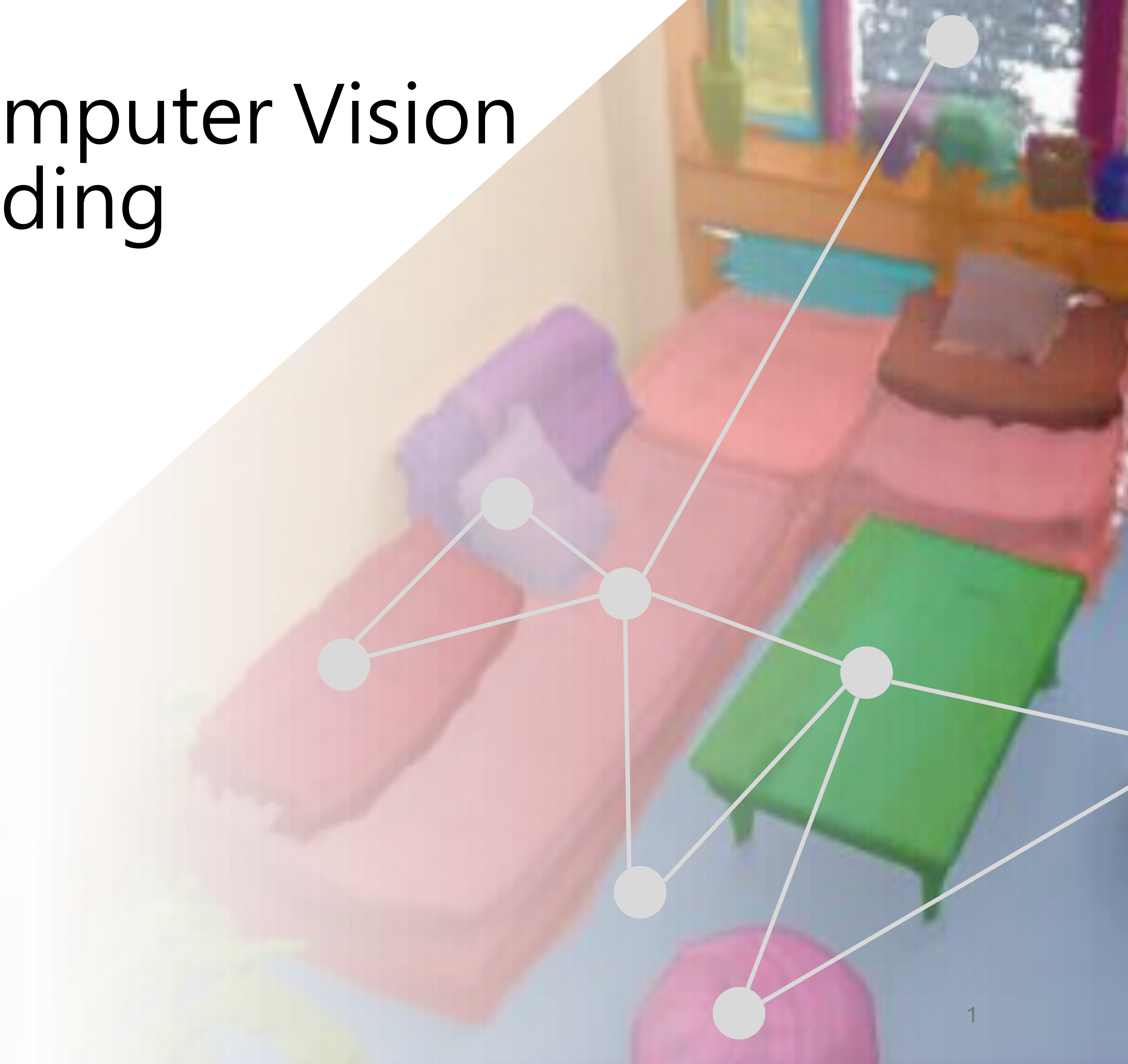
Deep Learning for Computer Vision and Scene Understanding



Dr. Helisa Dhamo
Research Scientist in 3D Vision
Huawei Noah’s Ark Lab London

About me

Bachelor's
Computer Engineering
Polytechnic University of Tirana
Albania

Master's
Computational Science and Engineering
Technical University of Munich
Germany

Master's thesis

PhD at CAMP Lab
Technical University of Munich
Germany

Research Intern
Computer Vision
Meta Reality Labs, USA (remote)

Research Scientist
3D Computer Vision
Huawei Noah's Ark Lab, London
United Kingdom

