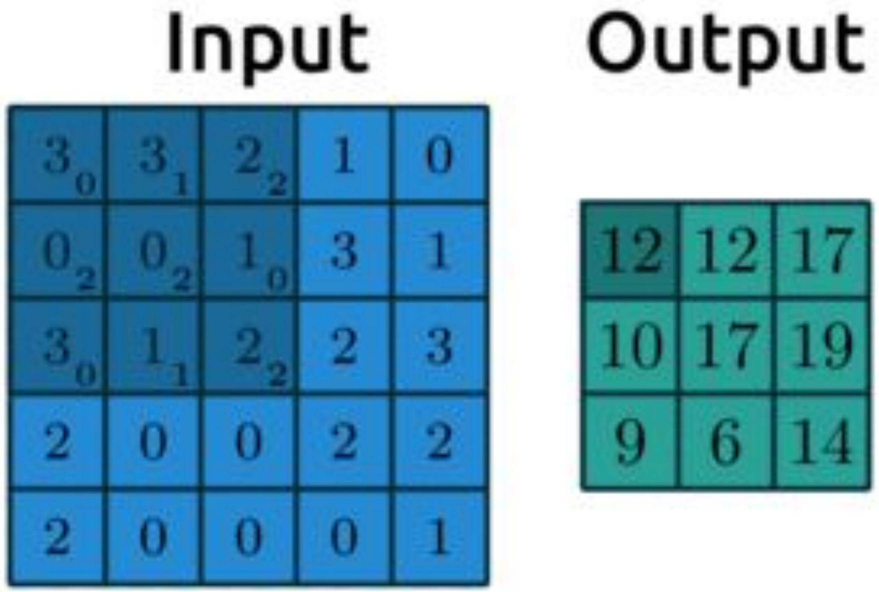


Convolution Operation for color image



https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41

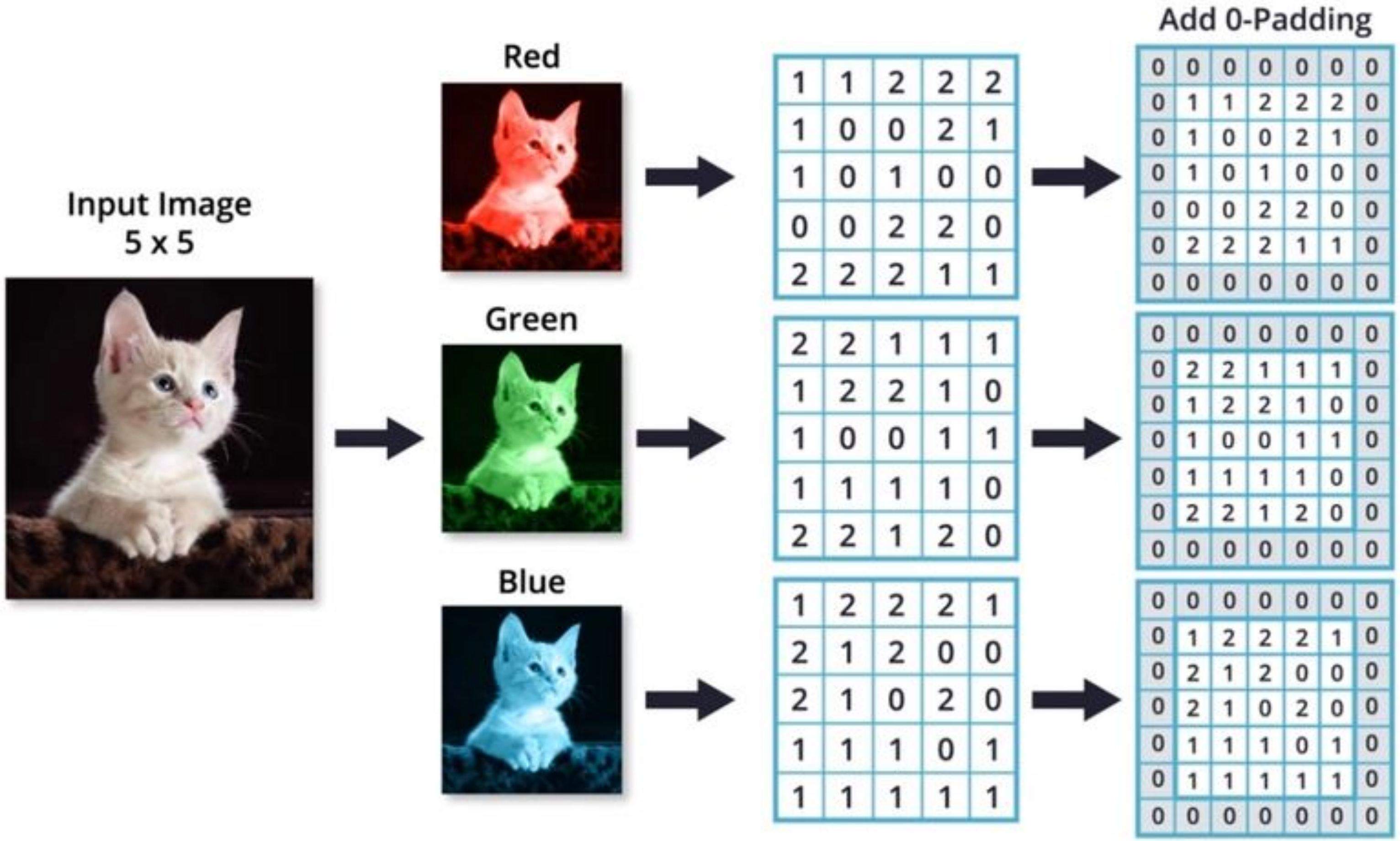
Kernels Are Only Applied Where They Fully Fit on the Input



- We usually use odd numbers for filters so that they are applied symmetrically around our input data
- Notice in the GIF earlier that the output from applying our kernel was smaller than the input.

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Padding: Ensures edge pixels are included in convolution



https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41

Padding: valid vs same

Padding: "valid"

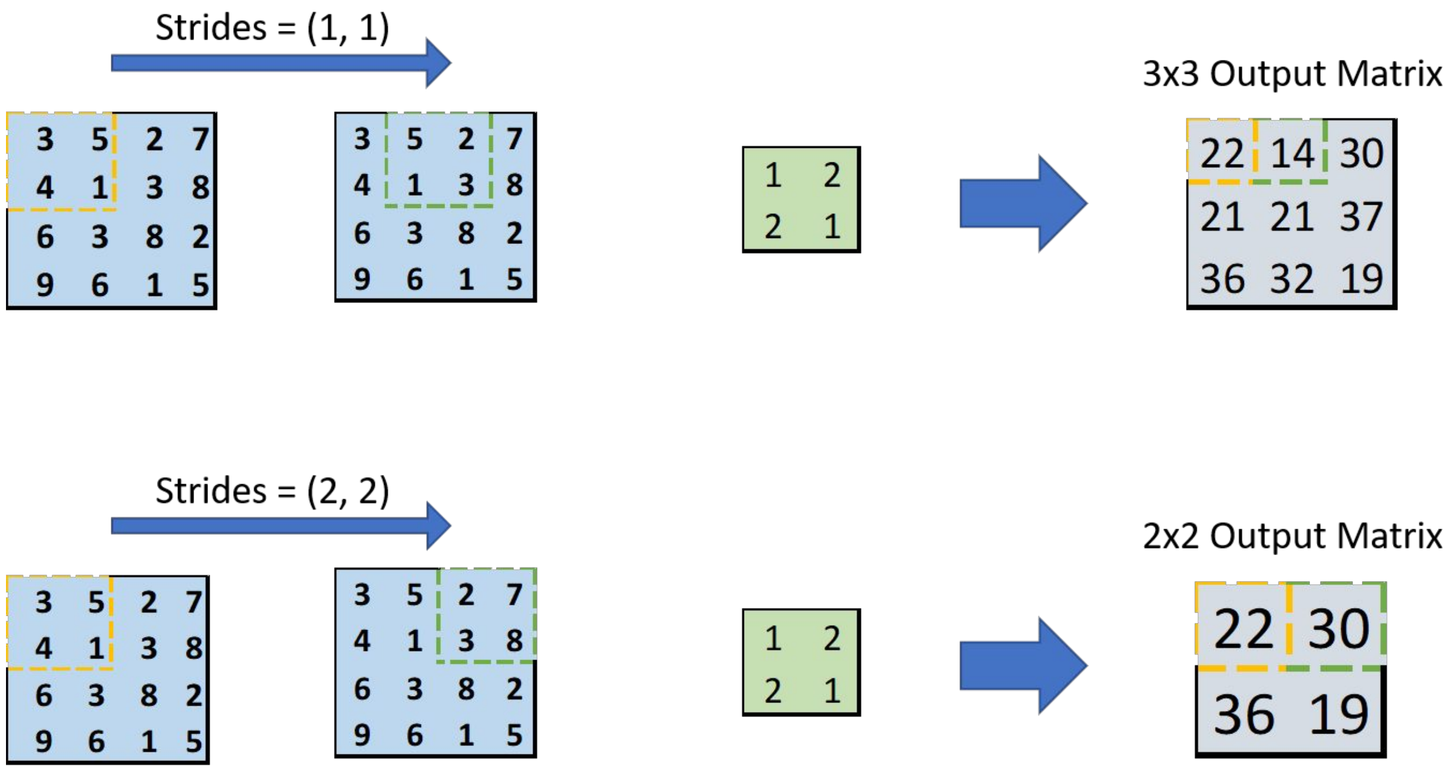
$$\begin{bmatrix}3&5&2&7\\4&1&3&8\\6&3&8&2\\9&6&1&5\end{bmatrix} \quad \begin{bmatrix}1&2&1\\2&1&2\\1&1&2\end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix}55&52\\57&50\end{bmatrix}$$

Padding: "same" **output size to stay the same**: use padding = (kernel_size - 1) / 2

$$\begin{bmatrix}0&0&0&0&0&0\\0&3&5&2&7&0\\0&4&1&3&8&0\\0&6&3&8&2&0\\0&9&6&1&5&0\\0&0&0&0&0&0\end{bmatrix} \quad \begin{bmatrix}1&2&1\\2&1&2\\1&1&2\end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix}19&26&46&22\\29&55&52&40\\42&57&50&43\\36&46&44&19\end{bmatrix}$$

https://towardsdatascience.com/understand-transposed-convolutions-and-build-your-own-transposed-convolution-layer-from-scratch-4f5d97b2967/

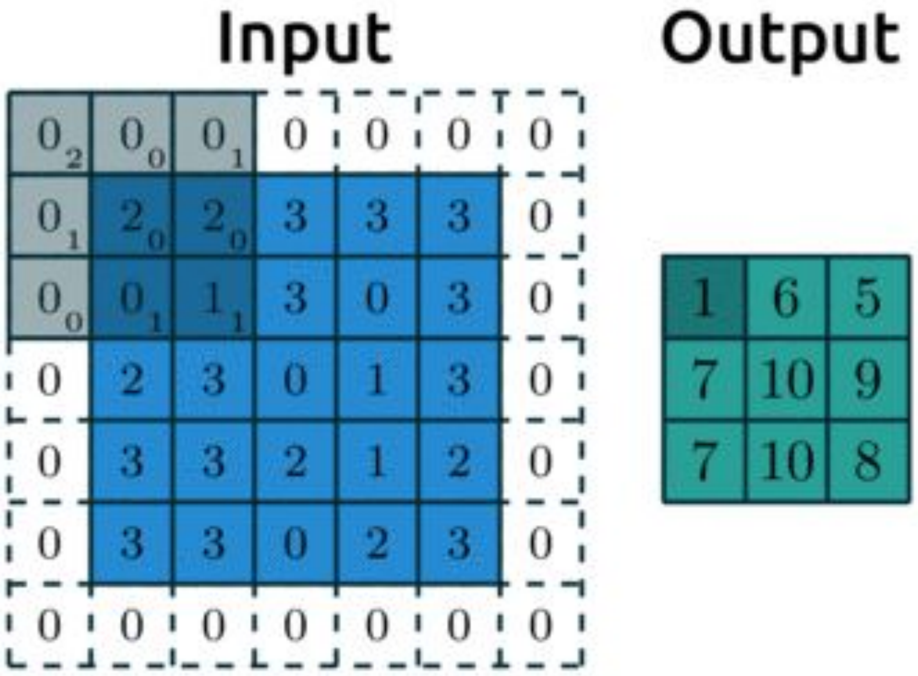
Stride: Controls How Far the Kernel Moves Over the Input Image



https://towardsdatascience.com/understand-transposed-convolutions-and-build-your-own-transposed-convolution-layer-from-scratch-4f5d97b2967/

Using of both strides and padding

- Be default, our kernels are only applied where the filter fully fits on top of the input
- But we can control this behaviour and the size of our output with:
  - `padding`: “pads” the outside of the input with 0’s to allow the kernel to reach the boundary pixels
  - `strides`: controls how far the kernel “steps” over pixels.



