Tutorial 3: Convolutional Neural Networks

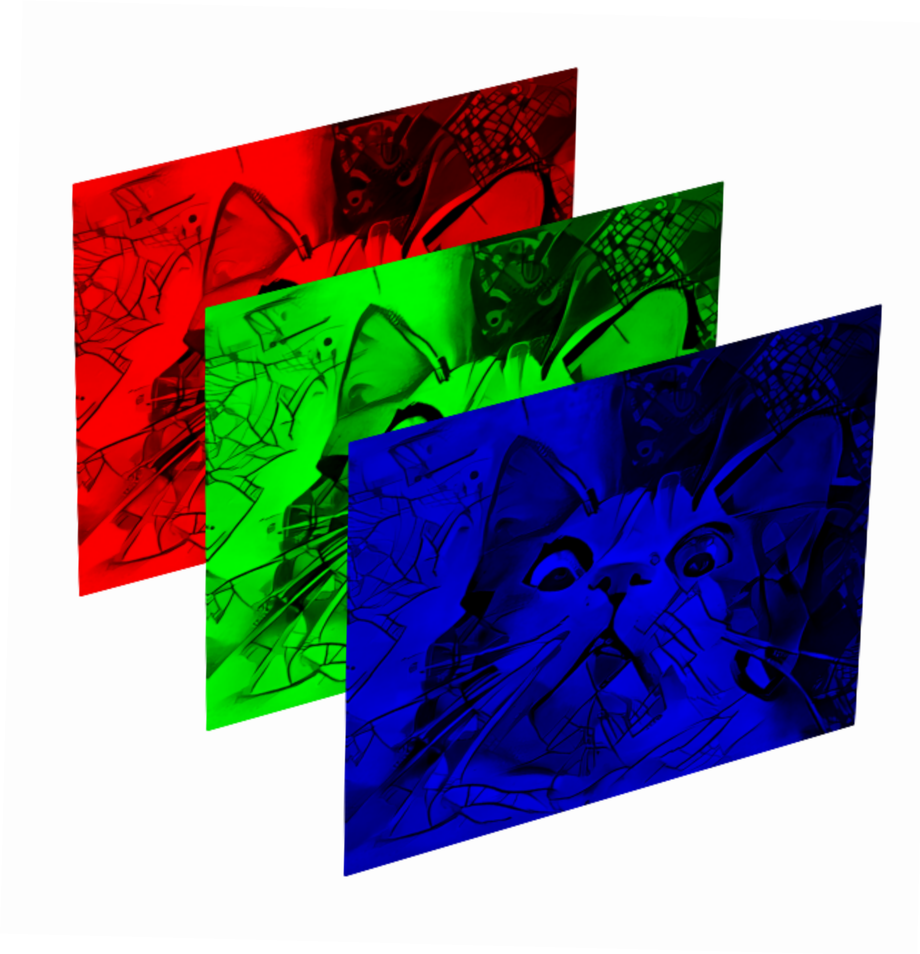
Practical Deep Learning for Science
10 April, 2025