Course outline

Monday

Motivation

Fundamentals of Deep Learning and Computer Vision

Tuesday

Convolutional Neural Nets

Deep Learning for 2D Vision

Wednesday

3D Understanding from Images

Deep Learning for Video and 3D Vision

Thursday

3D Vision

Extended Scene Understanding

Friday

Exam

Recent trends

Ask me anything

Grading

Theoretical Exam on Friday 2 August 2024

- Mostly multichoice questions
- 1-2 practical exercises

Practical coding exercises for **bonus** grade

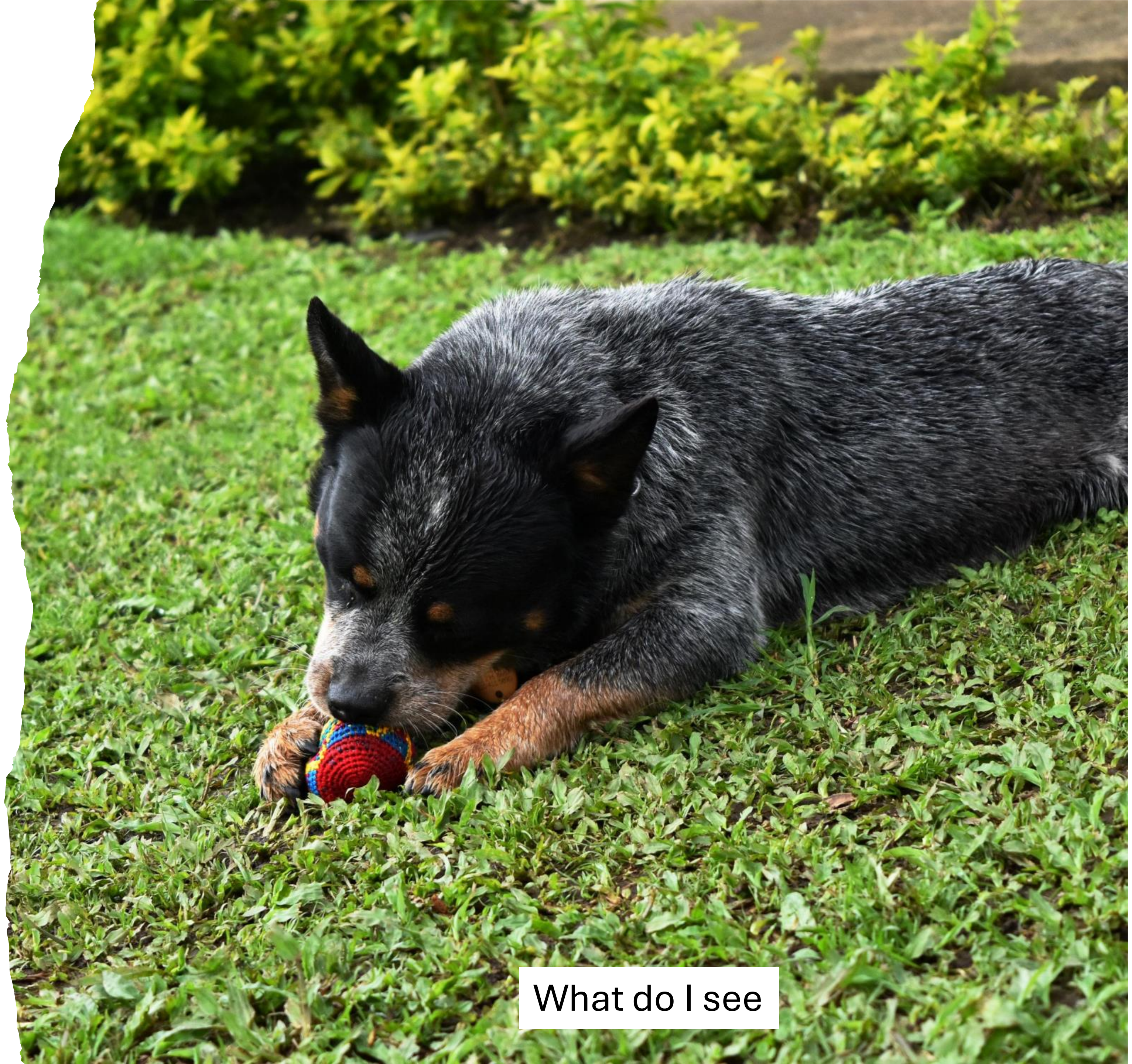
- Send back via email before 11 August 2024

Motivation and Fundamentals

Lecture 1

?

What does the computer see?



24	53	126	249	20	9	9	0	5	44	22	8
10	42	4	255	250	48	43	22	27	64	86	33
3	88	77	32	123	43	62	84	89	89	0	0
2	2	5	3	60	2	70	123	43	120	115	
45	52	0	126	249	20	9	9	0	123	43	
3	9	0	5	44	22	8	97	123	43	120	
4	43	62	84	89	89	0	40	66	68	80	
88	77	32	123	43	120	115	40	50	88	77	
0	9	0	5	44	22	8	97	88	77	32	
88	77	32	123	43	62	84	89	0	76	4	
3	9	0	5	44	22	8	97	156	149	120	
12	43	120	115	40	50	88	255	250	48	43	
6	9	0	5	44	22	8	97	88	77	32	
43	62	84	89	89	0	123	43	120	115	40	

What does the computer see?