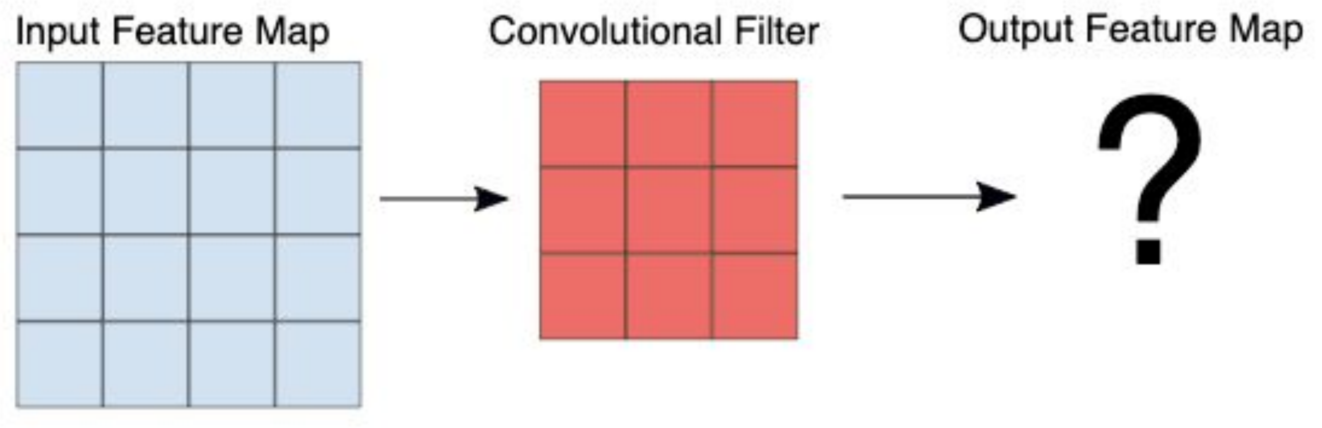
RIght side there’s an example with:

- `padding=1`: we have `1` layer of 0’s around our border
- `strides=(2, 2)`: our kernel moves 2 data points to the right for each row, then moves 2 data points down to the next row

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Exercise

A two-dimensional, 3x3 convolutional filter is applied to a two-dimensional 4x4 input feature map (no padding added):



What is the shape of the output feature map?

- (A) 4 by 4
- (B) 3 by 3
- (C) 2 by 2

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Calculation of output feature map

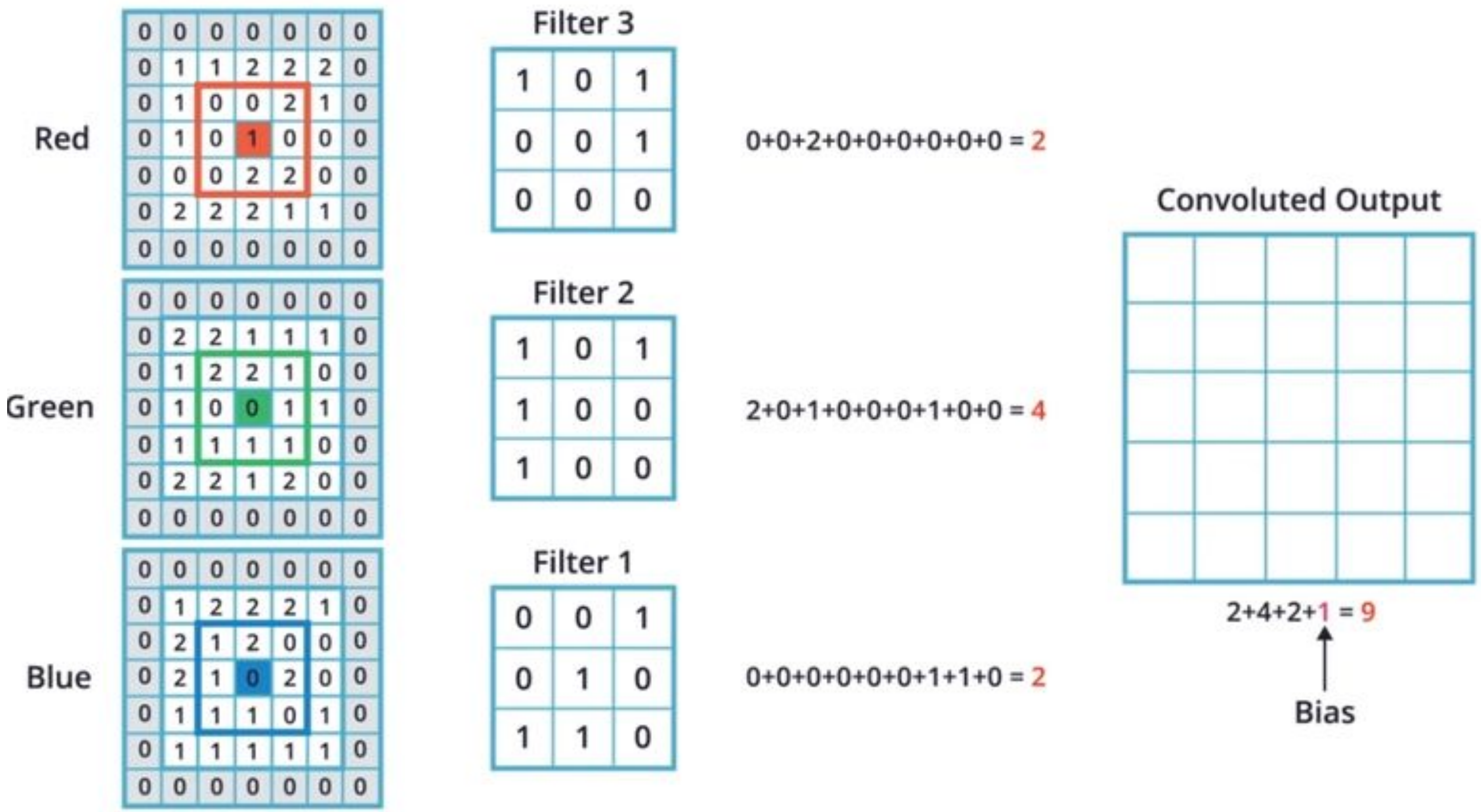
If the $\mathbf{m{\times}m}$ image convolved with the $\mathbf{n{\times}n}$ kernel with stride $\mathbf{s}$ and padding $\mathbf{p}$,

Size of output image: $$\left\lfloor \frac{m+(2\times p)-n}{s}+1 \right\rfloor \times \left\lfloor \frac{m+(2\times p)-n}{s}+1 \right\rfloor$$

Example: $\mathbf{4{*}4}$ image convolved with $3{*}3$ kernel withe stride and padding 1,

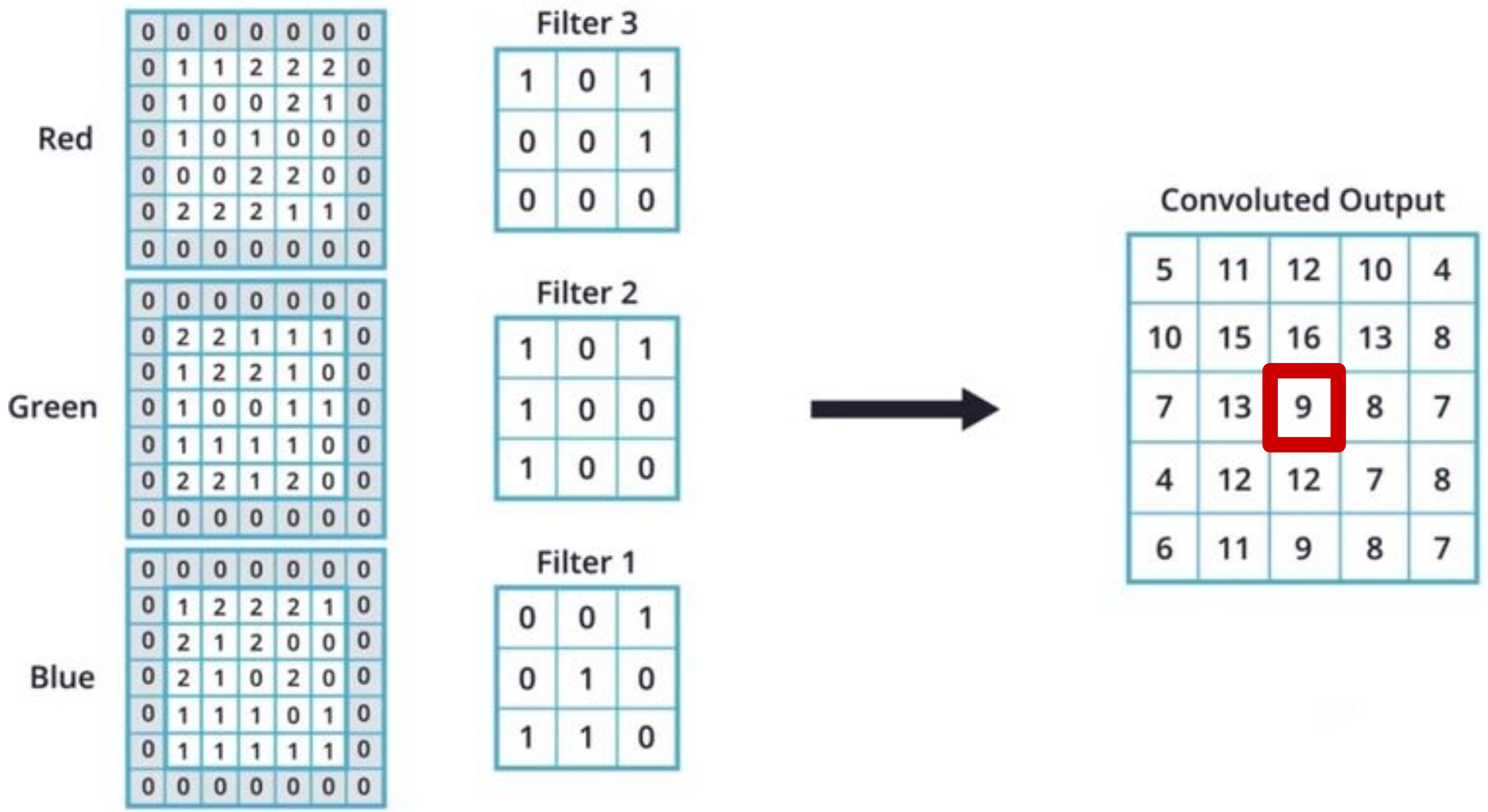
Size of output image: $$\left\lfloor \frac{4+(2\times 1)-3}{1}+1 \right\rfloor \times \left\lfloor \frac{4+(2\times 1)-3}{1}+1 \right\rfloor = 4\times 4$$

