# Lecture 5

# 13.10.2025

# Today's plan and announcements

- Review: last week's summary
- Convolutional neural networks

- Quiz 1: 15.10 during the exercise hour. They are 30 minutes and no aid is allowed (no books/notes, no electronics).
- Quizzes are optional
- Python exercise on neural nets with solutions are posted. They help you understand the concepts and see them in practice.

# Review: neural networks exercise

- You have constructed a neural network for a regression problem with $x \in \mathbb{R}^2, y \in \mathbb{R}$. Your network has 2 hidden layers and 4 nodes per layer.
  1. Draw the network
  2. How many weights and biases need to be determined per layer and in total?
  3. Let the activation function of each layer be given by $\tanh(\,.\,)$. Write the prediction outcome for a given point $x^{test}$
  4. Now, repeat the same exercise but for a classification problem with $x \in \mathbb{R}^2, y \in \{1,2,3,4,5,6\}$.

# Review: neural network exercise solution

2. weights & biases

input layer → hidden layer 1

2 × 4 weights, 4 biases

hidden layer 1 → hidden layer 2

4 × 4 weights, 4 biases

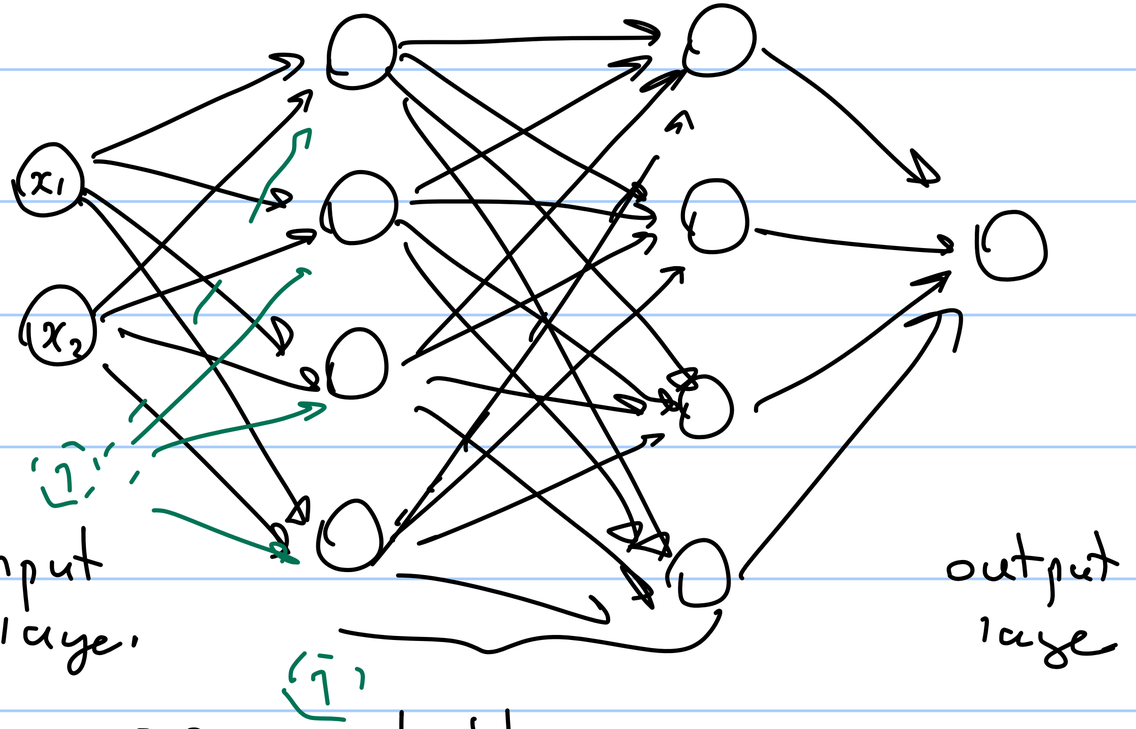
hidden layer 2 → output layer

4 × 1 weights, 1 bias

# of parameters = 37

3. $\hat{y}(x) = W^{[3]} \tanh\left( W^{[2]} \tanh\left( W^{[1]} x + b^{[1]} \right) + b^{[2]} \right) + b^{[3]}$

Network



# Review: neural network exercise solution

2. weights & biases

input to hidden layer 1

2x 4 weights, 4 biases

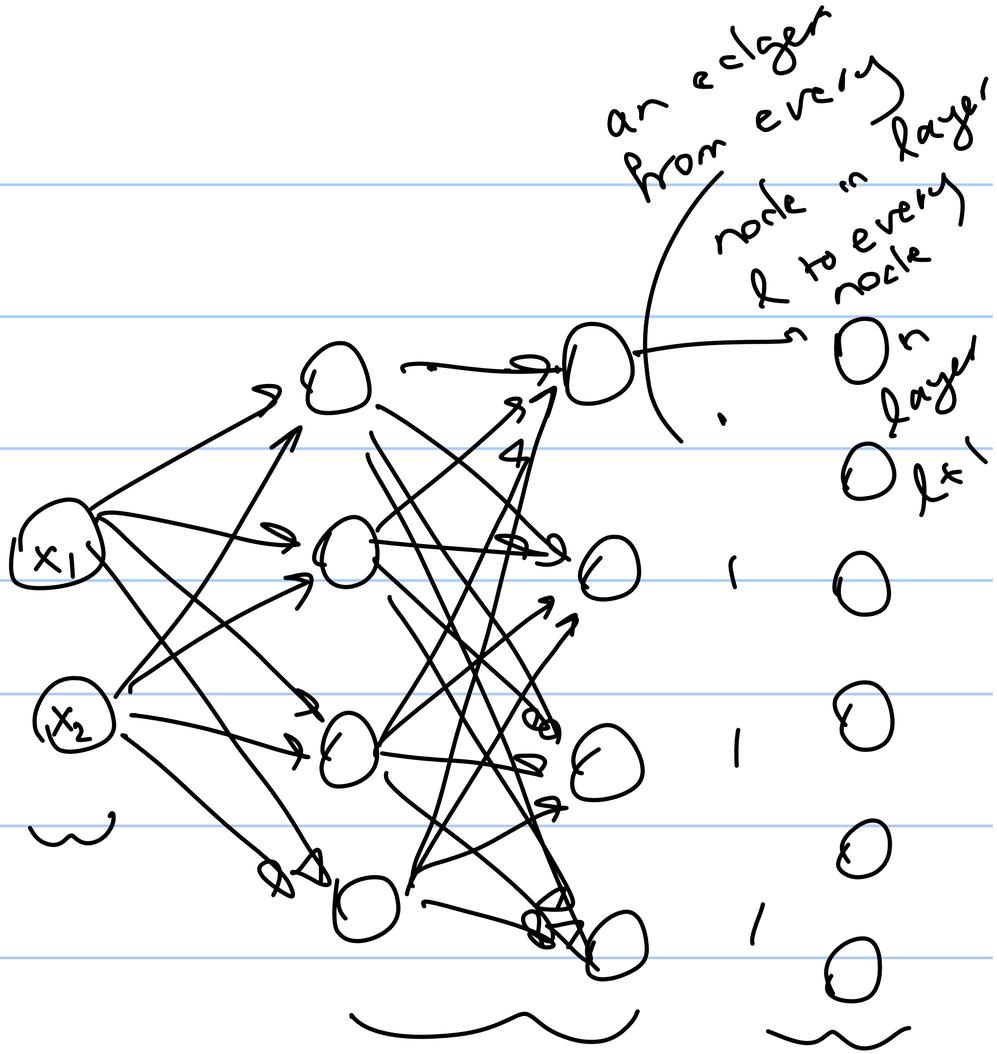
hidden layer 1 to hidden layer 2

4x4 weights, 4 biases

hidden layer 2 to output layer

4x6 weights, 6 biases

# of parameters: 62




$$\hat{y}_{3\times 1} = \text{Softmax}\left(W^{[3]}\tanh\left(W^{[2]}\tanh\left(W^{[1]}x + b^{[1]}\right) + b^{[2]}\right) + b^{[3]}\right)$$

# Recall: Determining the neural network predictor

- Choose the neural network architecture

Number of layers $L$, number of nodes per layer $n_l$, for layer $l$, activation function at each layer

- Choose a loss function, let $\theta$ denote all neural network parameters
  - For classification $$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{K}\mathbf{1}_{\{y^{(i)}=c\}}\log\left(\frac{\exp(z_c^i)}{\sum_{j=1}^{K}\exp(z_j^i)}\right)$$
  - For regression $J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^i - y^i)^2$

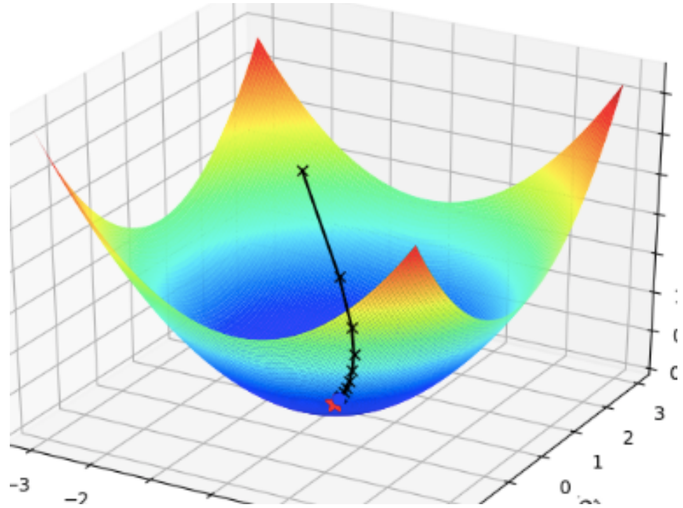
- Minimize the loss function given the training data $\{x^i, y^i\}_{i=1}^N$