מכון ויצמן למדע
WEIZMANN INSTITUTE OF SCIENCE

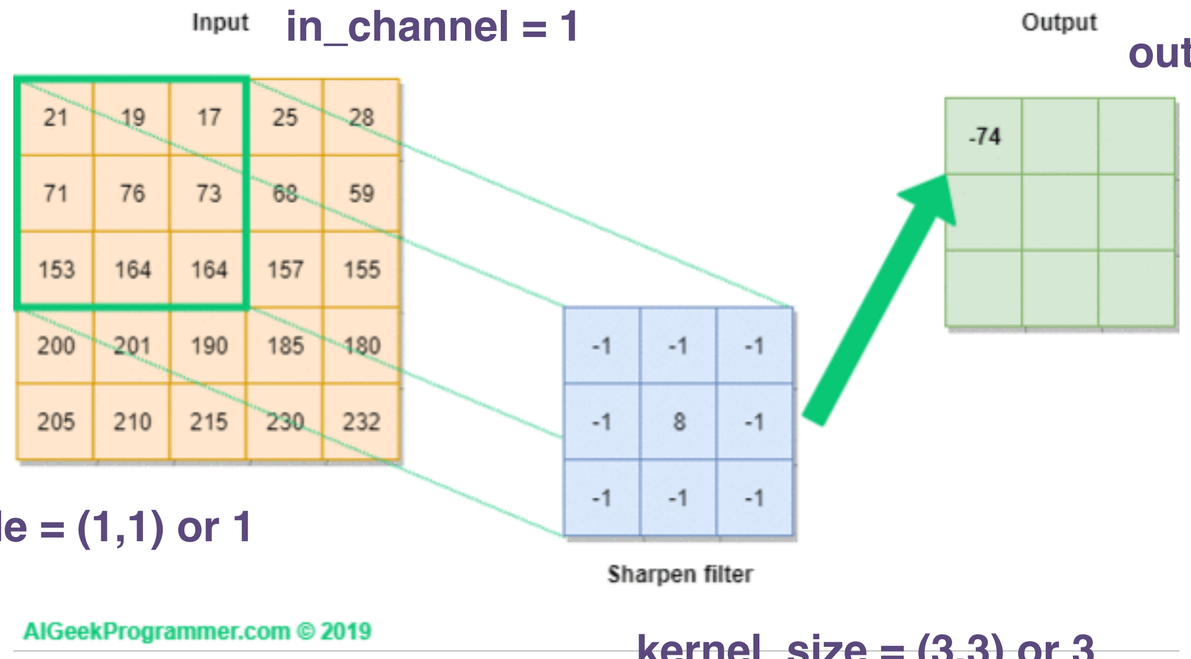
- Nilotpal

CNN Vocabulary

Channels



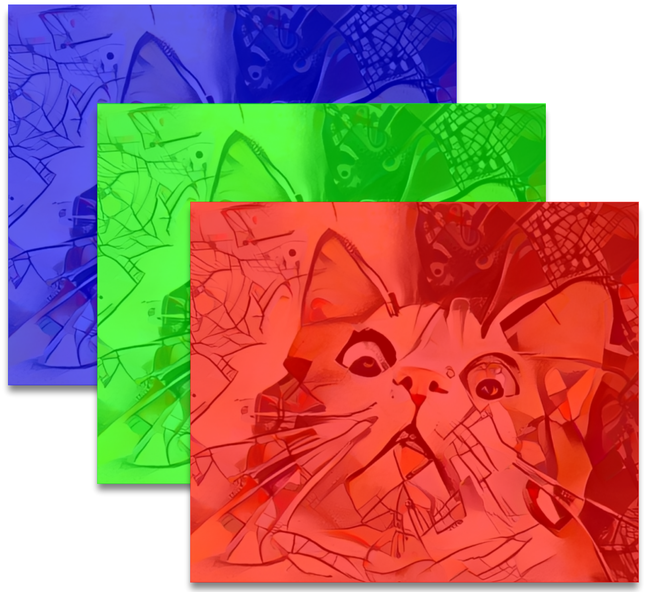
Not necessarily always 3



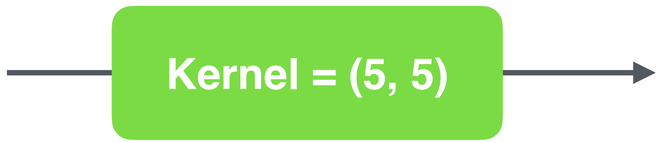
- in_channels
- out_channels
- kernel_size
- stride
- padding