Convolution for color image



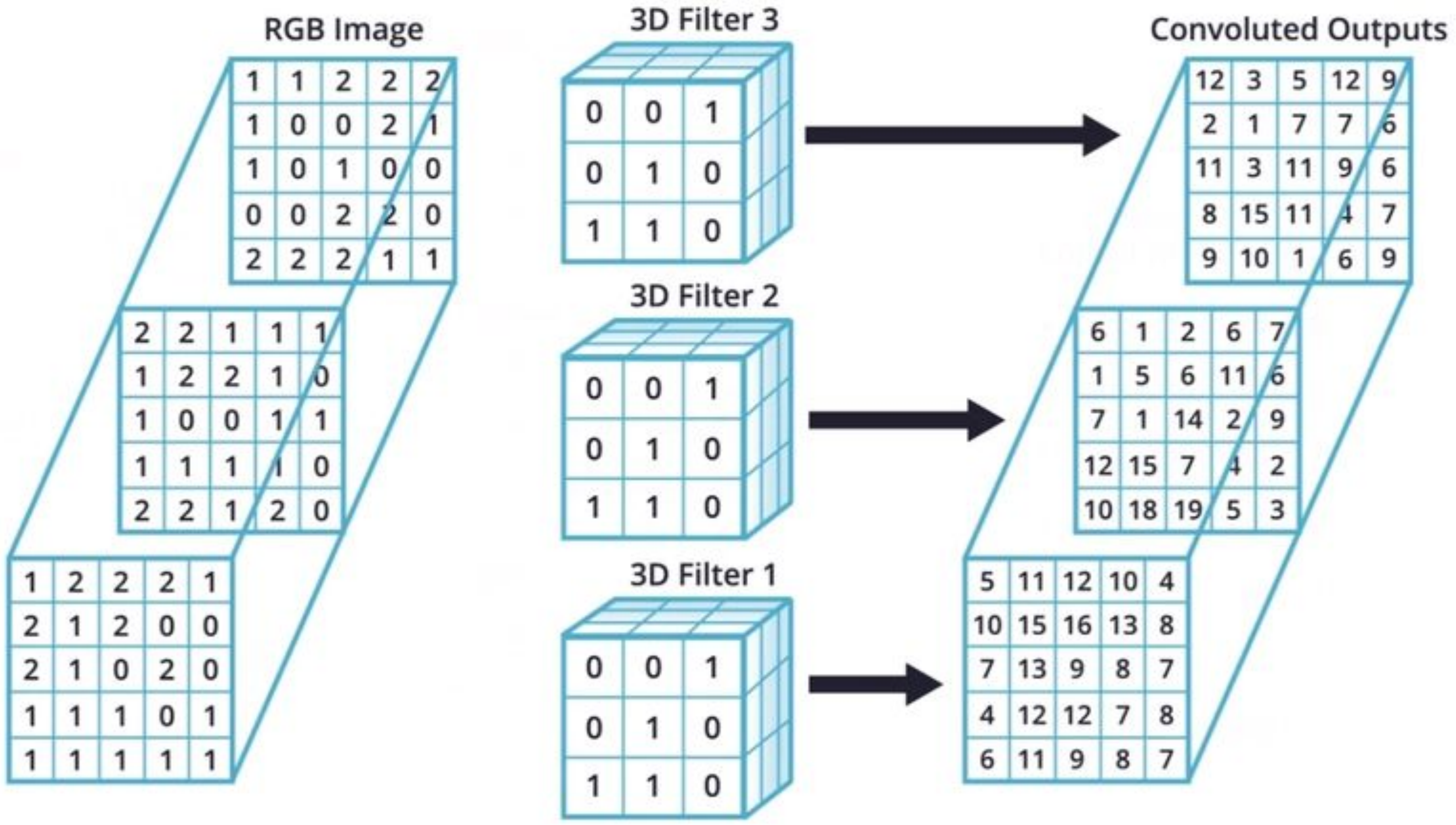
https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41

Convolution for color image



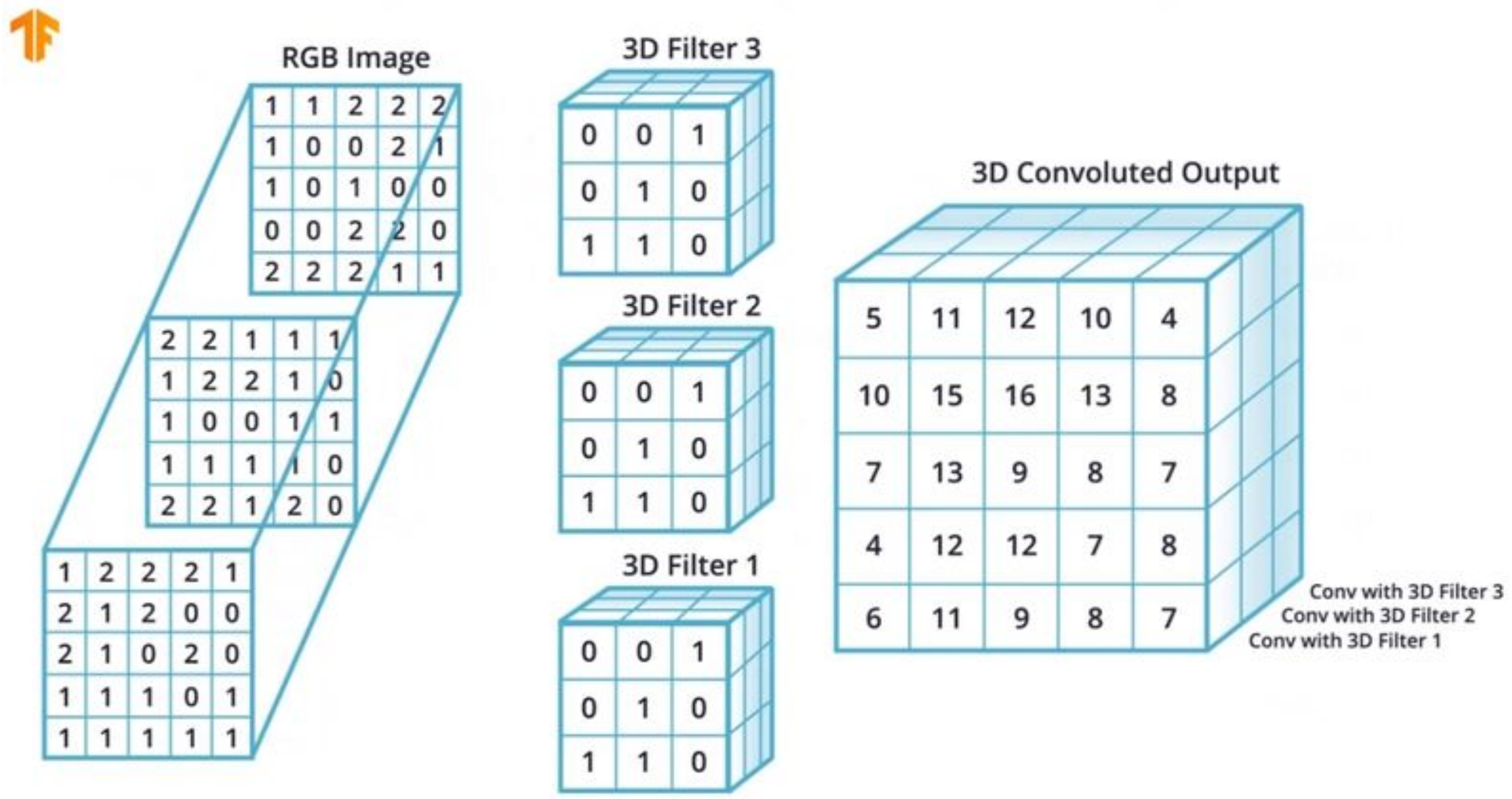
https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41

Multiple filters for each color channel



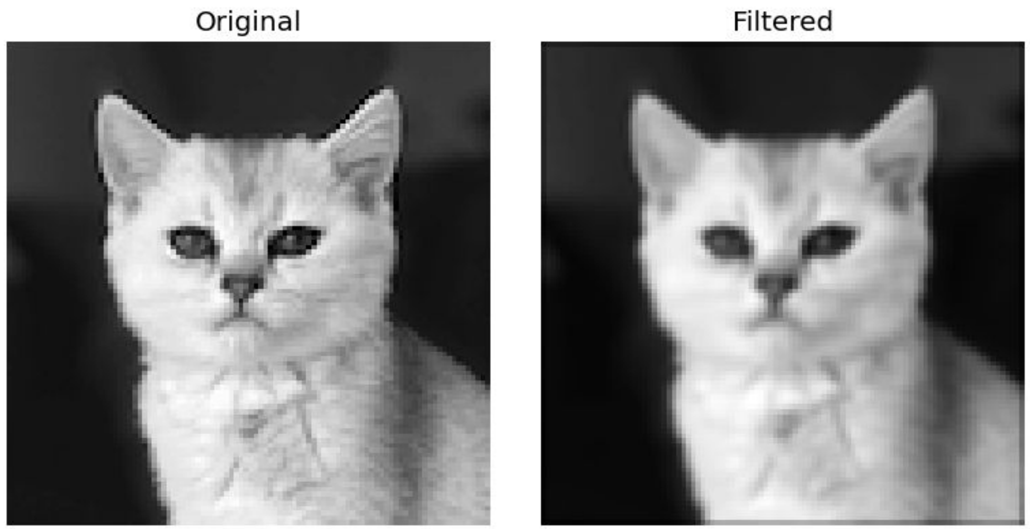
https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41

Convolve with 3D filters



https://dev.to/sandeepbalachandran/machine-learning-convolution-with-color-images-2p41

Effect of a Kernel on an Image



We can blur this image by applying a filter with the following weights:

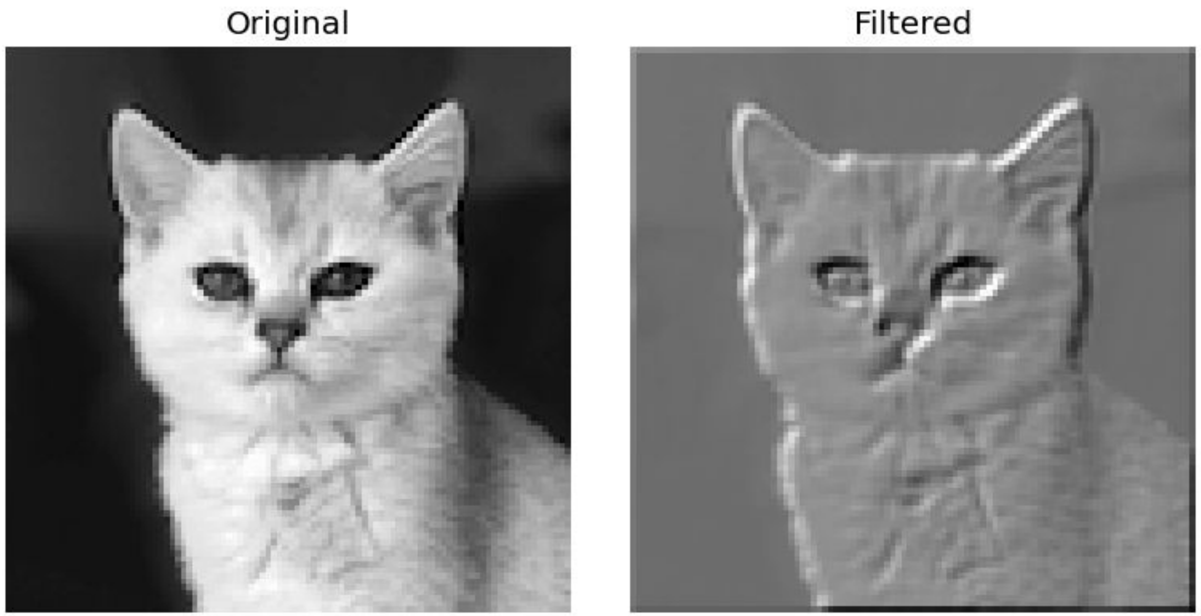
$$\begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

```
kernel = torch.tensor([[[[ 0.0625,  0.1250,  0.0625],
                         [ 0.1250,  0.2500,  0.1250],
                         [ 0.0625,  0.1250,  0.0625]]]])
```

```
plot_conv(image, kernel)
```

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Effect of a Kernel on an Image



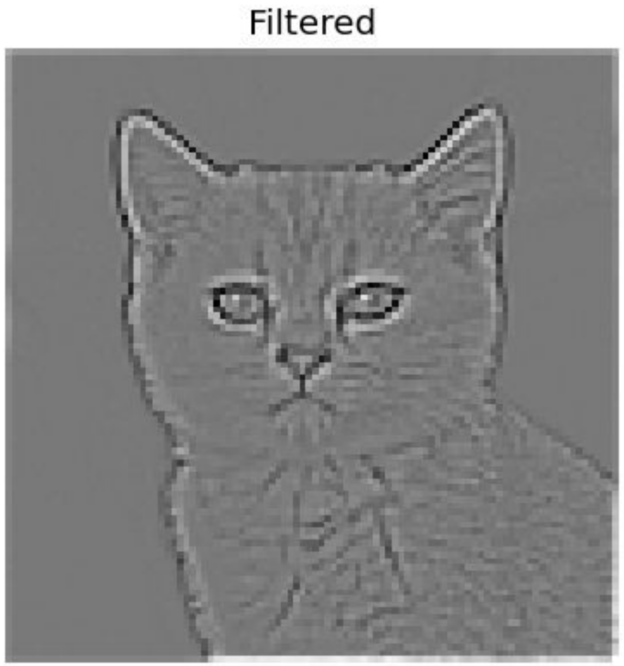
How about this one:

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

```
kernel = torch.tensor([[[[ -2,  -1,  0],
                         [ -1,   1,  1],
                         [  0,   1,  2]]]])
plot_conv(image, kernel)
```

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Effect of a Kernel on an Image



[Here's a great website](#) where we can play around with other filters.