# Training neural networks with gradient descent

Consider a regression problem

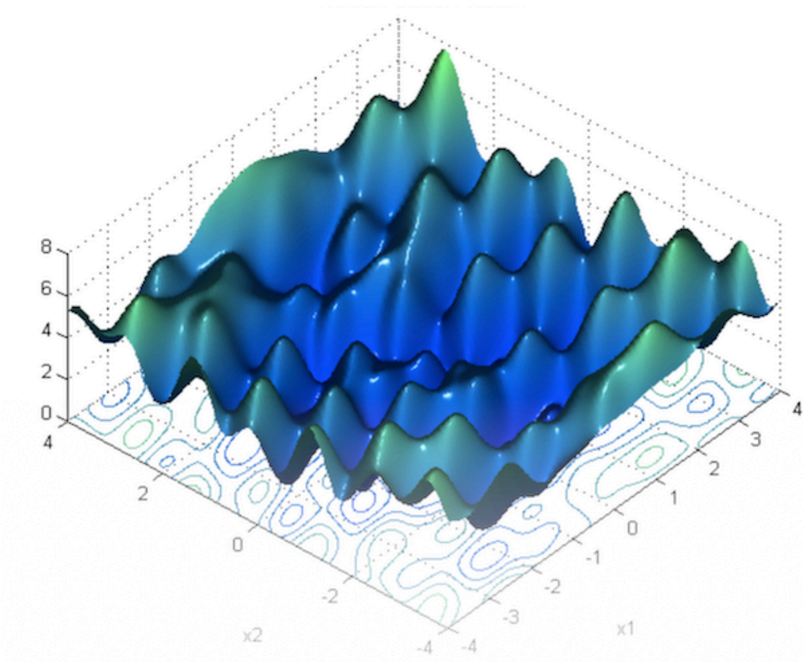
$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^i - y^i)^2, \theta = \{\mathbf{W}^l, b^i\}_{\{l \in L\}}$$

$\hat{y}^i$ — prediction of neural net for $x^i$



Need to compute: $\frac{\partial J}{\partial \mathbf{W}^{[l]}}, \frac{\partial J}{\partial \mathbf{b}^{[l]}}$

=> Gradient of loss with respect to weights and biases of each layer



Once gradients are computed,

update weights with:

- $\mathbf{W}_{t+1}^{[l]} := \mathbf{W}_t^{[l]} - \alpha_t \frac{\partial J}{\partial \mathbf{W}^{[l]}}(\mathbf{W}_t, \mathbf{b_t})$
- $\mathbf{b}_{t+1}^{[l]} := \mathbf{b}_t^{[l]} - \alpha_t \frac{\partial J}{\partial \mathbf{b}^{[l]}}(\mathbf{W}_t, \mathbf{b_t})$

where $\alpha_t$ is the learning rate ( step size )

# Neural networks
## Forward / Backward pass

- Forward pass: Compute the output of a neural network for a given input
  - Forward pass computes result of an operation and save any intermediates needed for gradient computation in memory

- Backward pass: Compute derivatives of the network parameters given the output. This is referred to as backpropogation (see Section 5.2 inML4Engineers book.)

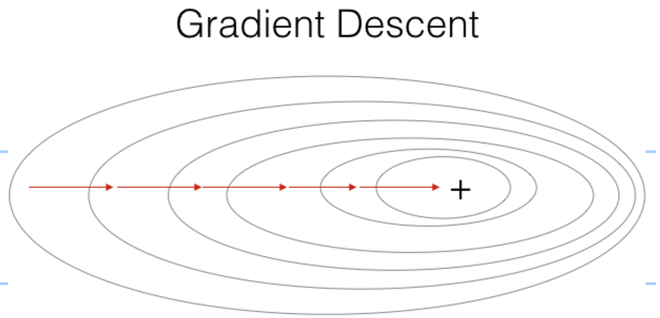
  - Backward pass applies the chain rule to compute the gradient of the loss function with respect to the inputs: we did an example in class last time
- During prediction (inference), you only need the forward pass.

- Prediction (inference): the process of using a trained machine learning model for prediction

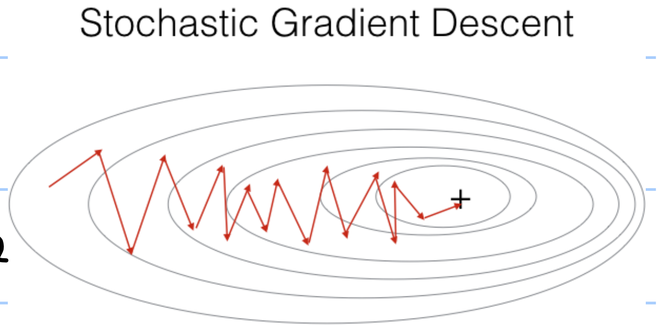
# Variants of gradient descent

Consider the loss: $J(\theta) = \frac{1}{N}\sum_{i=1}^{N} J^{(i)}(\theta)$ $= \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^i - y^i)^2$

Gradient descent $\theta_{t+1} = \theta_t - \alpha_t \frac{1}{N}\sum_{i=1}^{N} \frac{\partial J^i}{\partial \theta}(\theta_t)$



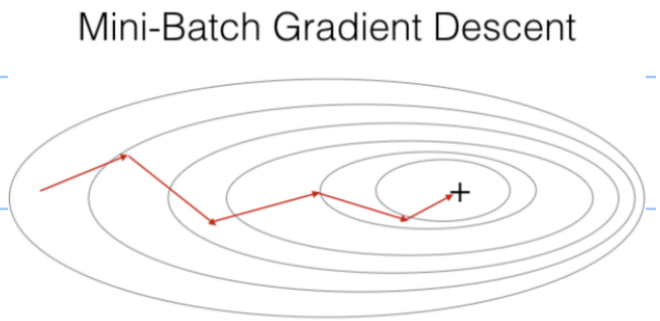

- High computation and memory due to using all data points




- Stochastic gradient descent: sample an index $i_s \in \{1,2,\ldots,N\}$

$\theta_{t+1} = \theta_t - \alpha_t \frac{\partial J^{i_s}}{\partial \theta}(\theta_t)$, $J^{i_s}(\theta) = (\hat{y}^{i_s} - y^{i_s})^2$

- Lower computational and memory burden, but high variance




- Mini-batch gradient descent: shuffle data set to $b$ batches of size $N_b$

example - $N = 100$, consider 5 batches of size 20,

$\theta_{t+1} = \theta_t - \frac{1}{N_b}\sum_{j=1}^{N_b} \frac{\partial J^{i_j}}{\partial \theta}(\theta_t)$, $i_j$ : index of data in the batch