padding = ‘valid’ or 0

Convolution with multiple channels



3 channels

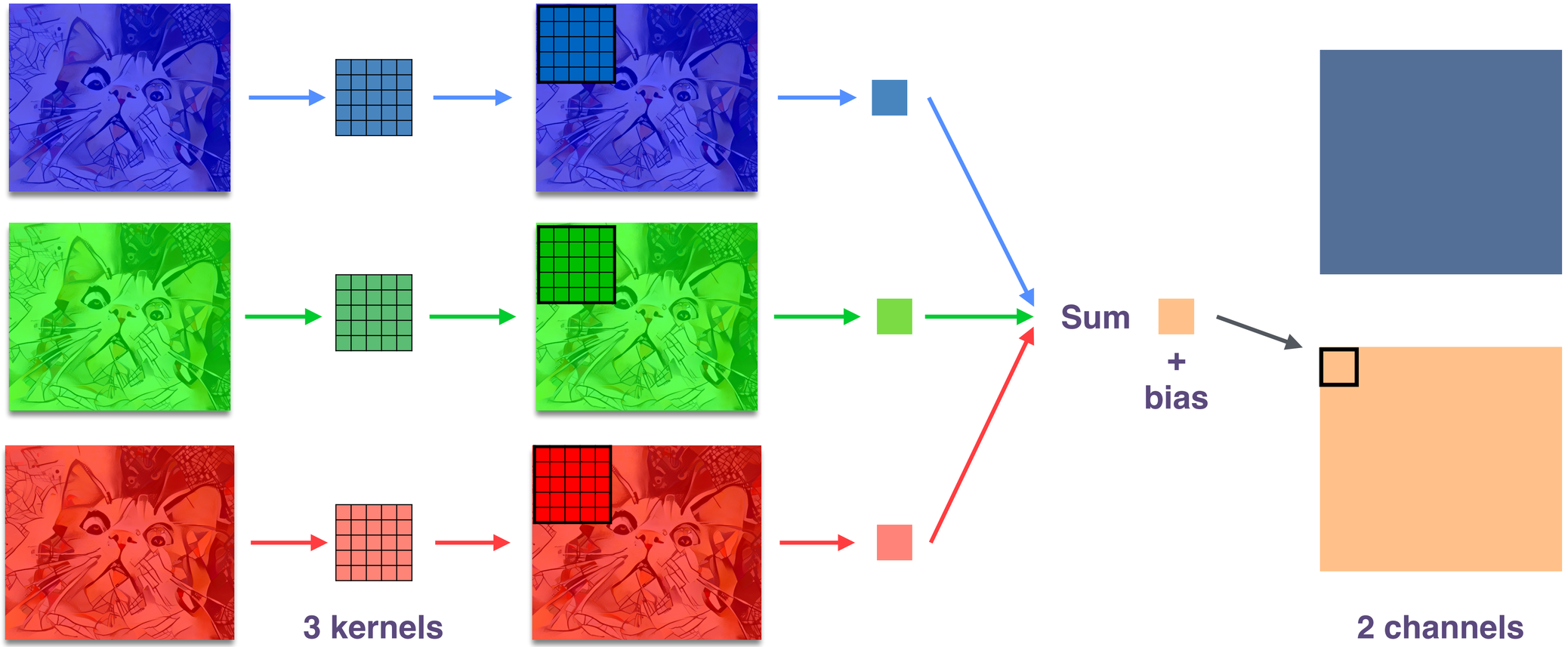


2 channels

Ignore padding, stride etc.

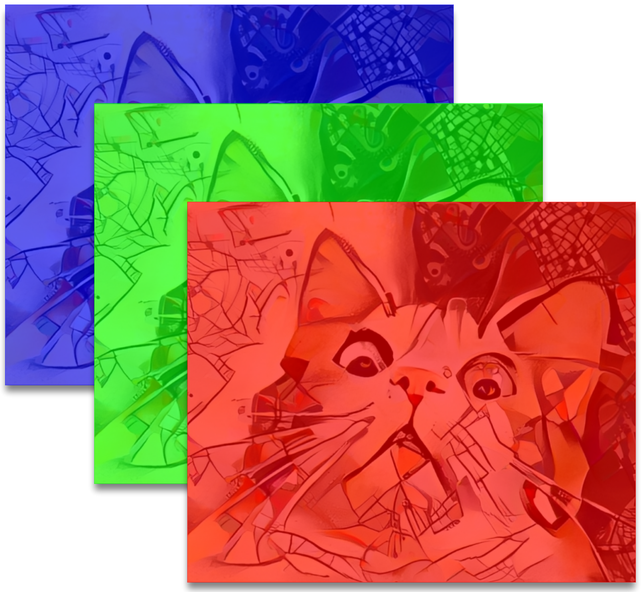
Convolution with multiple channels

Think in terms of one output pixel
Cause we'll just move the kernel around for the full image



*For **one channel**, we have **3** kernels of size **5x5***

Convolution with multiple channels



3 channels

Kernel matrix shape

(2, 3, 5, 5)