One more:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

```
kernel = torch.tensor([[[[  -1,  -1,   -1],
                         [  -1,   8,   -1],
                         [  -1,  -1,   -1]]]])
plot_conv(image, kernel)
```

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Convolutional Layers

In PyTorch: convolutional layers are defined as torch.nn.Conv2d.

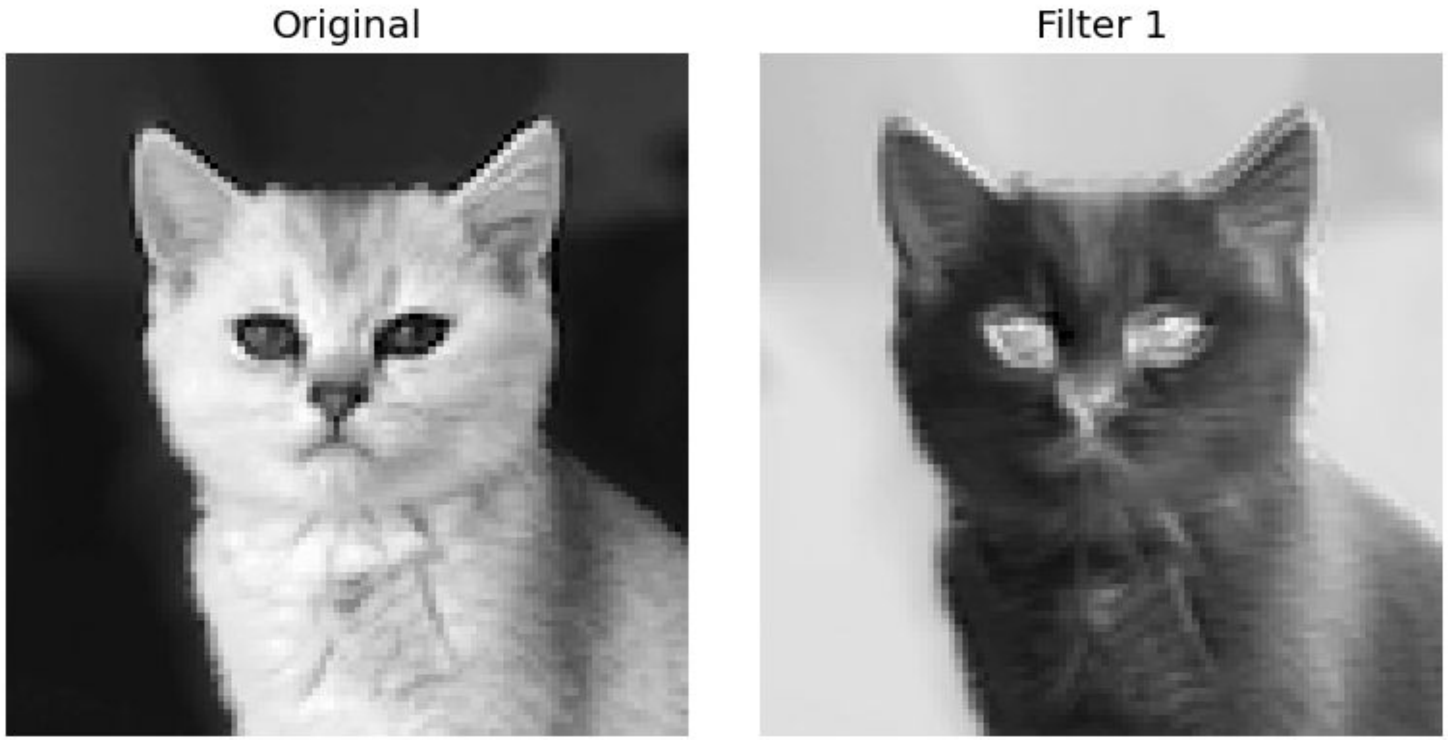
Key arguments to define a convolutional layer:

a) in_channels→ Number of input features (e.g.,
   i) 1 for grayscale images
   ii) 3 for RGB color images)
b) out_channels: → Number of kernels (filters) to learn
   i) Similar to the number of hidden neurons in a dense layer
c) kernel_size → Size of the filter (e.g., 3×3, 5×5, 7×7)
d) Kernels (Filters) → Kernels Are Weights That CNNs Learn during training to Recognize Patterns
e) stride→ Step size of the filter as it moves across the image
f) padding→ Extra pixels added around the border.
   i) Ensures edge pixels are included in convolution

Example 1: Output with 1 Filter

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

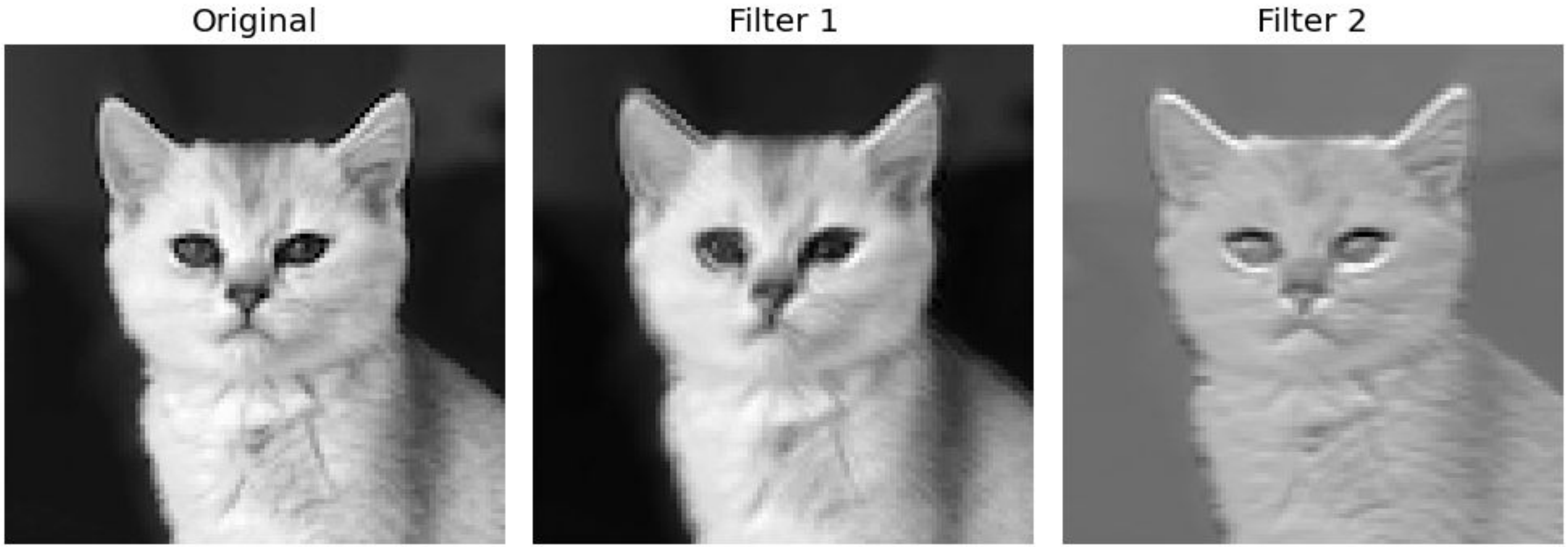
```
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(3, 3))
plot_convs(image, conv_layer)
```



Example 2: Output with 2 Filters

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

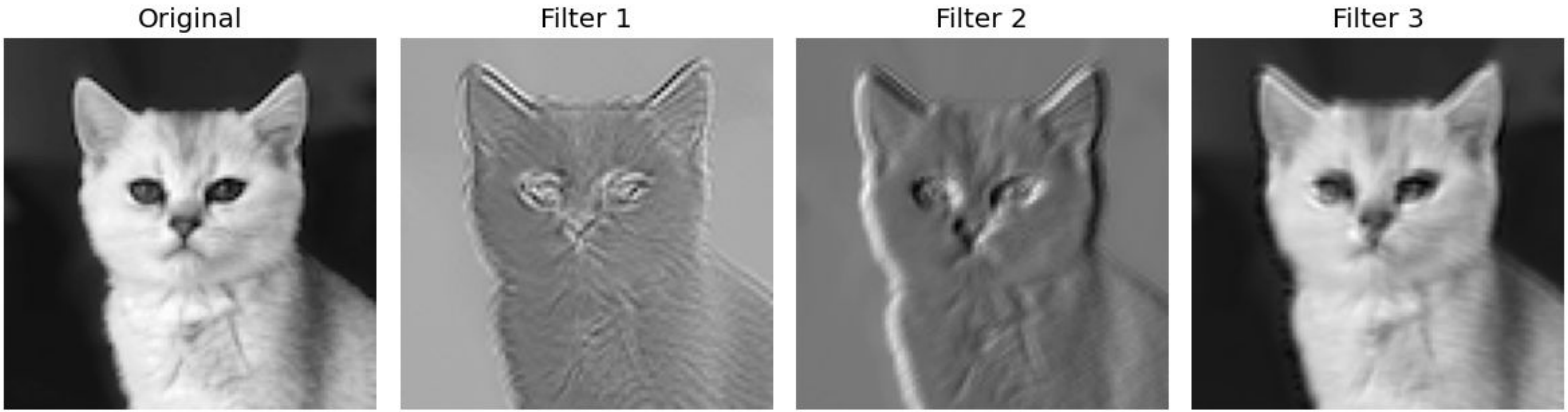
```
conv_layer = torch.nn.Conv2d(1, 2, kernel_size=(3, 3))
plot_convs(image, conv_layer)
```



Example 3: Output with 3 Filters

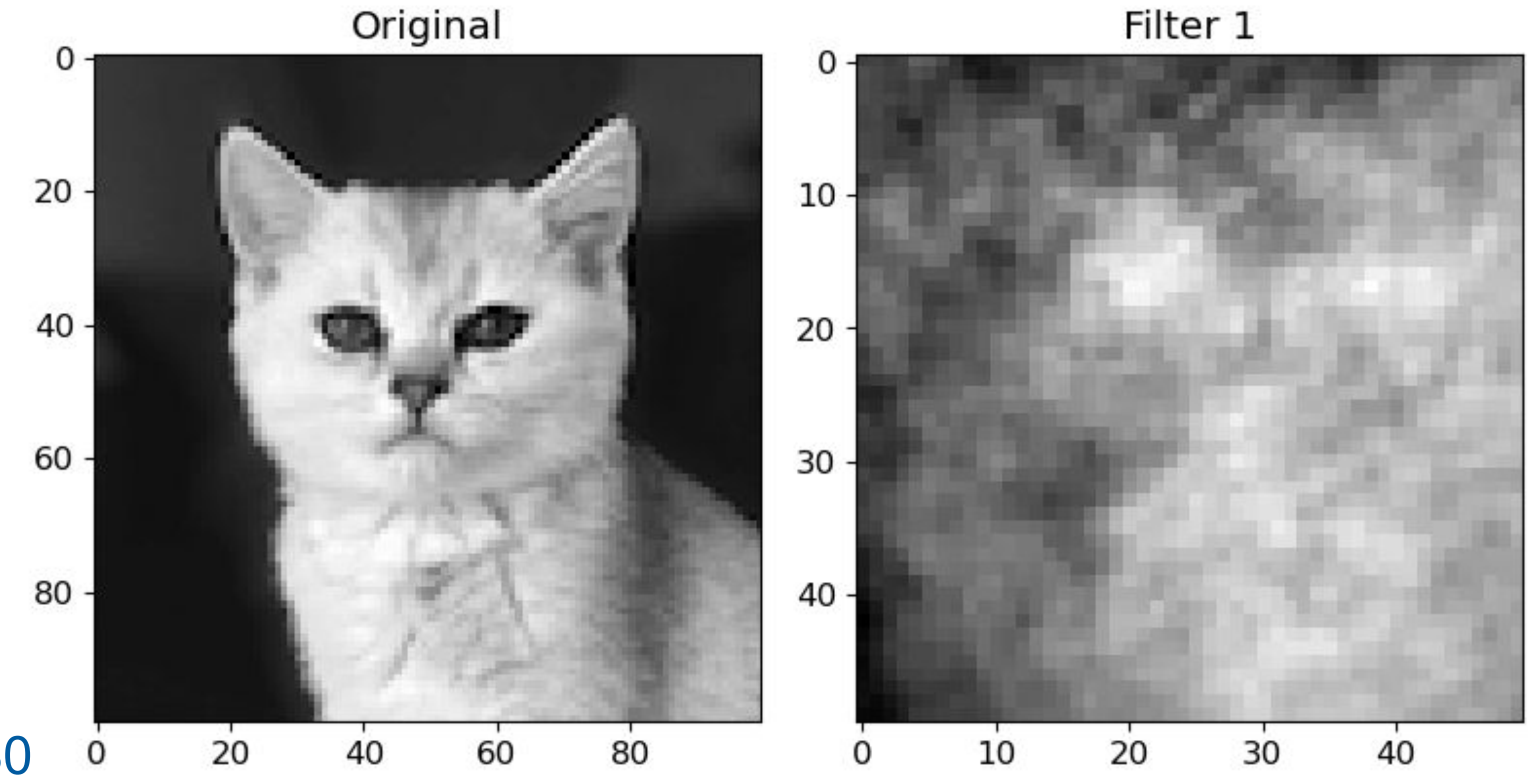
https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

```
conv_layer = torch.nn.Conv2d(1, 3, kernel_size=(5, 5))
plot_convs(image, conv_layer)
```