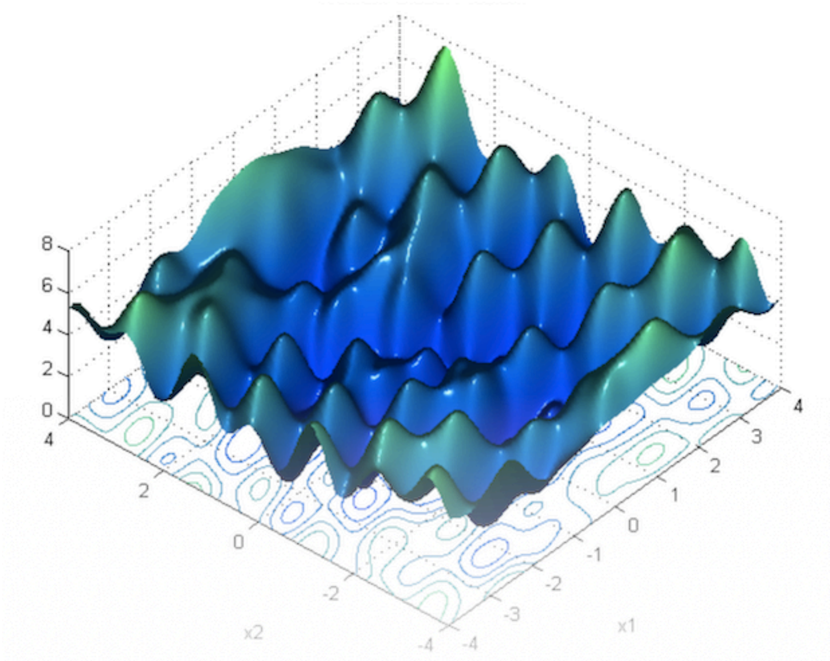
# Problems with training

**Loss function non-convex ->**

Choice of initialization

Choice of learning rate

affect the local minimum found

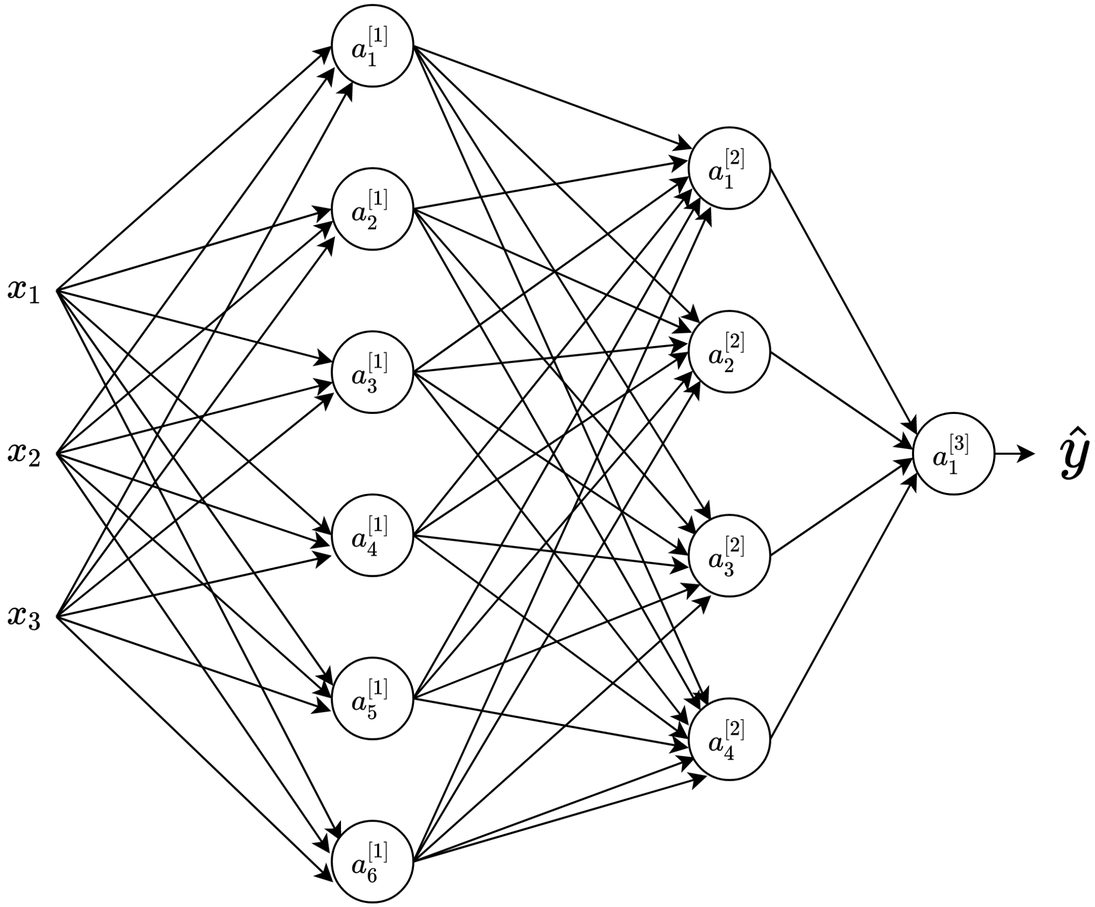



Variants of gradient descent are commonly used in practice to speed-up and improve convergence:

- Momentum update
- Nesterov Accelerated Gradient (NAG)
- Adam
- and more…

# Deep learning frameworks

## Implementing a simple neural network in PyTorch




```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3, 6)
        self.fc2 = nn.Linear(6, 4)
        self.fc3 = nn.Linear(4, 1)

    def forward(self, x):
        # First layer
        x = self.fc1(x)
        x = F.relu(x)
        # Second layer
        x = self.fc2(x)
        x = F.relu(x)
        # Output layer
        x = self.fc3(x)
        return x
```

# Summary - neural networks

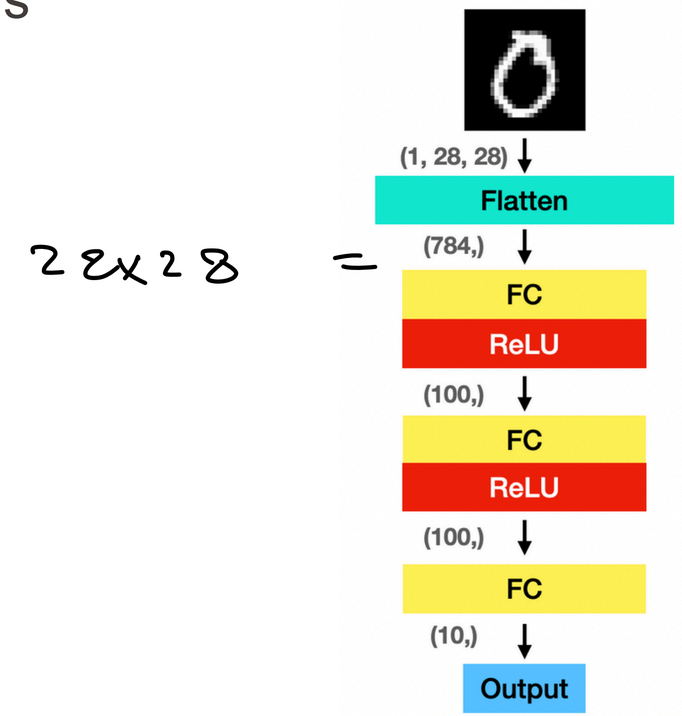
- Powerful predictors: universal function approximation means given an activation function, such as sign, with sufficient depth/nodes per layer, we can approximate any Lipschitz continue function
- Caveat: “sufficient depth/nodes per layer” -> requires
  - Large number of training data
  - Large computational resources
  - Might overfit to data (if many parameters), or underfit (training loss is non-convex)
  - The predictor might not be so easy to interpret

- Advice: first try to understand your problem and use simpler/interpretable methods. If these don’t work, then try neural networks

# Convolutional Neural Networks

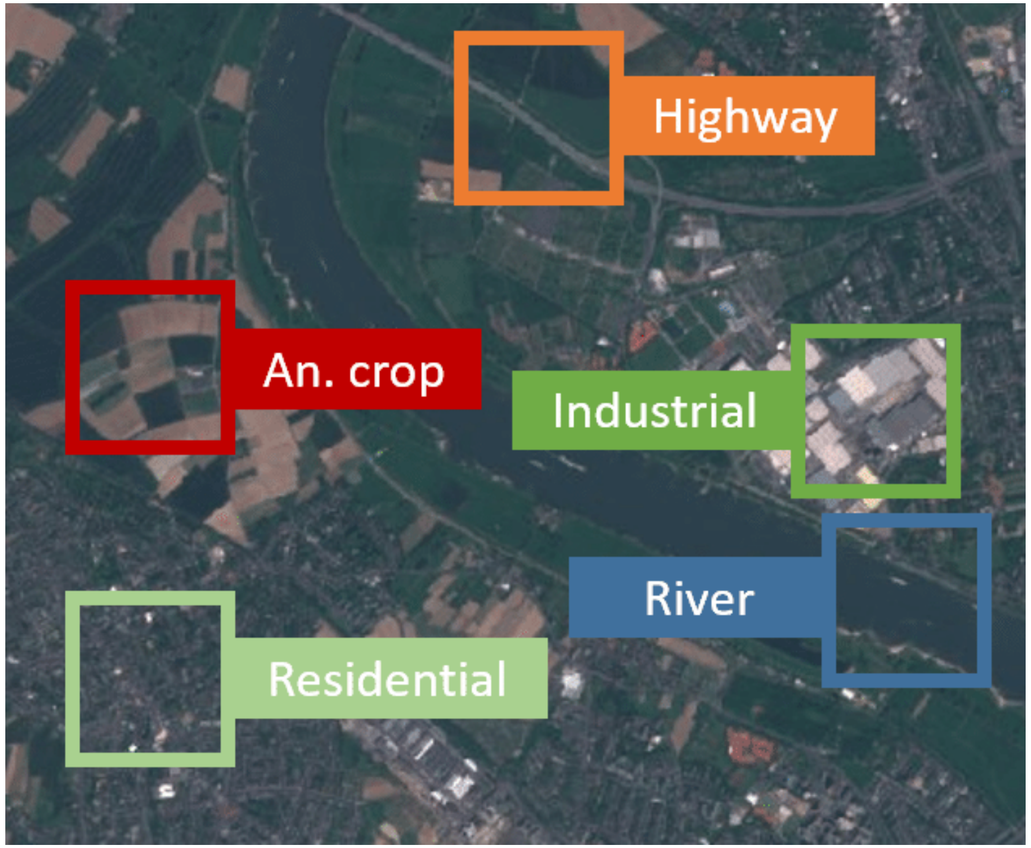
# Python exercise - digit classification

- A neural network for hand-written digit classification
- Training data is based on MNIST dataset: 70,000 images, hand-written digits
  - each image is $28 \times 28$ with grayscale values in $[0,255]$
- How does the classifier “see” an image? What are the features given to the classifier?
  - First, you will try logistic regression (a neural net with one-hidden layer) -> 7850 parameters
  - Next, you will try a neural net with 3 hidden layers -> 89610 parameters



# Motivations (you have a python exercise on this as well)

- **Datasets:** EuroSAT (images taken by Sentinel-2 satellite)
- **Goal:** Classify land cover of pictures taken by Sentinel-2

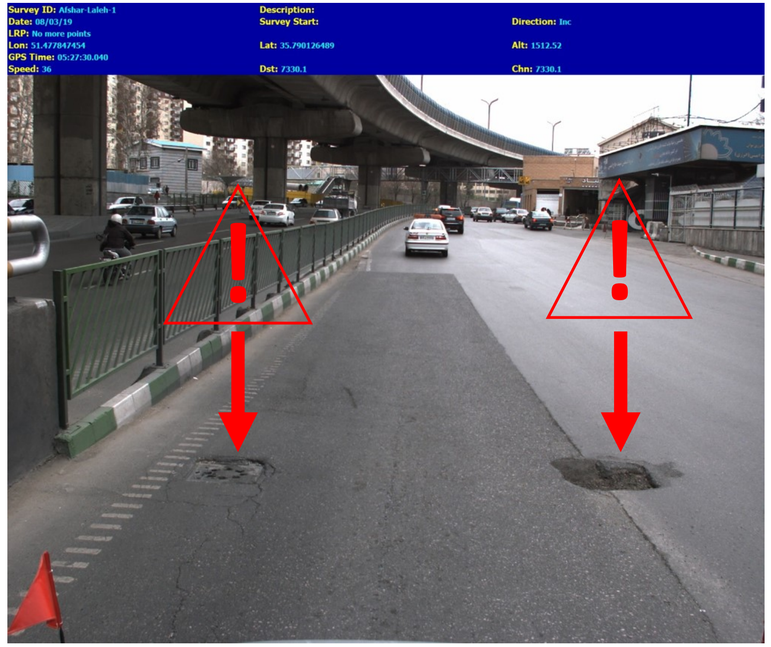



```python
class EuroSAT(ImageFolder):

    classes = [
        "0 - AnnualCrop",
        "1 - Forest",
        "2 - HerbaceousVegetation",
        "3 - Highway",
        "4 - Industrial",
        "5 - Pasture",
        "6 - PermanentCrop",
        "7 - Residential",
        "8 - River",
        "9 - SeaLake",
    ]
```