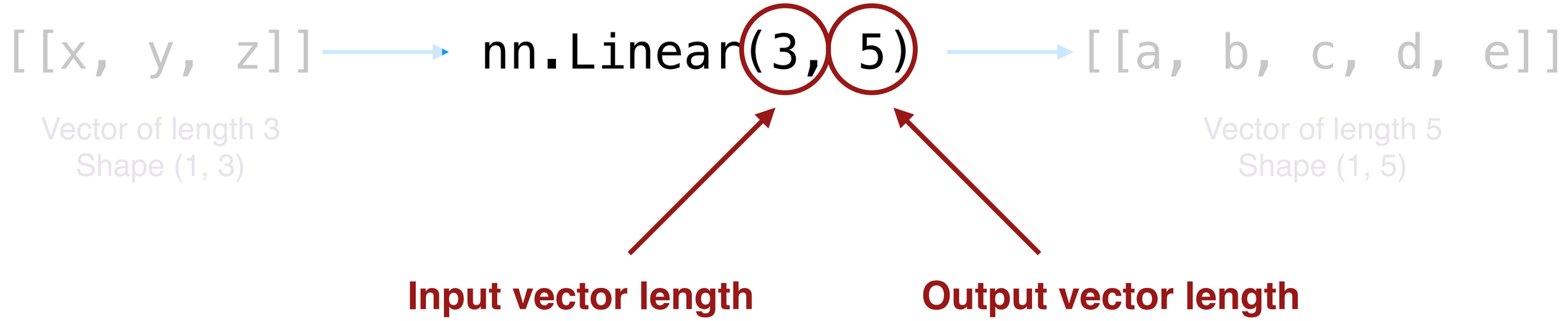
Output channel

Input channel

Kernel size

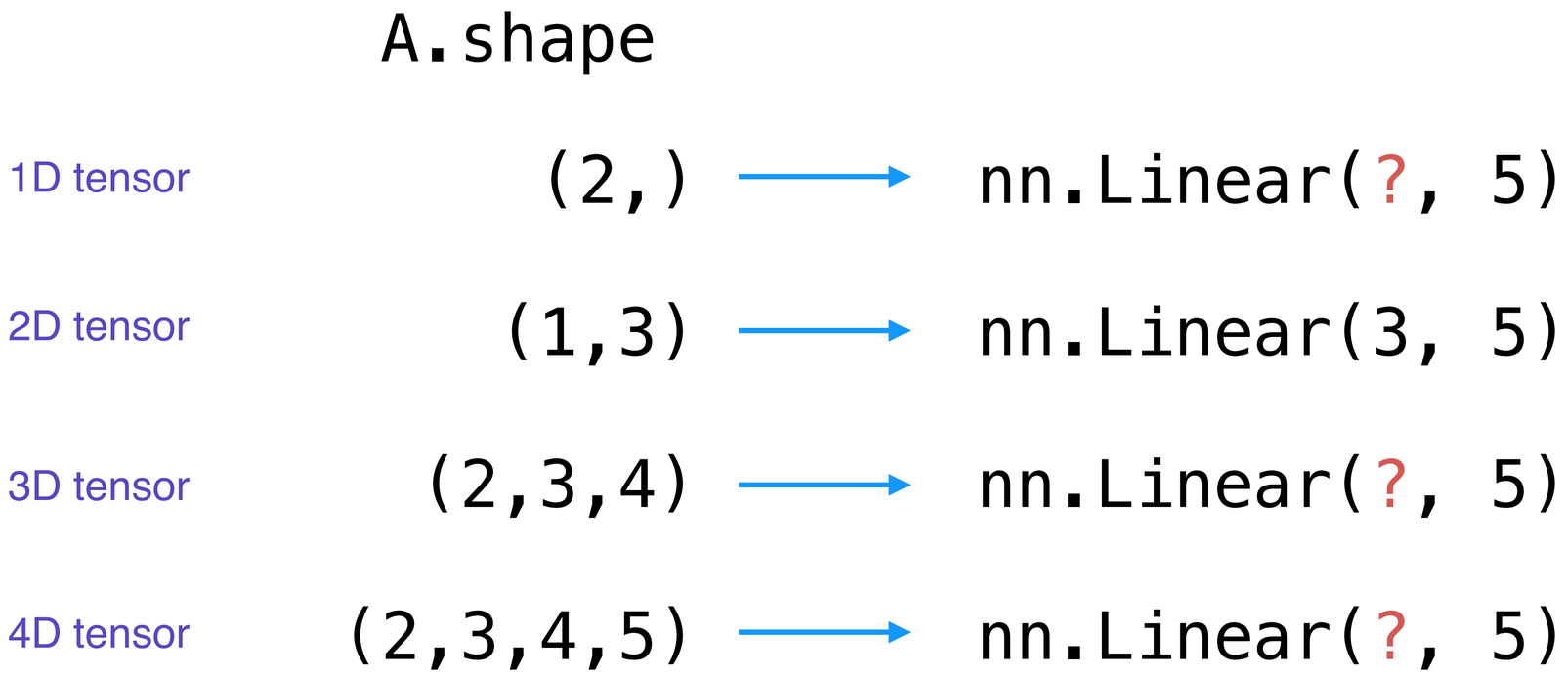
2 channels

NHWC vs NCHW



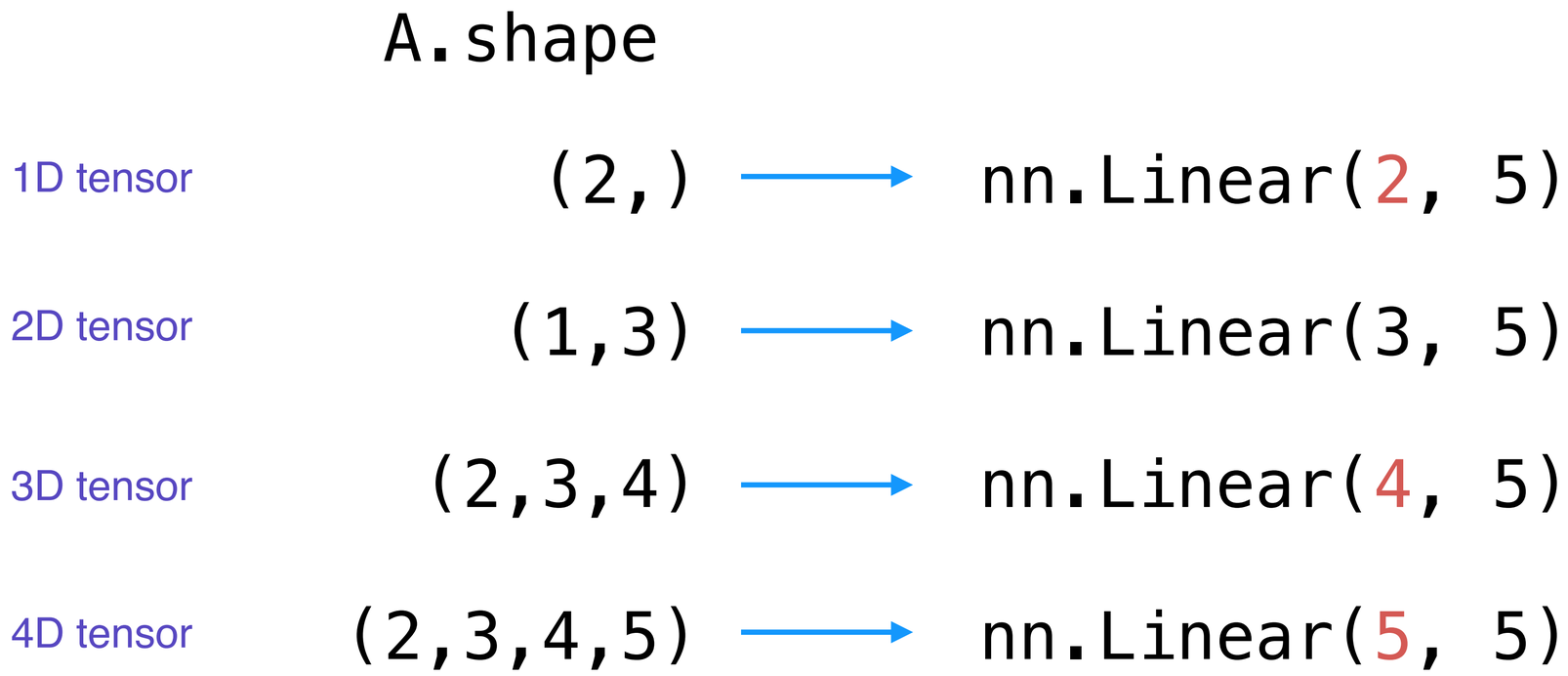
$$\underset{1\times 3}{(x,y,z)}\underset{3\times 5}{\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{pmatrix}} + \underset{1\times 