Example 4: Effect of No Padding

```
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(51, 51))
plot_convs(image, conv_layer, axis=True)
```

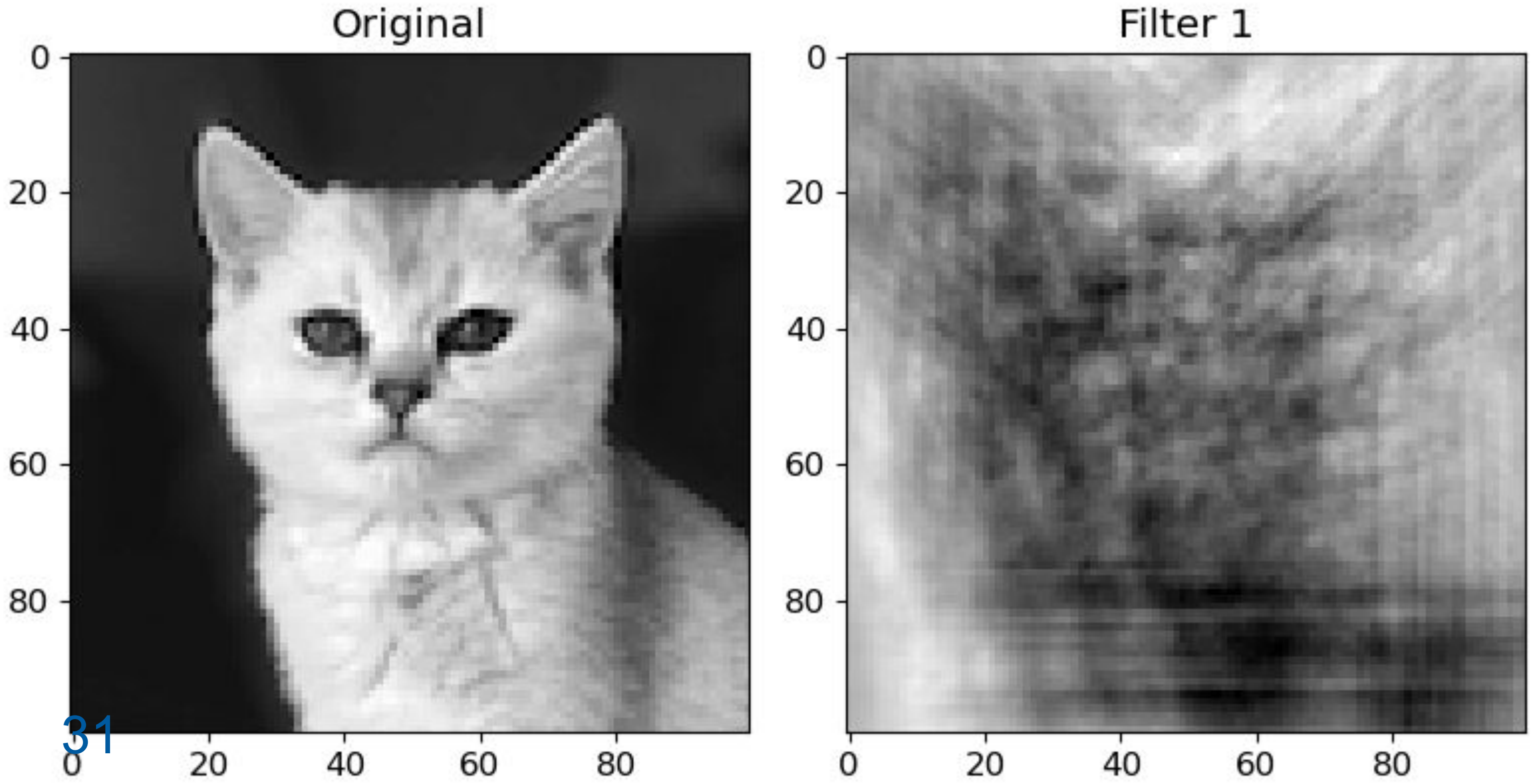


https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

- Output image becomes smaller
- Edges lost since kernel doesn’t cover full image
- Using a larger kernel shows this effect clearly

Example 5: With Padding

```
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(51, 51), padding=25)
plot_convs(image, conv_layer, axis=True)
```

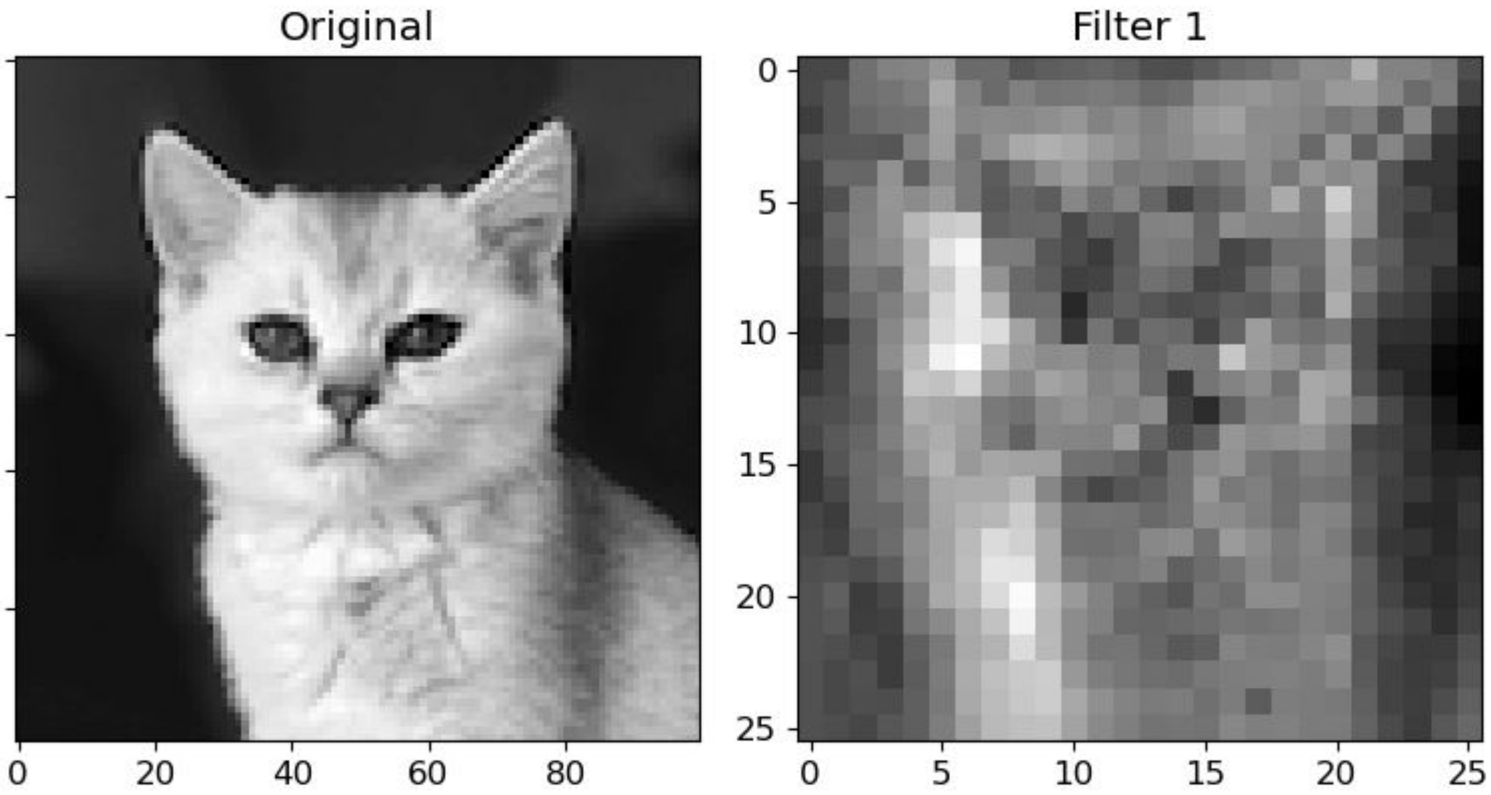


- Padding is added to the outside of the image to avoid this
- Setting padding = kernel_size // 2 will always generate an output of the same shape as the input.

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Example 6: With Strides

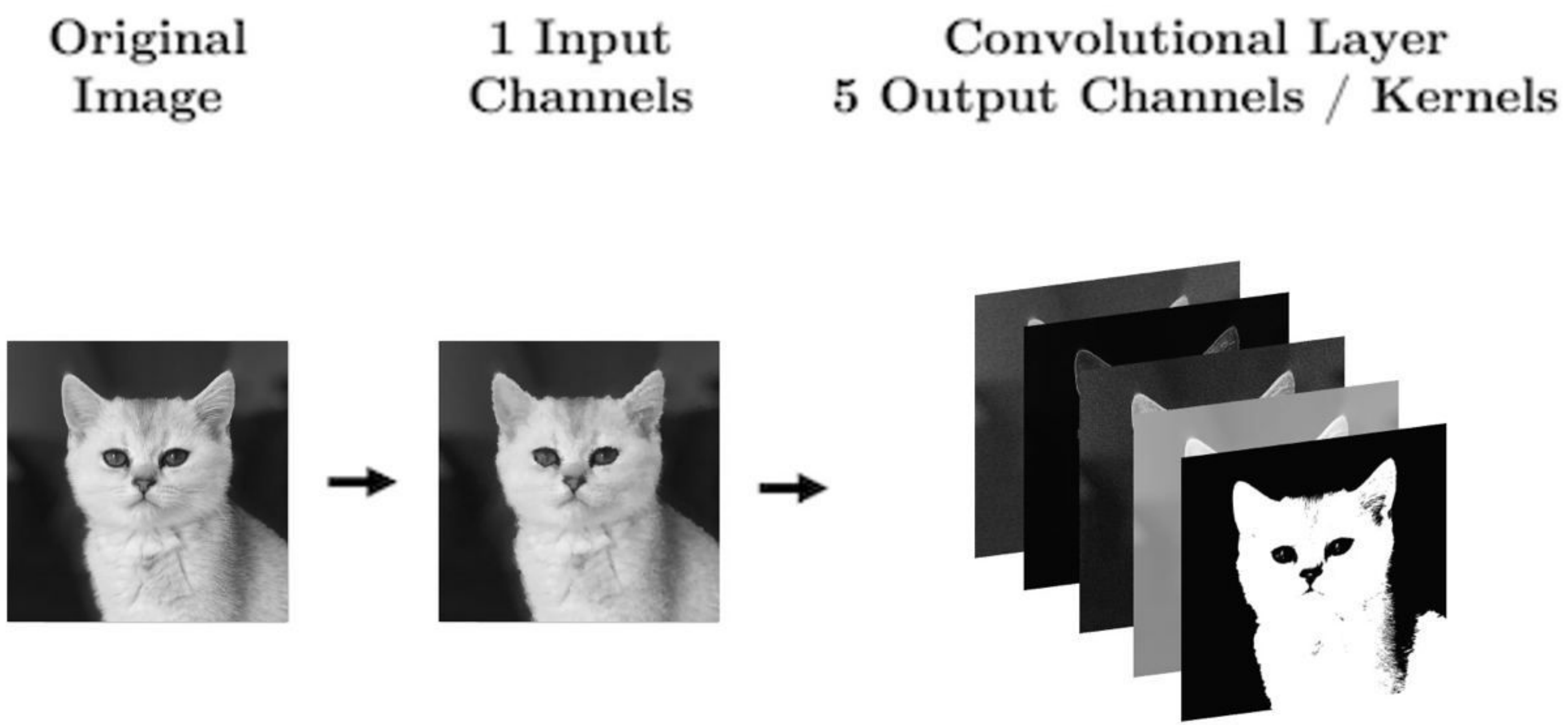
```
conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(25, 25), stride=3)
plot_convs(image, conv_layer, axis=True)
```