# Motivation: real-World Problem

## Detecting and Classifying Pavement Distress



On-time  preventive maintenance to reduce

- maintenance cost
- traffic delay
- fuel consumption
- accidents
- …

# Convolutional Neural Networks (CNN)

## Intro - Handling images with fully-connected NN

3x32x32 image

height (32)

width (32)

depth (3)

→ for the 3 color channels: R, G, B

Flatten

$3 \times 32 \times 32 =$ 3072x1 input

**By flattening, spatial structure gets lost!**

# Convolutional Neural Networks

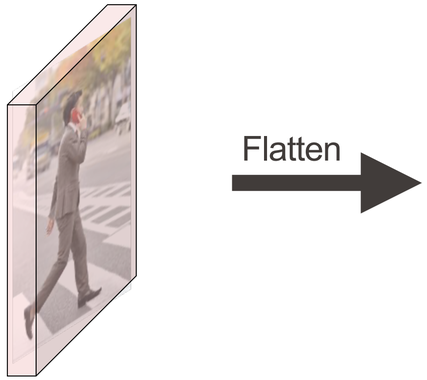
## Intro - Handling images with fully-connected NN

**A fully-connected neural net:**

- Requires flattening the image

  → spatial structure gets lost

- Doesn’t scale well to large images
  - e.g. 1024x1024x3 image results in 3’145’728 weights for each neuron of first hidden layer




How to efficiently model correlation between neighboring pixels?

=> **Convolutional Neural Networks**

# Motivation - convolutional neural networks

- Using neural networks for images has some caveats
  - Lose spatial information: image is flattened to a vector, so pixels' interrelations are ignored
  - Not robust to small shifts or translations: example: the same digit slightly moved looks like a new pattern
  - Requires a huge number of parameters -> fully connected layers on large images make training expensive and inefficient
- Convolutional address these issues while improving interpretability and prediction error
  - Convolution layers → linear operations that extract local features
  - Activation functions → introduce nonlinearity
  - Pooling layers → summarize nearby activations for robustness

# Convolution definition

- Signal $x \in \mathbb{R}^d$
- Filter (also referred to as kernel) $w \in \mathbb{R}^{2m+1}$, $(w_{-m}, w_{-m+1}, \dots, w_0, \dots, w_m)$
- Convolution outcome $z = w * x$ $\quad z_i = \sum_{k=-m}^{m} w_k x_{i+k}$

for $i = m+1, \dots, d-m$, $z \in \mathbb{R}^{d-2m}$

- Example $x = (4,1,1,2,3,5,8,13) \in \mathbb{R}^8$, $w = (1,-2,1) \in \mathbb{R}^3$

$w_{-1} = 1$, $w_0 = -2$, $w_1 = 1$

$(1,-2,1)$

$z_2 = 1 \times 4 + (-2) \times 1 + 1 \times 1 = 3$, $z_3 = 1 \times 1 + (-2) \times 1 + 1 \times 2 = 1$, $\dots, z_7$

# Convolution - handling boundaries

- Zero padding to get an output of the same dimension as the input $z \in \mathbb{R}^d$

$$x = (4,1,1,2,3,5,8,13) \in \mathbb{R}^8,\ w = (1,-2,1) \in \mathbb{R}^3$$

$\tilde{x} = (0,4,1,1,2,3,5,8,13,0) \in \mathbb{R}^{10}$

$z = w * \tilde{x} \in \mathbb{R}^8$

$z_1 = 1 \times 0 + (-2) \times 4 + 1 \times 1 = -7$

$\vdots$

$z_8 = 1 \times 8 + (-2) \times 13 + 1 \times 0 = -18$

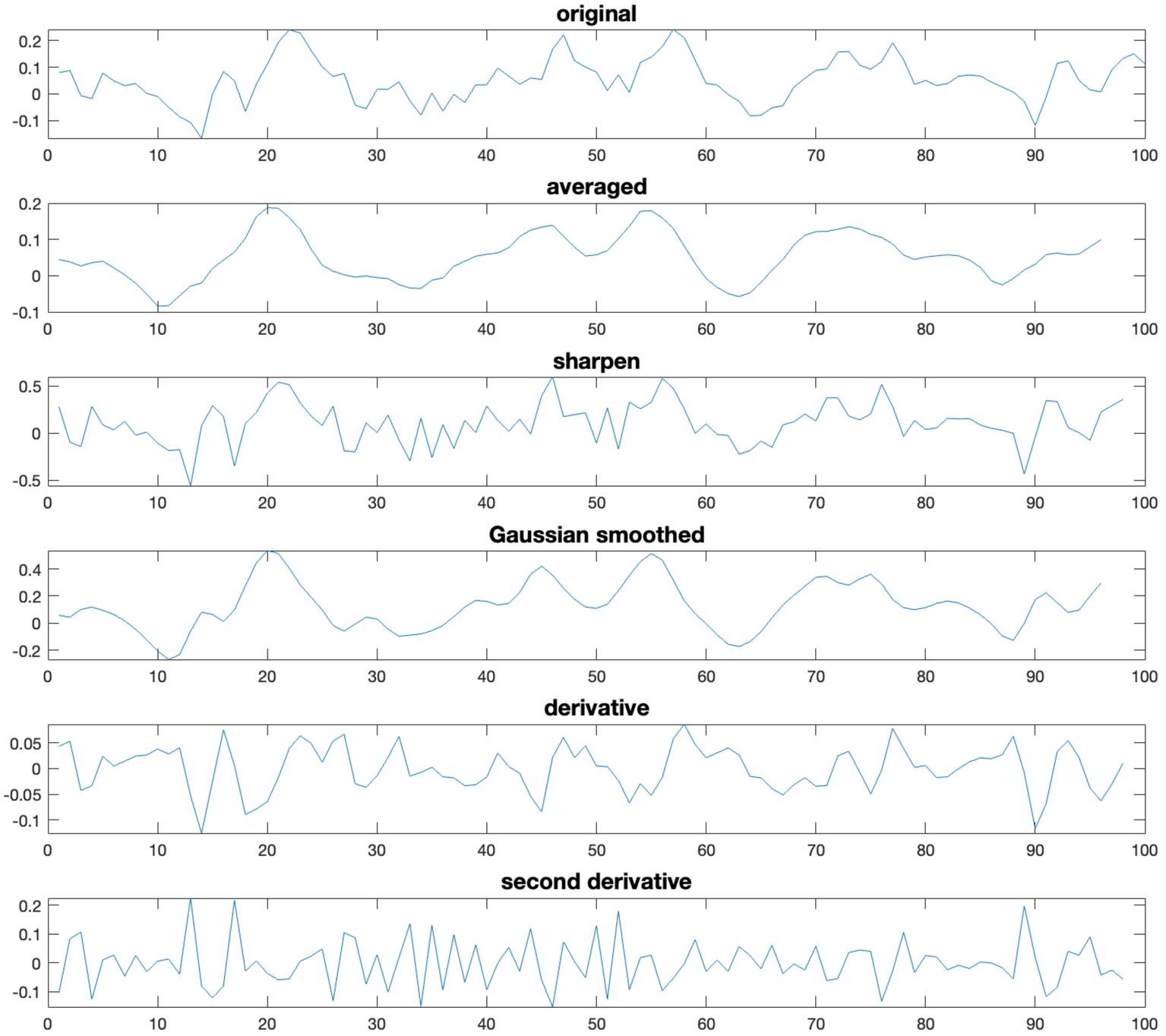
(Note : there can be other ways to resolve boundaries )

# Convolution to get features

- Convolution with suitable filters help understand what happens in a neighborhood
- Examples:
  - Give average value $w = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ $z_i = \frac{x_{-i} + x_i + x_{i+1}}{3}$
  - Sharpen the signal $w = (-1, 4, -1)$ $z_i = -x_{-i} + 4x_i - x_{i+1}$
  - Blur the signal $w = \left(e^{-\frac{1}{2\sigma^2}}, e^{0}, e^{-\frac{1}{2\sigma^2}}\right)$ (weighted average)
  - Approximate derivative $w = \frac{(-1, 0, 1)}{2}$ $z_i = \frac{x_{i+1} - x_{i-1}}{2}$
  - Approximate second derivative $w = (1, -2, 1)$ $z_i = x_{i-1} - 2x_i + x_{i+1}$

EPFL

$x \in \mathbb{R}^{100}$



# Convolutional in 2 dimension

To show how the convolution operation is computed in 2D, let's use a simpler example:

**5x5** input, **3x3** filter

signal

Input (5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

(kernel)