5}{\begin{pmatrix} b_1 & b_2 & b_3 & b_4 & b_5 \end{pmatrix}} = \underset{1\times 5}{\begin{pmatrix} a & b & c & d & e \end{pmatrix}}$$

A is a torch.Tensor(). Previously we saw a 2D tensor



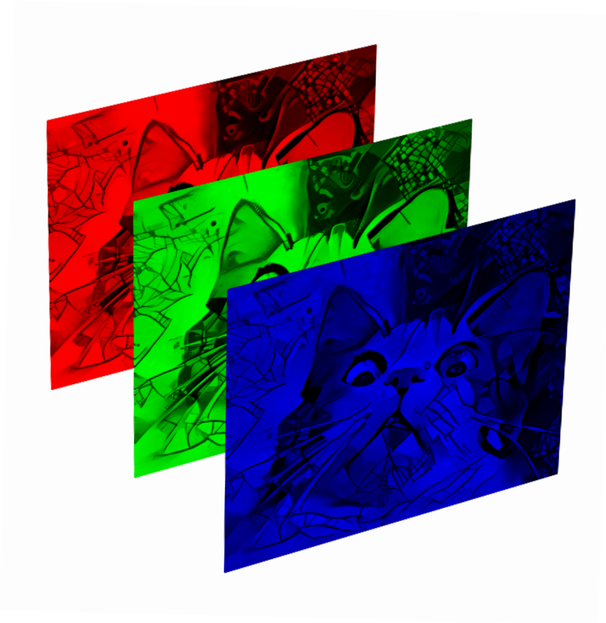
A is a torch.Tensor(). Previously we saw a 2D tensor