strides influence the size of the output

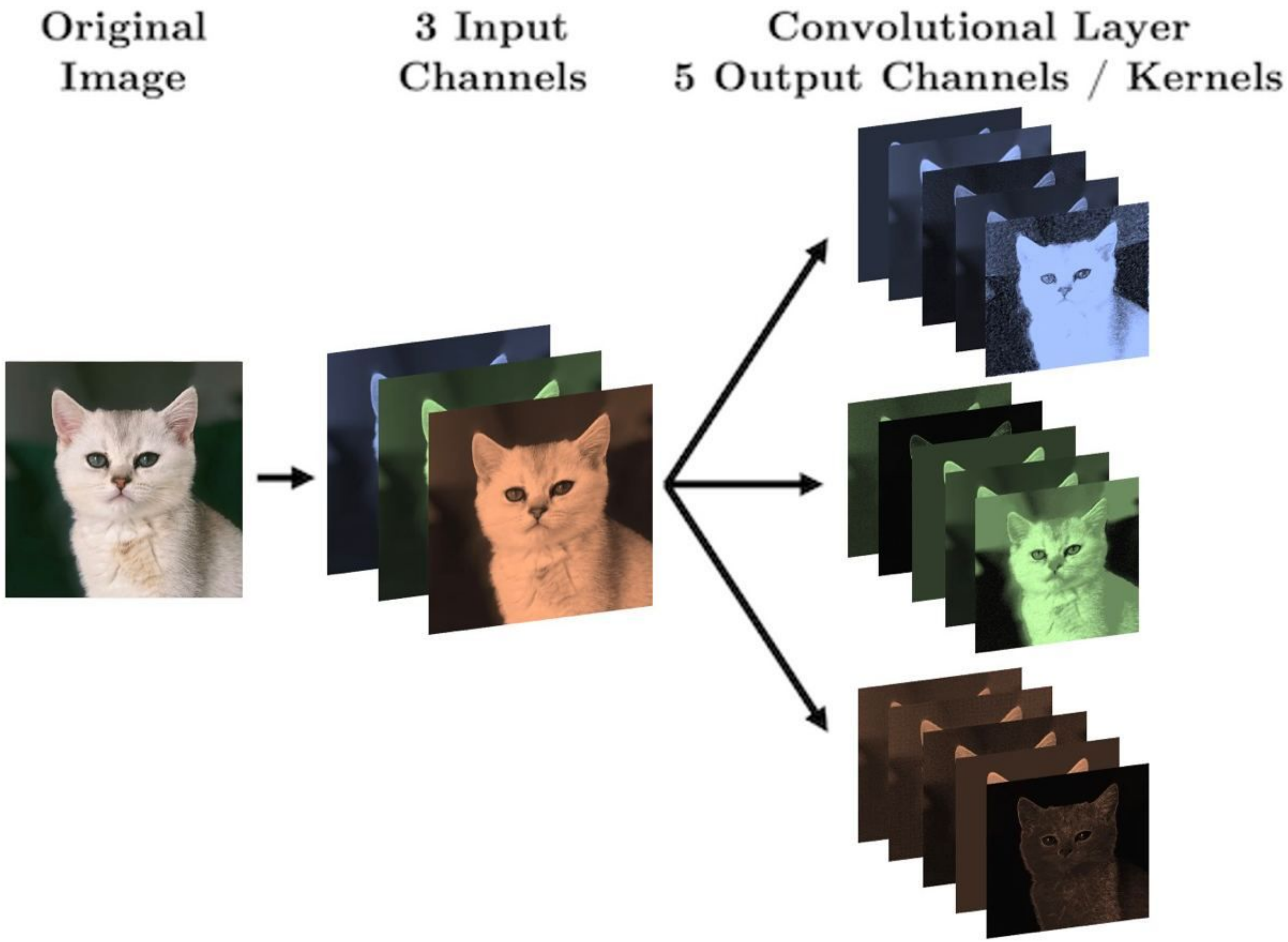
https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Features → Output Channels



https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Features → Output Channels



https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Let's make the simple CNN in PyTorch

20,000 parameters in that last layer, that's a lot of parameters

Is there a way we can reduce this?

```
class CNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.main = torch.nn.Sequential(

            torch.nn.Conv2d(in_channels=1, out_channels=3, kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),

            torch.nn.Conv2d(in_channels=3, out_channels=2, kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),

            torch.nn.Flatten(),
            torch.nn.Linear(20000, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```
model = CNN()
summary(model, (1, 100, 100));
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 1]                   --
|    └─Conv2d: 2-1                       [-1, 3, 100, 100]         30
|    └─ReLU: 2-2                         [-1, 3, 100, 100]         --
|    └─Conv2d: 2-3                       [-1, 2, 100, 100]         56
|    └─ReLU: 2-4                         [-1, 2, 100, 100]         --
|    └─Flatten: 2-5                      [-1, 20000]               --
|    └─Linear: 2-6                       [-1, 1]                   20,001
==========================================================================================
Total params: 20,087
Trainable params: 20,087
Non-trainable params: 0
Total mult-adds (M): 0.85
==========================================================================================
Input size (MB): 0.04
Forward/backward pass size (MB): 0.38
Params size (MB): 0.08
Estimated Total Size (MB): 0.50
==========================================================================================
```

please see appendix for more better understanding

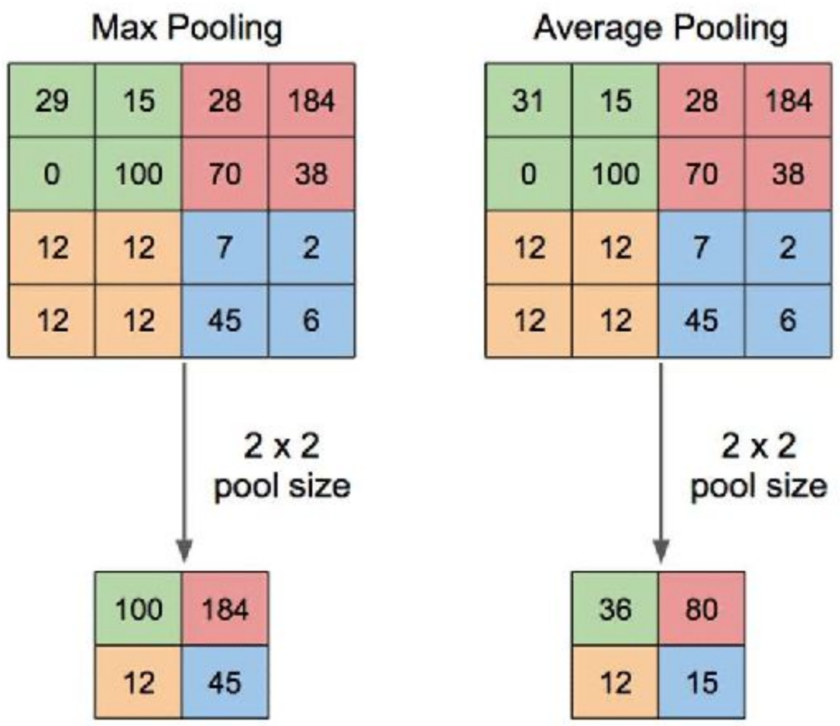
Pooling Layers in CNN

- ❑ Used to **reduce dimensionality** of feature maps
  → fewer parameters after flattening (via torch.nn.Flatten())

- ❑ Common pooling methods:
  - **Max Pooling**
  - **Average Pooling**

- ❑ **Max Pooling** often performs better
  → captures the **sharpest and most important features** of the image

Example: applying **Max Pooling** to highlight key patterns in a transformed image



https://www.researchgate.net/publication/333593451_Application_of_Transfer_Learning_Using_Convolutional_Neural_Network_Method_for_Early_Detection_of_Terry's_Nail

Implementing pooling with torch.nn.MaxPool2d()

reduce the number of parameters

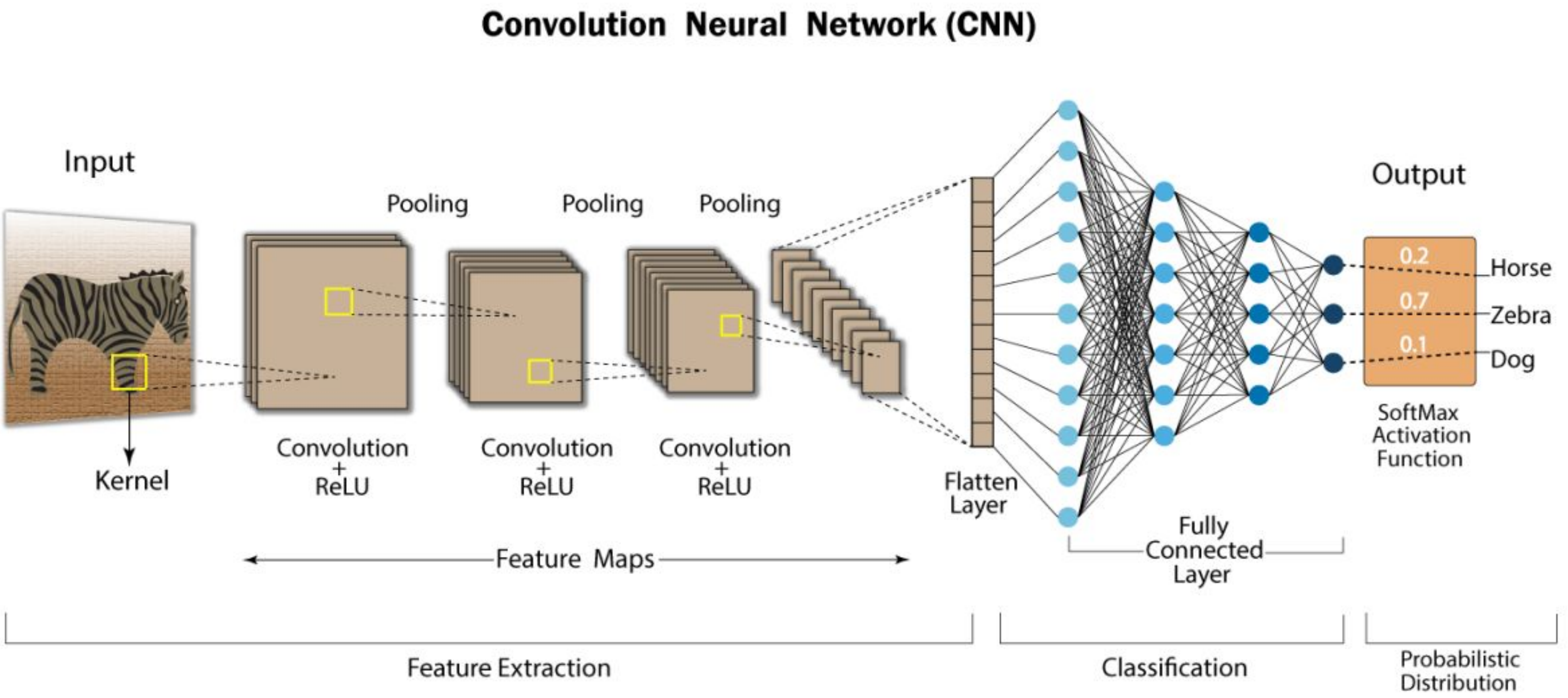
```
class CNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.main = torch.nn.Sequential(

            torch.nn.Conv2d(in_channels=1, out_channels=3, kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d((2, 2)),

            torch.nn.Conv2d(in_channels=3, out_channels=2, kernel_size=(3, 3), padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d((2, 2)),

            torch.nn.Flatten(),
            torch.nn.Linear(1250, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```
model = CNN()
summary(model, (1, 100, 100));
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 1]                   --
|    └─Conv2d: 2-1                       [-1, 3, 100, 100]         30
|    └─ReLU: 2-2                         [-1, 3, 100, 100]         --
|    └─MaxPool2d: 2-3                    [-1, 3, 50, 50]           --
|    └─Conv2d: 2-4                       [-1, 2, 50, 50]           56
|    └─ReLU: 2-5                         [-1, 2, 50, 50]           --
|    └─MaxPool2d: 2-6                    [-1, 2, 25, 25]           --
|    └─Flatten: 2-7                      [-1, 1250]                --
|    └─Linear: 2-8                       [-1, 1]                   1,251
==========================================================================================
Total params: 1,337
Trainable params: 1,337
Non-trainable params: 0
Total mult-adds (M): 0.41
==========================================================================================
Input size (MB): 0.04
Forward/backward pass size (MB): 0.27
Params size (MB): 0.01
Estimated Total Size (MB): 0.31
==========================================================================================
```