Filter (3x3)

| | | |
|---|---|---|
| 1 | -1 | 0 |
| 0 | 2 | -3 |
| -1 | 0 | 2 |

Bias: $b = 0$

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | | |
| | | |
| | | |

Bias: $b = 0$

1x1 + 0x(-1) + 3x0 + 0x0 + 3x2 +
4x(-2) + 1x(-1) + 0x0 + 2x2 + 0
= 2

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | 5 | |
| | | |
| | | |

Bias: $b = 0$

0x1 + 3x(-1) + 0x0 + 3x0 + 4x2 +
0x(-2) + 0x(-1) + 2x0 + 0x2 + 0
= 5

# Convolutional
## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| | | |
| | | |

Bias: $b = 0$

3x1 + 0x(-1) + 2x0 + 4x0 + 0x2 +
2x(-2) + 2x(-1) + 0x0 + 1x2 + 0
= -1

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| -15 | | |
| | | |

Bias: $b = 0$

0x1 + 3x(-1) + 4x0 + 1x0 + 0x2 +
2x(-2) + 8x(-1) + 12x0 + 0x2 + 0
= -15

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

Bias: $b = 0$

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| -15 | -7 | |
| | | |

3x1 + 4x(-1) + 0x0 + 0x0 + 2x2 +
0x(-2) + 12x(-1) + 0x0 + 1x2 + 0
= -7

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| -15 | -7 | 2 |
| | | |

Bias: $b = 0$

4x1 + 0x(-1) + 2x0 + 2x0 + 0x2 +
1x(-2) + 0x(-1) + 1x0 + 0x2 + 0
= 2

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| -15 | -7 | 2 |
| 31 | | |

Bias: $b = 0$

1x1 + 0x(-1) + 2x0 + 8x0 + 12x2
+ 0x(-2) + 0x(-1) + 6x0 + 3x2 + 0
= 31

# Convolutional

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

Bias: $b = 0$

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| -15 | -7 | 2 |
| 31 | -8 | |

0x1 + 2x(-1) + 0x0 + 12x0 + 0x2 +
1x(-2) + 6x(-1) + 3x0 + 2x2 + 0
= -8

# Convolutional Neural Networks

## Convolution computation example

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

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

=

| | | |
|---|---|---|
| 2 | 5 | -1 |
| -15 | -7 | 2 |
| 31 | -8 | 1 |

Bias: $b = 0$

2x1 + 0x(-1) + 1x0 + 0x0 + 1x2 +
0x(-2) + 3x(-1) + 2x0 + 0x2 + 0
= 1

# Convolution example

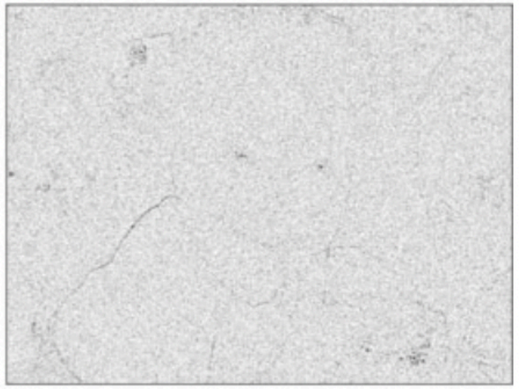
Example from ML4Engineers book.

Filter based on finite approximation of Laplacian operator:

(a)

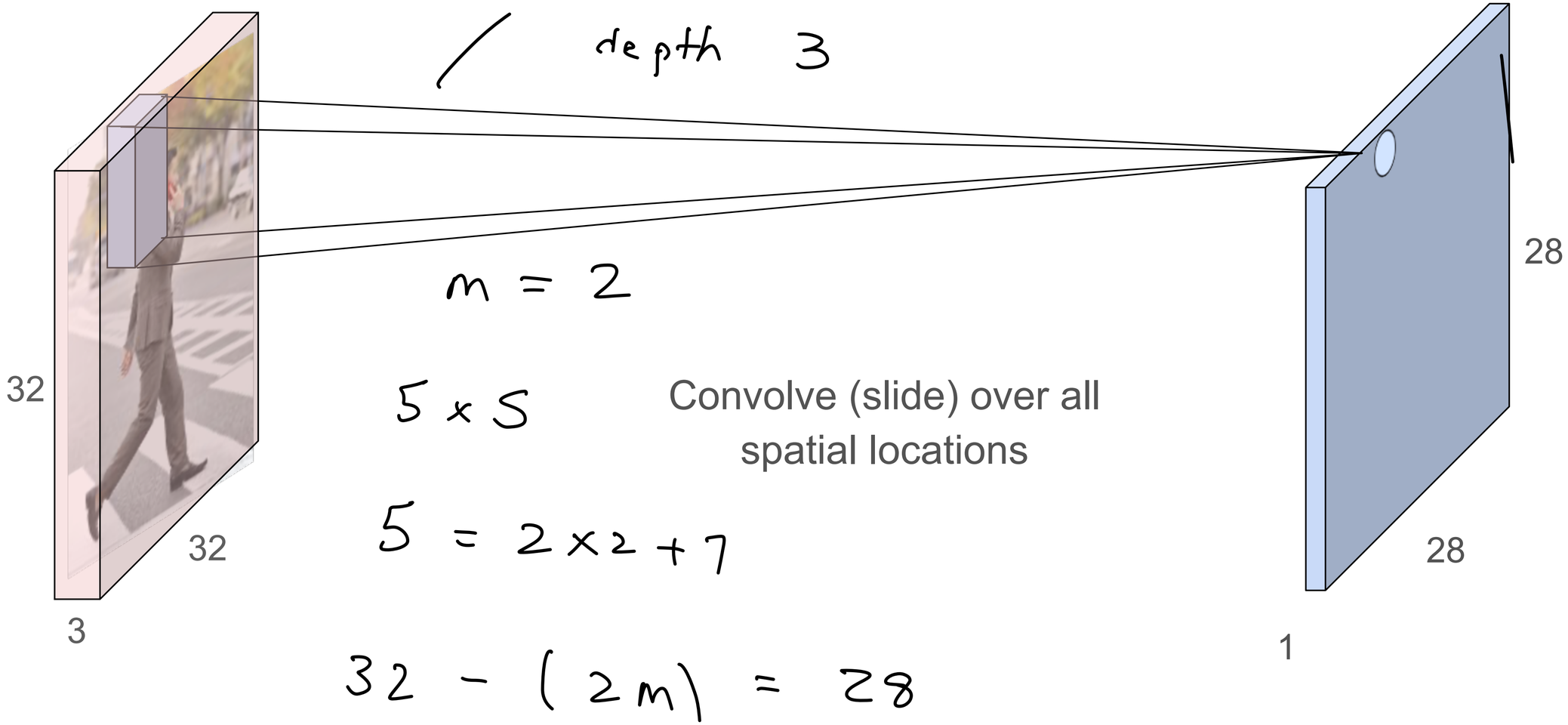
(b)

(c)



(d)