In torch, It'll always act on the last dimension

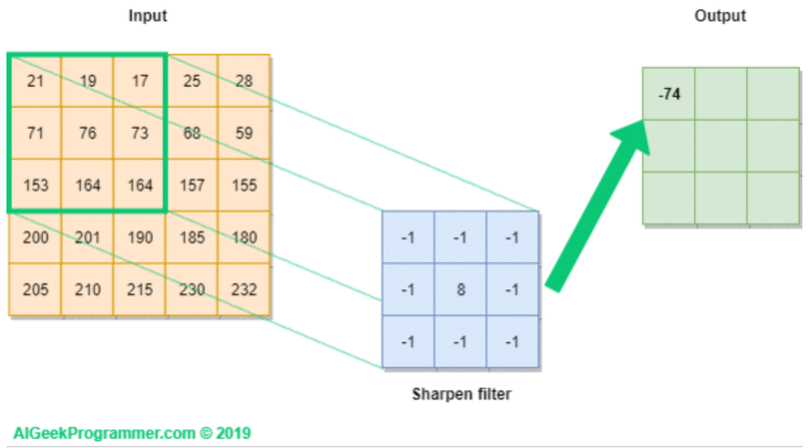


What about Conv2D

- In general, images will have shape
  - ➡ **(batch_size, H, W, 3)** (3 channels - RGB)
- Kernel will have shape, say
  - ➡ **(Kernel_H, Kernel_W)**



- With this setup, we will act the kernel on **(W, 3)**, but it should act on **(H, W)**
- So, we need to change the image format a bit



NCHW vs NHWC

- **N** - number of example in the batch
- **C** - channel number
- **H** - height
- **W** - width

- Torch expects data to be in **(N,C,H,W)** format

Permute and reshape are two completely different operations!

- An **RGB** image of size **100x50**

What we want

```
>>> img_tensor.shape
torch.Size([1, 3, 100, 50])
```

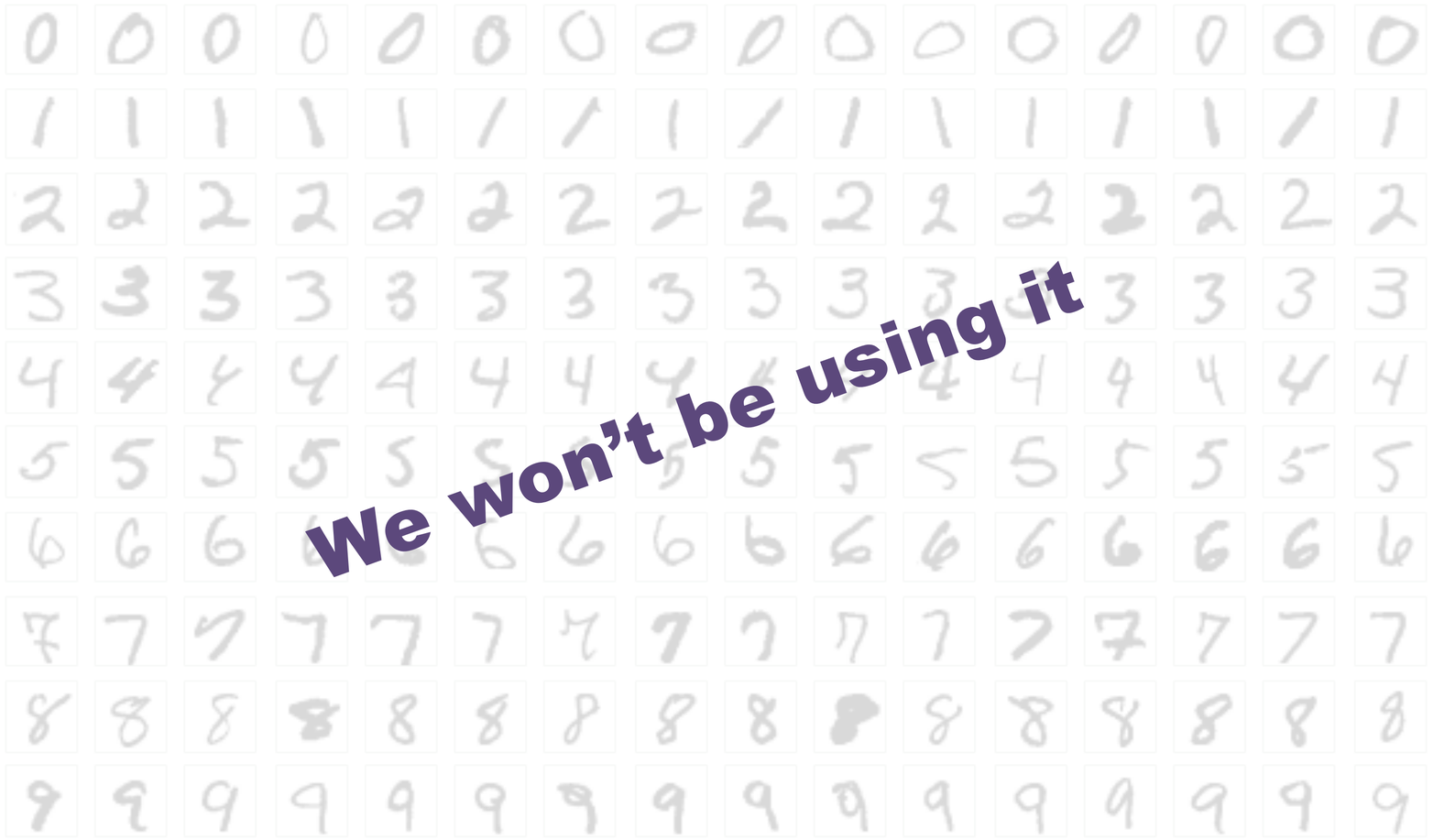
What we do not want

```
>>> img_tensor.shape
torch.Size([1, 100, 50, 3])
```