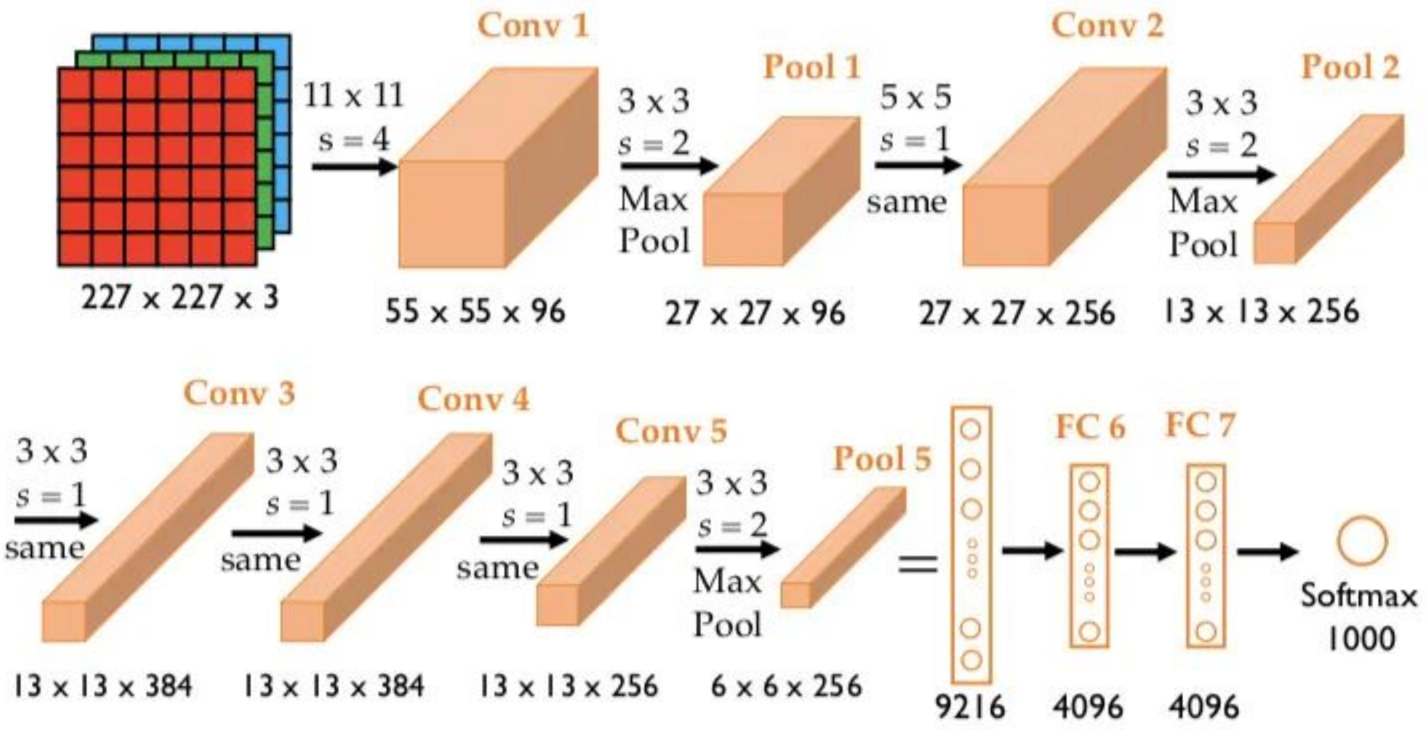
We reduced that last layer to 1,251 parameters (~16 times smaller than the original number)

Example: CNN Architecture



https://medium.com/@sushmita2310/understanding-convolutional-neural-networks-cnns-for-beginners-e85ad21fe432

AlexNet Architecture Overview

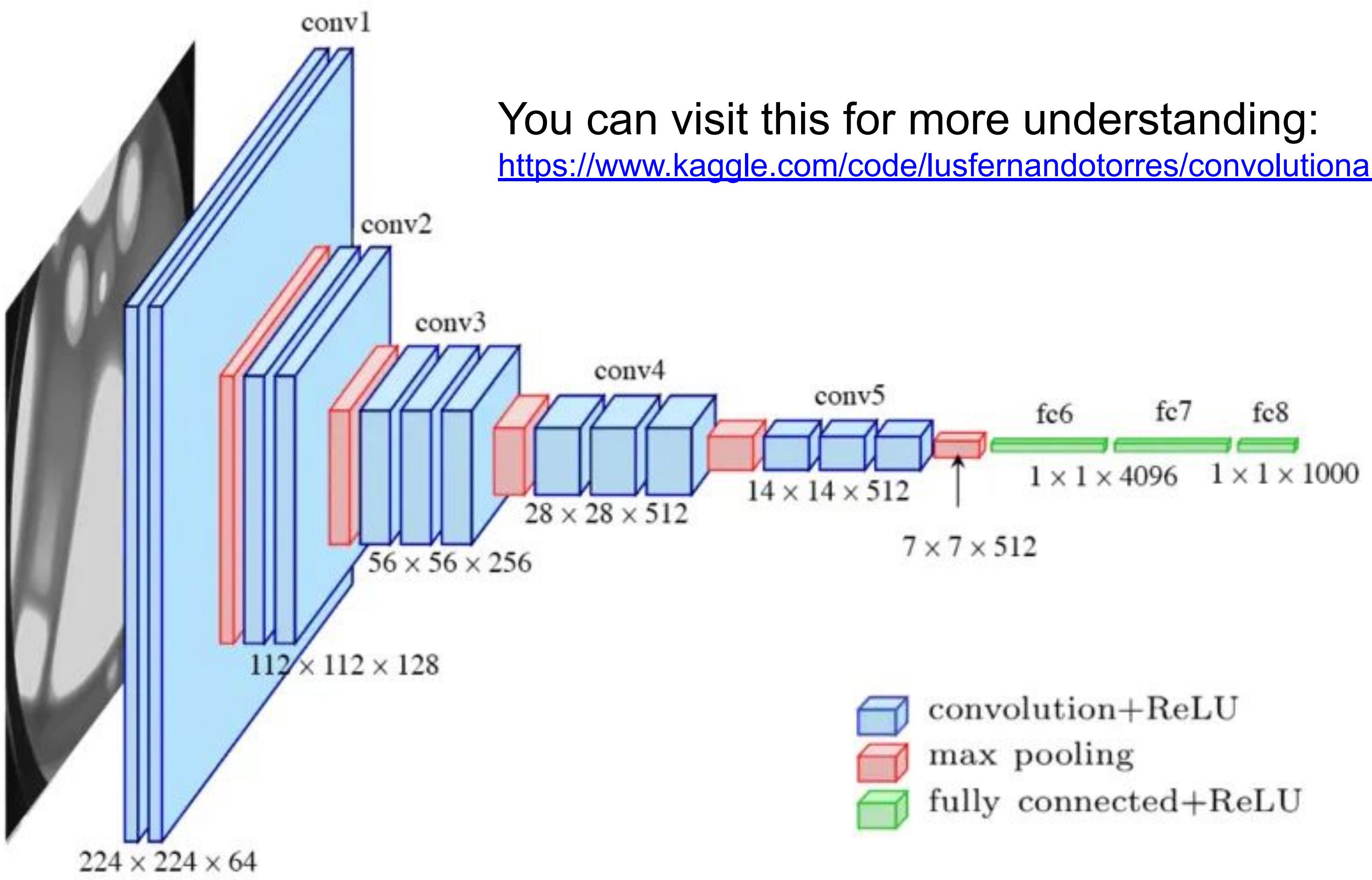


First CNN to use GPU for faster training performance

- **5 Convolutional layers** → extract image features
- **3 Max-Pooling layers** → reduce spatial dimensions
- **2 Fully Connected layers** → combine learned features
- **1 Softmax layer** → output final class probabilities

- Each Conv layer uses **ReLU activation** (introduces non-linearity)
- **Input size: ~227 × 227 × 3** (often referred to as 224 × 224 × 3)

VGG-Net Architecture



You can visit this for more understanding:
https://www.kaggle.com/code/lusfernandotorres/convolutional-neural-network-from-scratch

https://medium.com/latinxinai/convolutional-neural-network-from-scratch-6b1c856e1c07

Thank You Any Questions?

Time for exercise

Appendix: Convolution using pytorch for gray scale image

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

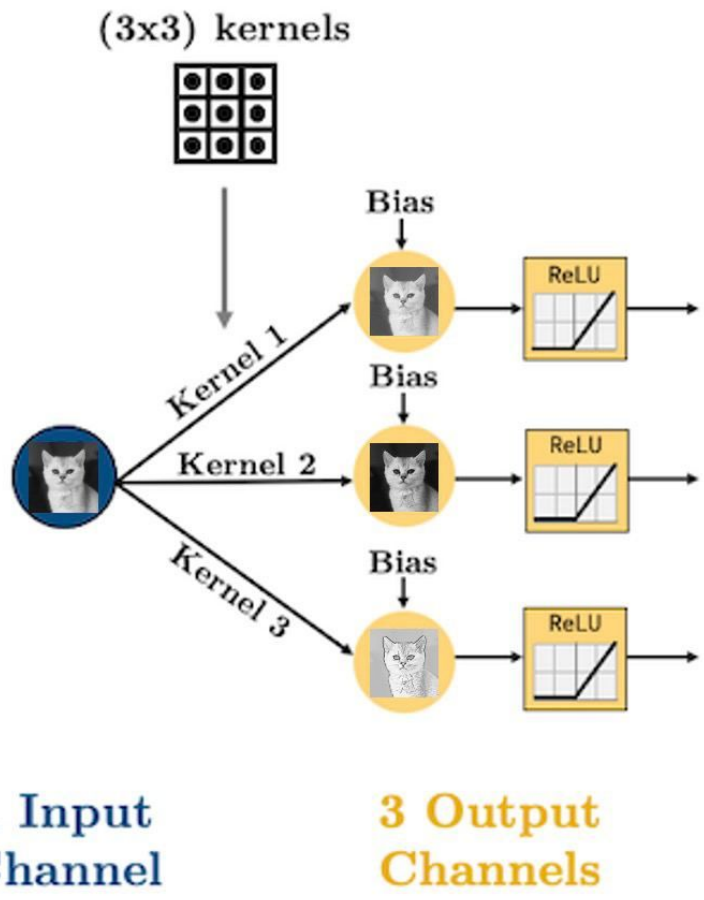
```
nn.Conv2d(
        in_channels=1,
        out_channels=3,
        kernel_size=(3,3)
)
```

Total number of parameters:

- 1 input channel
- 3 output channels
- (3 x 3) filters

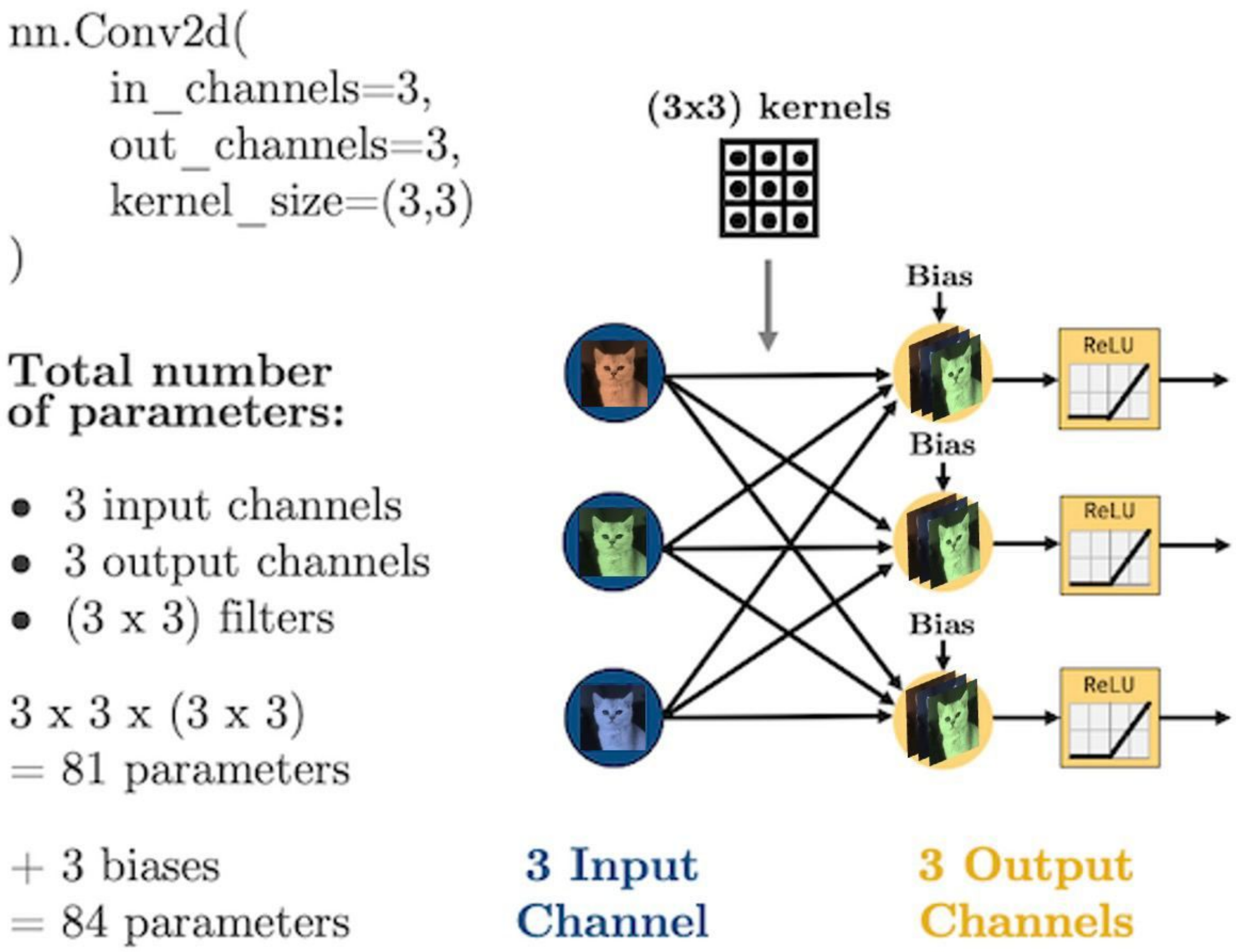
1 x 3 x (3 x 3)
= 27 parameters

+ 3 biases
= 30 parameters



Appendix: Convolution using pytorch for gray scale image

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html



Appendix: Understanding Tensor Dimensions in PyTorch

PyTorch needs data in a **very specific shape** for convolution operations.

Image Tensor Shape

All images (inputs or outputs of convolution) are stored as **4D tensors** in PyTorch, with this shape:

$$(\text{batch\_size}, \text{n\_channels}, \text{height}, \text{width})$$

So for one grayscale image of 100×100, shape should be:

$$[1, 1, 100, 100]$$

Dimension	Meaning	Example
`batch_size`	Number of images processed together	1 (if just one image)
`n_channels`	Number of color channels	1 (grayscale) or 3 (RGB)
`height`	Image height (pixels)	100
`width`	Image width (pixels)	100

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Appendix: Kernel (Filter) Tensor Shape

Each convolutional layer also has a **4D tensor** for its filters:

$$(\text{n\_kernels}, \text{n\_channels}, \text{kernel\_height}, \text{kernel\_width})$$

Example: `conv_layer = torch.nn.Conv2d(1, 1, kernel_size=(5,5))`

means:

- 1 input channel (grayscale)
- 1 output channel (so 1 kernel)
- Each kernel is 5×5 in size

Dimension	Meaning
`n_kernels`	Number of filters (output feature maps)
`n_channels`	Must match number of input channels
`kernel_height`, `kernel_width`	Filter size (e.g., 3×3, 5×5)

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Appendix: Why the Shape Error Happened

First loaded the image:

```
image.shape
torch.Size([100, 100, 4])
```

That's the **NumPy-style format** — `(height, width, channels)` (note the channel last).

But PyTorch expects **channels first**, like `(channels, height, width)` — and with an extra dimension for batch.

So we have to reshape it like this:

```
image = image[None, None, :]
image.shape
torch.Size([1, 1, 100, 100])
```

This means:

- `1` batch
- `1` channel
- `100×100` pixels

Now PyTorch knows exactly how to apply the convolution.

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Appendix: Why Output Is Smaller

- After convolution:

```
conv_out.shape
torch.Size([1, 1, 96, 96])
```

- It shrinks because the kernel (5×5) can't fit around edge pixels — this is **"no padding"**.
- Formula for output size (*with stride=1, no padding*):

$$\text{Output size} = \text{Input size} - \text{Kernel size} + 1$$

So:

$$100 - 5 + 1 = 96$$

https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Appendix: Visualizing the Result

To plot it with Matplotlib, a few clean-up steps are needed:

1. **Detach:**
   .detach() removes the tensor from PyTorch's *computational graph* (so no gradients are tracked).

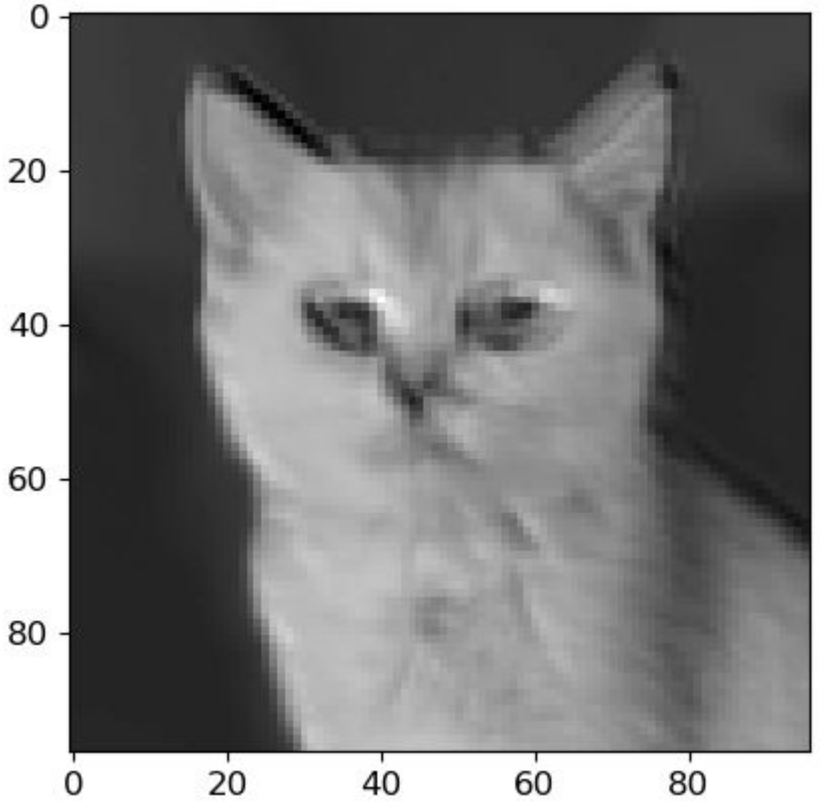
2. **Squeeze:**
   .squeeze() removes any dimensions of size 1 (batch and channel), leaving only the 2D image.

3. **Plot:**
   Then plt.imshow() can be used to show the image.

Example:

```
plt.imshow(conv_out.detach().squeeze(), cmap='gray')
```

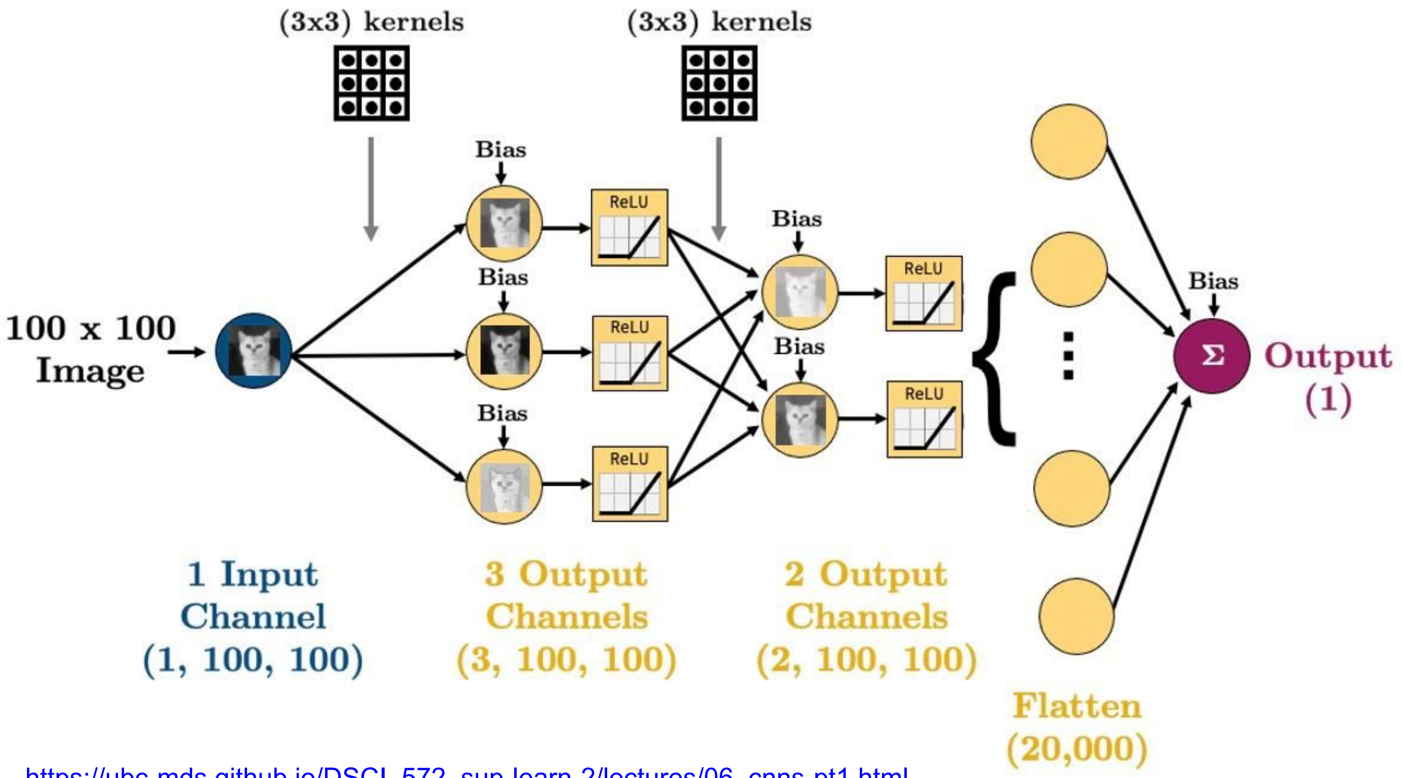
Now the shape is [96, 96] — which Matplotlib understands.



https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html

Appendix: Flattening

- With our convolutional layers, we're basically just passing images through the network
- But we're going to eventually want to do some regression or classification
- That means that by the end of our network, we are going to need to torch.nn.Flatten() our images:



https://ubc-mds.github.io/DSCI_572_sup-learn-2/lectures/06_cnns-pt1.html