# Convolution in 3 dimension, example



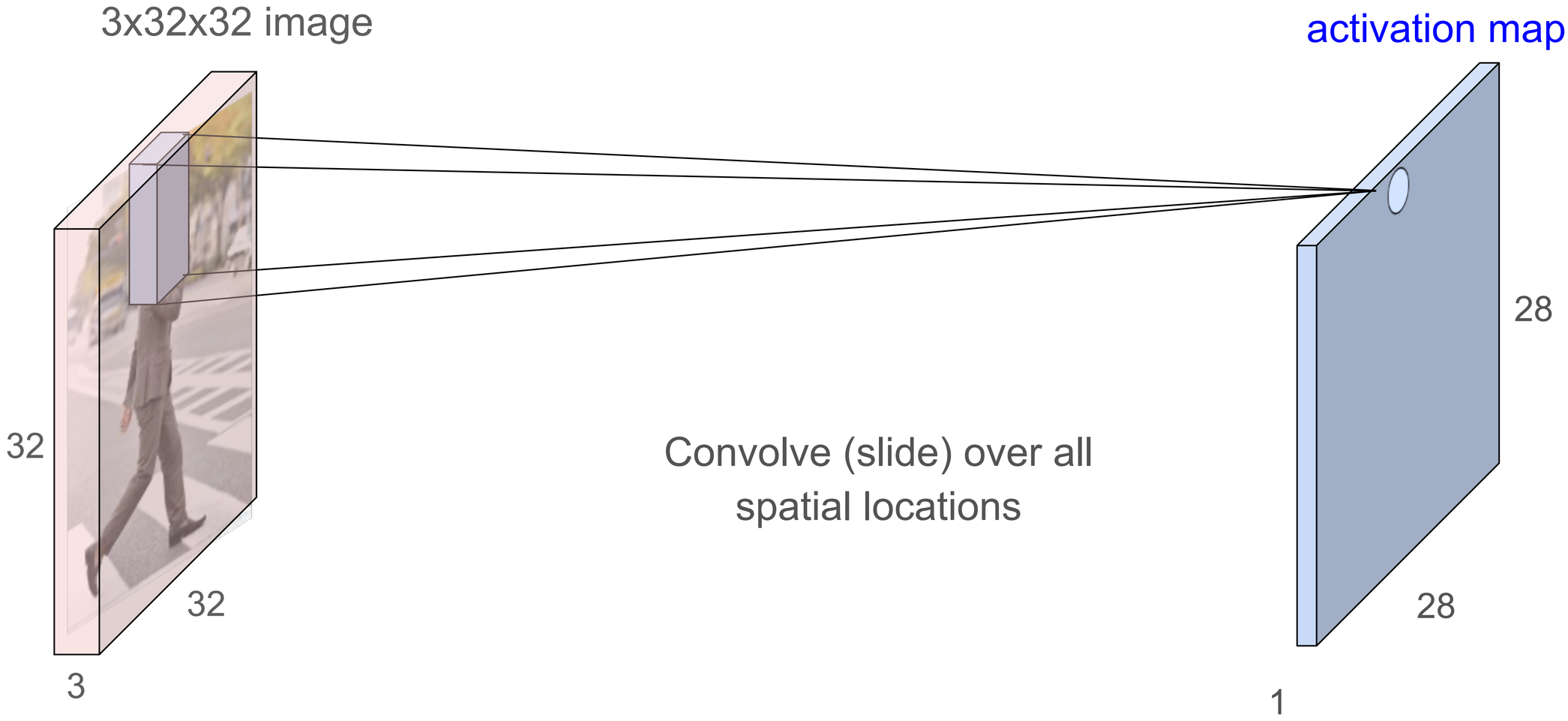

$m = 2$

$5 \times 5$

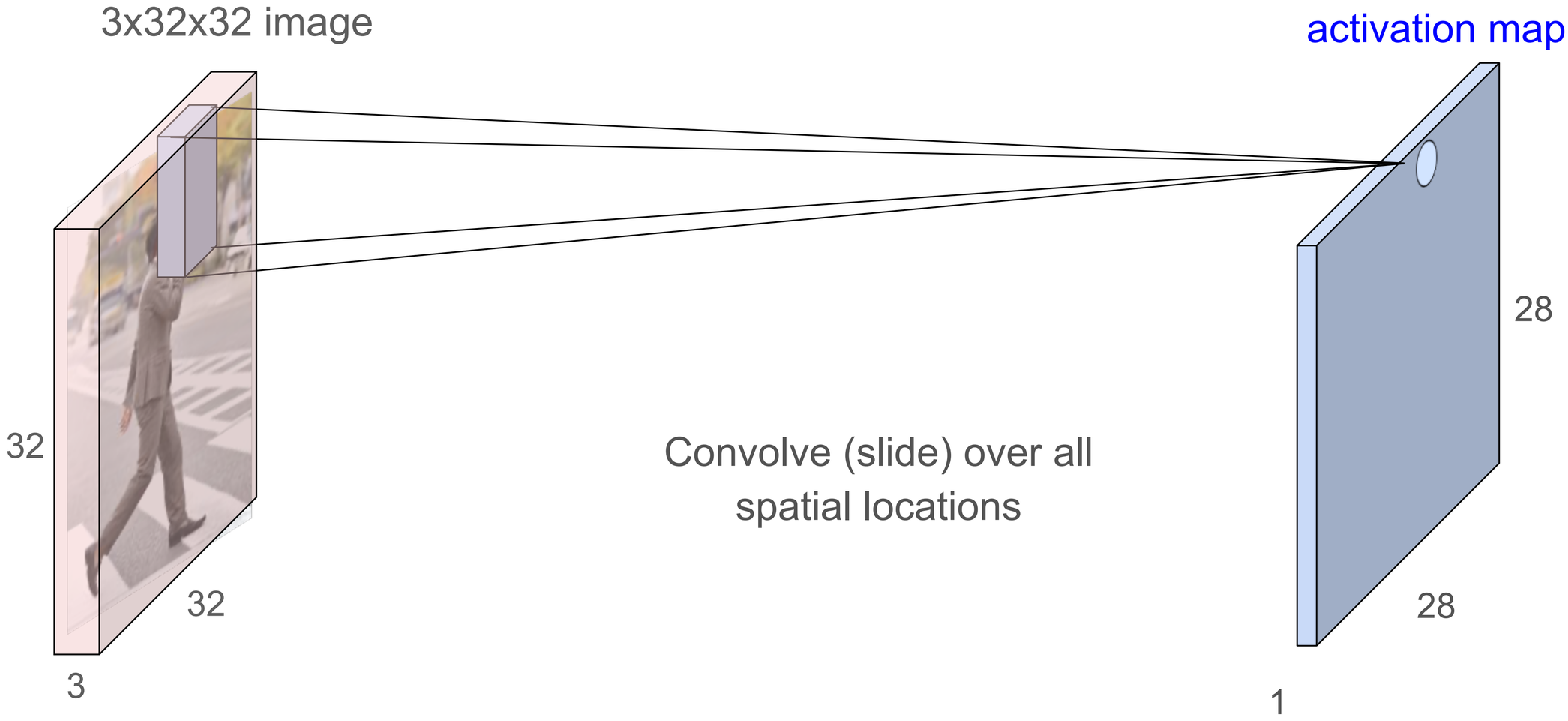
$5 = 2 \times 2 + 1$

$32 - (2m) = 28$

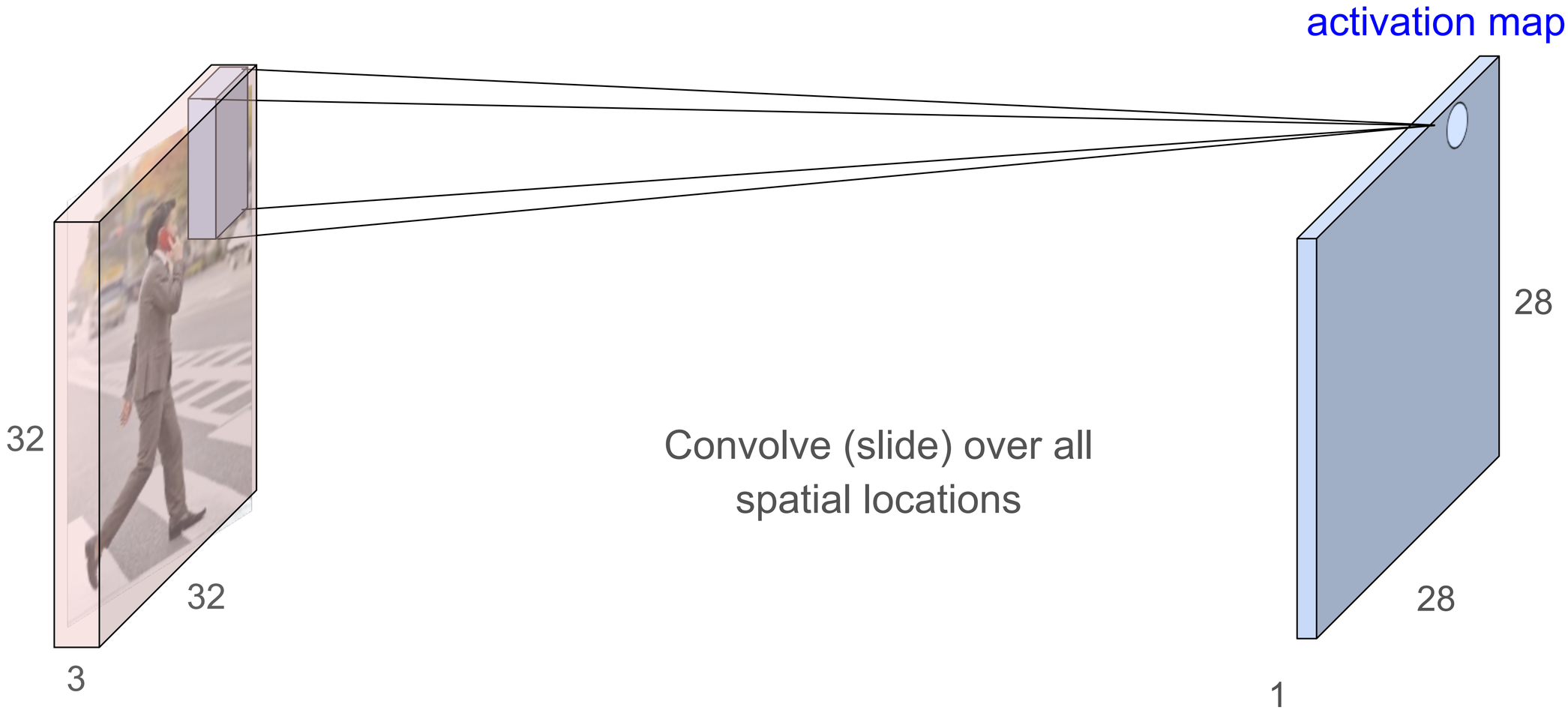
# Convolution in 3 dimension, example



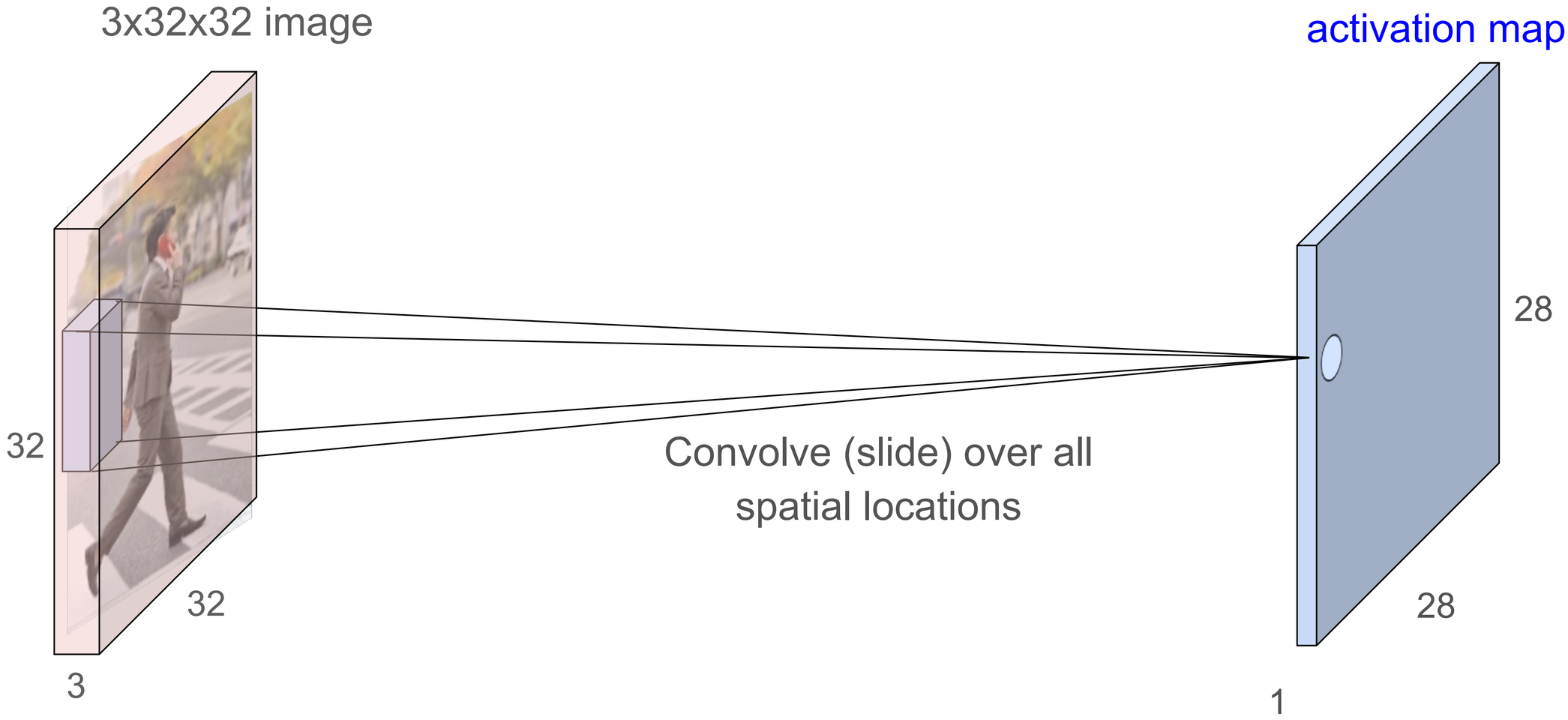
# Convolution in 3 dimension, example



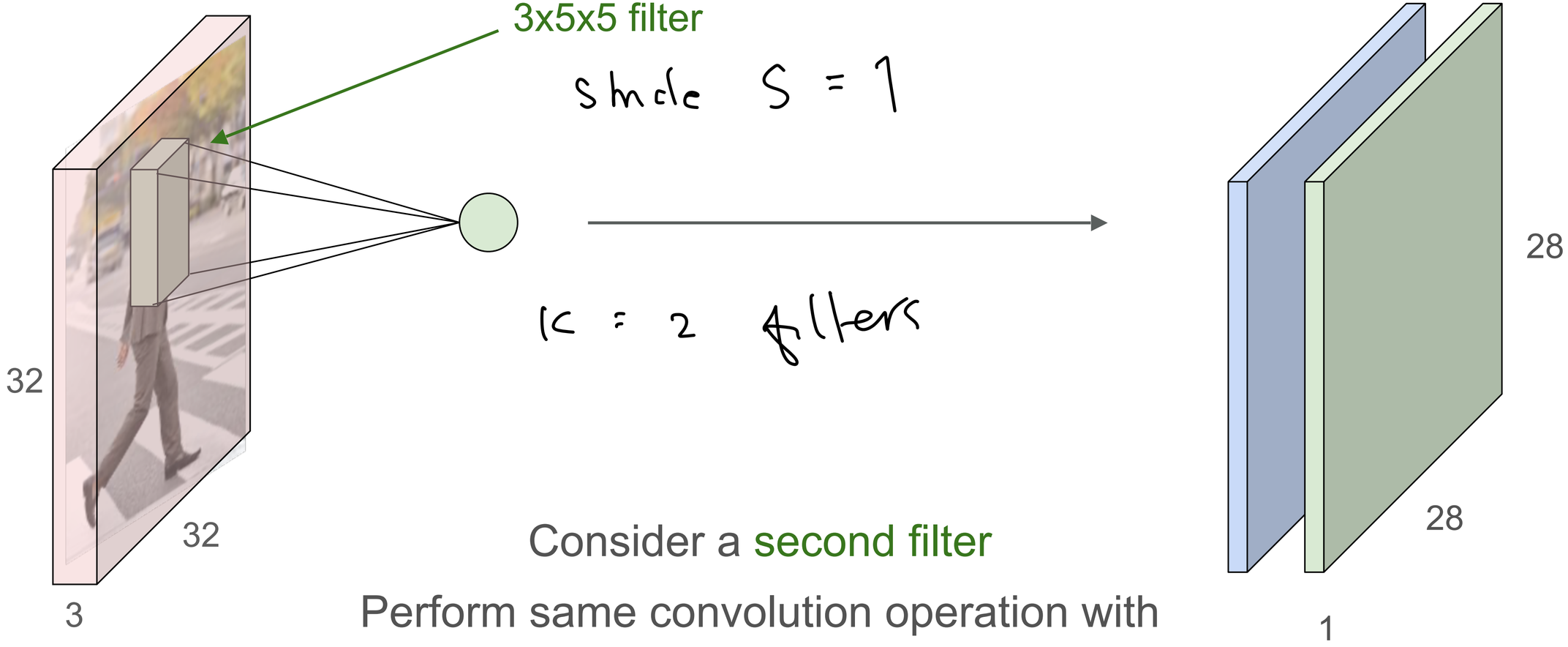
# Convolution in 3 dimension, example



# Convolution in 3 dimension, example



# Convolution in 3 dimension, example



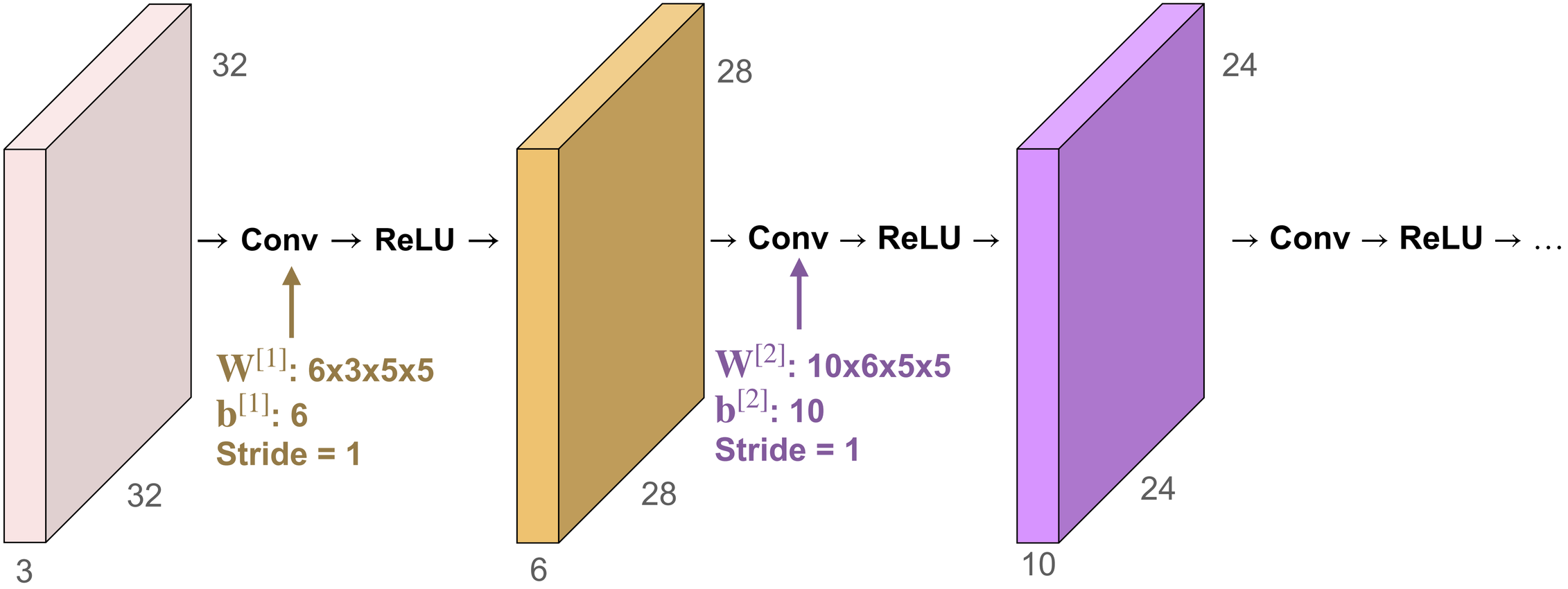

Consider a second filter

Perform same convolution operation with this filter to get a second activation map

Alexandre Alahi

# Convolutional Neural Networks
## Overview

**CNN** is a sequence of **convolutional layers**, interspersed with **activation functions**




**Input:**

3 x 32 x 32

**First hidden layer:**

6 x 28 x 28

**Second hidden layer:**

10 x 24 x 24

# Convolutional neural networks

There are a few other engineering tricks to make CNN work

- Sometimes convolution is applied with a so-called **stride**
  - Stride 1: move one pixel at a time → dense coverage, large output
  - Stride 2: skip every other pixel → smaller output (downsampling)
  - Can have stride S > 1 to reduce size and thus, memory and computation

- Sometimes after a convolution a so-called **pooling** layer is used
  - Reduces spatial size → fewer parameters, faster computation
  - Provides translation invariance → small shifts in the input don’t change the pooled value much
  - highlights the presence of a feature, not its exact location

# Convolutional applied with a stride

We can also apply convolution with a **stride of S**, that is, the filter moves S pixels at a time, skipping every other position and producing a smaller output.