Fix

```
>>> img_tensor = img_tensor.permute(0,3,1,2)
>>> img_tensor.shape
torch.Size([1, 3, 100, 50])
```

Today’s dataset



We won’t be using it

MNIST

- Handwritten digits
- 28x28
- Grayscale (1 channel)

- Pretty old (1994)
- Quite simple

Fashion-MNIST

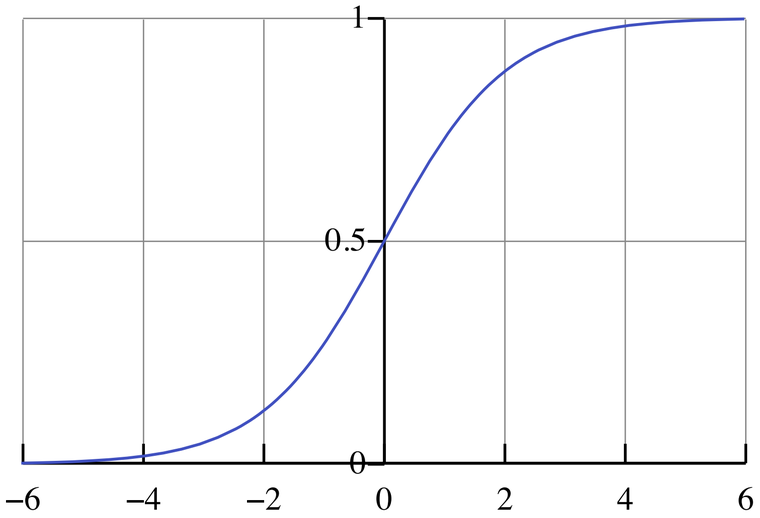
- 28 x 28
- grayscale images (1 channel)
- 10 classes
- Slightly more complex than MNIST

Activations and losses

Binary classifications

- **Class 0** vs **Class 1**
  - NN Prediction = **probability of class 0** (p)
  - *Probability of class 1* = $1 - p$
    - No need to predict this

- Activation: `torch.nn.Sigmoid()`



Binary classifications: Loss

- **What we want -**
  - End the network with `Linear()` → `Sigmoid()`
  - Compute the loss using -
    - `torch.nn.BCELoss(pred_prob, target)`

- **More common practice -**
  - End the network with `Linear()`
  - Compute the loss using -
    - `torch.nn.BCEWithLogitLoss(pred_logit, target)`
    - Internally applies `Sigmoid()`
    - Numerically more stable