Input (1x5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

Filter (1x3x3)

| | | |
|---|---|---|
| 1 | -1 | 0 |
| 0 | 2 | -3 |
| -1 | 0 | 2 |

Bias: $b$ = 0

# Convolutional Neural Networks

## Changing the stride

Back to our simple example, but change to **stride of 2**

Input (1x5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

Filter (1x3x3)

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

Bias: $b = 0$

=

Output (1x2x2)

| | |
|---|---|
| 2 | |
| | |

1x1 + 0x(-1) + 3x0 + 0x0 + 3x2 +
4x(-2) + 1x(-1) + 0x0 + 2x2 + 0
= 2

# Convolutional Neural Networks

## Changing the stride

Back to our simple example, but change to **stride of 2**

Input (1x5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

Filter (1x3x3)

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

Bias: $b = 0$

=

Output (1x2x2)

| | |
|---|---|
| 2 | -1 |
| | |

3x1 + 0x(-1) + 2x0 + 4x0 + 0x2 +
2x(-2) + 2x(-1) + 0x0 + 1x2 + 0
= -1

# Convolutional Neural Networks

## Changing the stride

Back to our simple example, but change to **stride of 2**

Input (1x5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

Filter (1x3x3)

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

Bias: $b = 0$

=

Output (1x2x2)

| | |
|---|---|
| 2 | -1 |
| 31 | |

1x1 + 0x(-1) + 2x0 + 8x0 + 12x2
+ 0x(-2) + 0x(-1) + 6x0 + 3x2 + 0
= 31

# Convolutional Neural Networks

## Changing the stride

Back to our simple example, but change to **stride of 2**

Input (1x5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

•

Filter (1x3x3)

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

Bias: $b = 0$

=

Output (1x2x2)

| | |
|---|---|
| 2 | -1 |
| 31 | 1 |

2x1 + 0x(-1) + 1x0 + 0x0 + 1x2 +
0x(-2) + 3x(-1) + 2x0 + 0x2 + 0
= 1

# Convolutional Neural Networks
## Zero-padding

Height and width shrink quite quickly due to the repeated convolutions

To avoid this, we can add zero-padding:

Input (1x5x5)

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 2 |
| 0 | 3 | 4 | 0 | 2 |
| 1 | 0 | 2 | 0 | 1 |
| 8 | 12 | 0 | 1 | 0 |
| 0 | 6 | 3 | 2 | 0 |

Zero-padding = 1

Zero-padded input (1x7x7)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 3 | 0 | 2 | 0 |
| 0 | 0 | 3 | 4 | 0 | 2 | 0 |
| 0 | 1 | 0 | 2 | 0 | 1 | 0 |
| 0 | 8 | 12 | 0 | 1 | 0 | 0 |
| 0 | 0 | 6 | 3 | 2 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

If we use a 3x3 filter with a stride of 1 on the padded input, we get a 5x5 output

→ same size as input

# Pooling layer in CNN

- After detecting features with convolution, we want to make the representation smaller and more robust to small shifts
- Pooling: summarizing small regions, by taking the maximum or average value locally.

Input (1x6x6)

| | | | | | |
|---|---|---|---|---|---|
| 3 | 0 | 1 | 0 | 2 | 4 |
| 0 | 1 | 8 | 12 | 0 | 0 |
| 4 | 0 | 0 | 3 | 2 | 2 |
| 2 | 0 | 1 | 0 | 1 | 1 |
| 3 | 2 | 0 | 6 | 0 | 5 |
| 1 | 0 | 6 | 0 | 0 | 9 |

max pools with a filter of 2x2 & stride 2

Output (1x3x3)

| | | |
|---|---|---|
| 3 | 12 | 4 |
| 4 | 3 | 2 |
| 3 | 6 | 9 |

# Pooling example

Example from ML4Engineers book.

Max-pooling of stride 9 applied to (a)

(a)

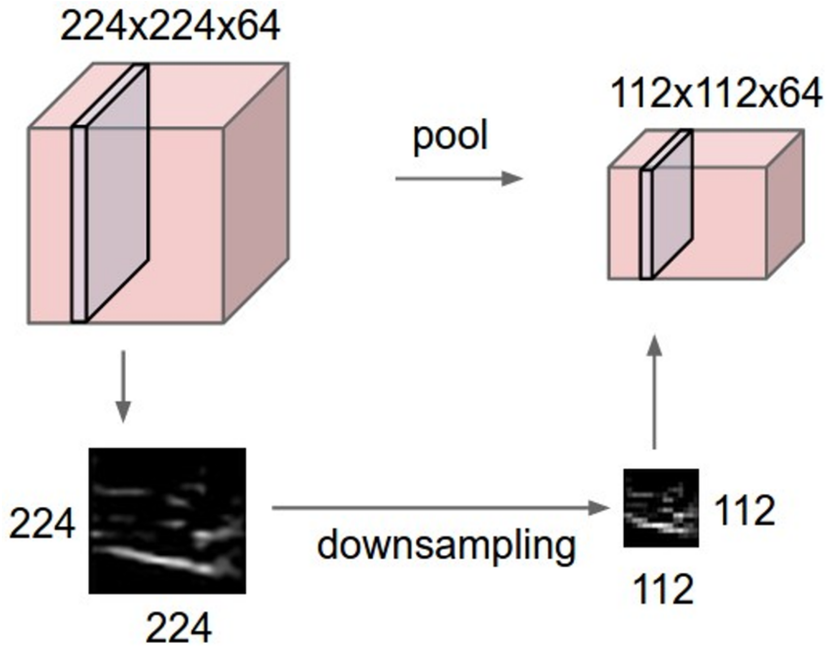
(b)

# Convolutional Neural Networks (CNN)
## Pooling layer

CNNs may include pooling layers **to reduce the spatial size of the representation**

Pooling layers require two hyper-parameters: their spatial extent $F$ and their stride $S$

- Most common layer uses 2x2 filters of stride 2 ($F = 2,\ S = 2$)



# Convolutional Neural Networks

## Convolution layer summary

**The convolution layer:**

- Accepts a volume of size $C_{in} \times H_1 \times W_1$

- Requires four hyper-parameters:
  - Number of filters $K$
  - Spatial extent of filters $F$
  - Stride $S$
  - Amount of zero (repetition) padding $P$

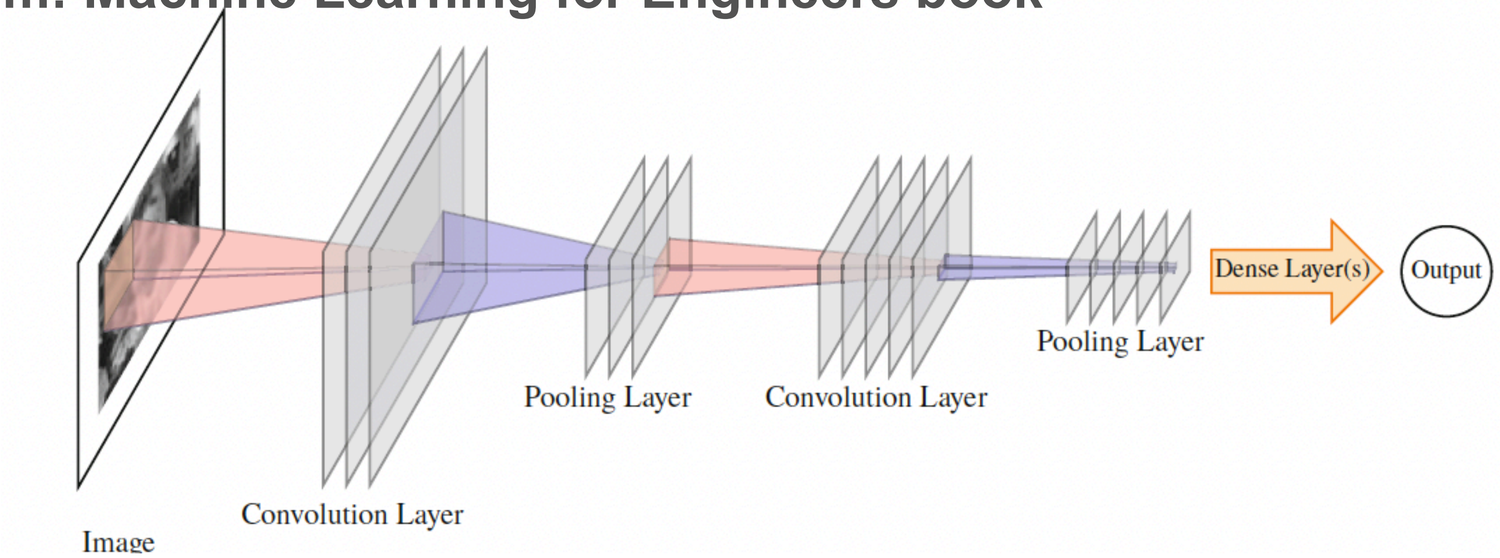
- Produces of a volume of size $C_{out} \times H_2 \times W_2$ where:
  - $C_{out} = K$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $W_2 = (W_1 - F + 2P)/S + 1$

**Note:**

There are $C_{in} \cdot F \cdot F$ weights per filter, for a total of $(C_{in} \cdot F \cdot F) \cdot K$ weights and $K$ biases per layer

# Convolutional neural net

**From: Machine Learning for Engineers book**




Usually the architecture is fixed for a given problem (object classification) based on trial and error

examples: LeNet-5, LeCun et al. ,1998, AlexNet Krizhevsky et al., 2012, GoogLeNet (Inception v1), etc.

It can be applied to transfer learning to a new problem

# Summary and your tasks for this week

**Neural networks**

- Powerful in expressibility but hard to train and limited in interpretability
- Training through (stochastic/mini-batch) gradient descent

**Convolutional neural network**

- A type of neural networks used for computer vision
- Build on the notion of convolution (linear operation) followed by nonlinear operation (pooling, ReLU, etc)

**Your tasks**

- Optional: quiz 1 during exercise hour
- Python exercises on neural networks and convolutional neural networks