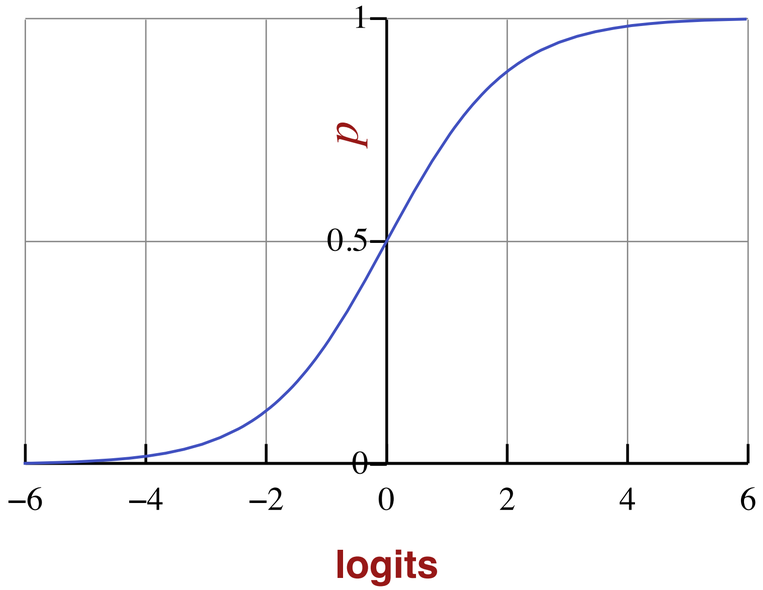
Logit

↓

Sigmoid

↓

Probability

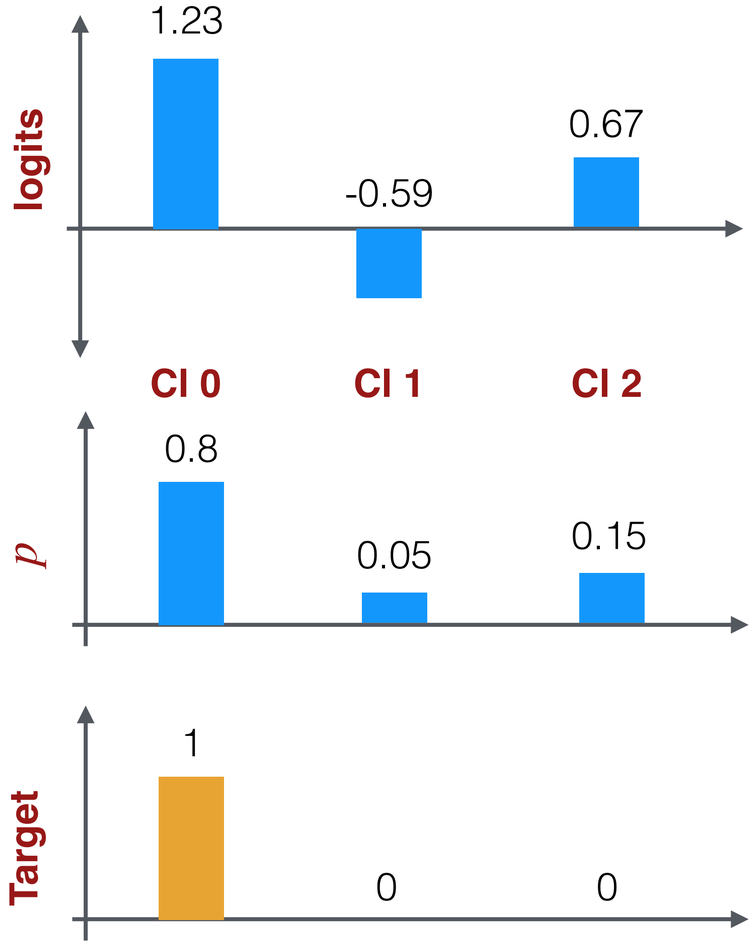


- **During inference -**
  - $p > 0.5 \implies$ class 0
  - $logit > 0 \implies$ class 0

Multi-class classifications

- **Option one -**
  - End the network with `Linear()` → `Softmax()`
  - Compute the loss using -
    - `torch.nn.NLLLoss(pred_prob, target)`

$$p_i = \frac{e^{x_i}}{\sum_i e^{x_i}}$$



- **More common practice -**
  - End the network with `Linear()`
  - Compute the loss using -
    - `torch.nn.CrossEntropy(pred_logit, target)`
    - Internally applies `Softmax()`
    - Numerically more stable

- **During inference -**
  - Class = `argmax(prob)`
  - Class = `argmax(logit)`