What do I see

Computer Vision

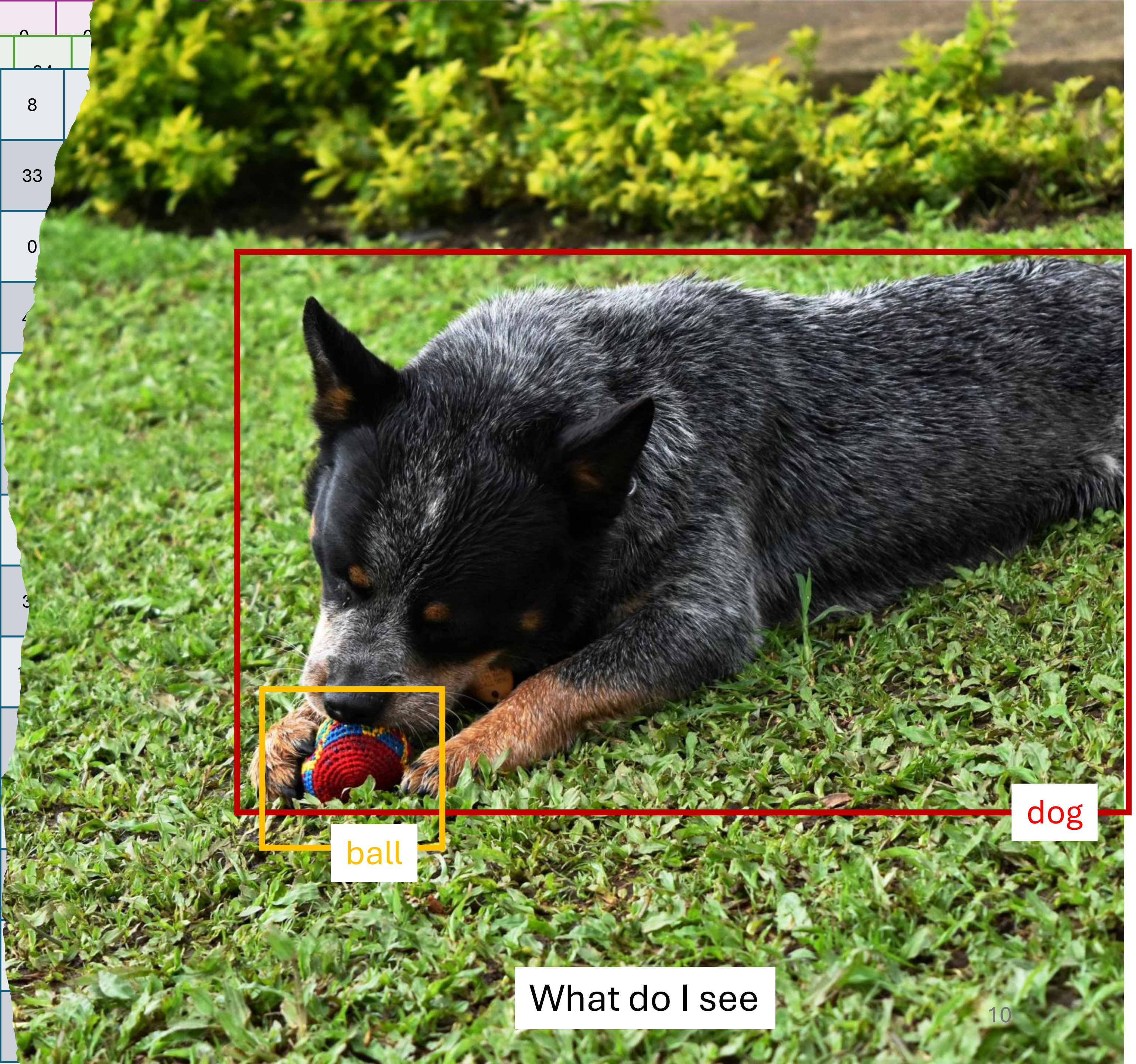
Field of Computer Science that aims to make sense at image/video inputs, i.e. **identify**, **understand** and **extract relevant information**

Scene Understanding

Aspect of Computer Vision that aims to identify and analyse objects and their context (surrounding scene, relations to other objects)

24	53	126	249	20	9	9	0	5	44	22	8
10	42	4	255	250	48	43	22	27	64	86	33
3	88	77	32	123	43	62	84	89	89	0	0
2	2	5	3	60	2	70	123	43	120	115	
45	52	0	126	249	20	9	9	0	123	43	
3	9	0	5	44	22	8	97	123	43	120	
4	43	62	84	89	89	0	40	66	68	80	
88	77	32	123	43	120	115	40	50	88	77	
0	9	0	5	44	22	8	97	88	77	32	
88	77	32	123	43	62	84	89	0	76	4	
3	9	0	5	44	22	8	97	156	149	120	
12	43	120	115	40	50	88	255	250	48	43	
6	9	0	5	44	22	8	97	88	77	32	
43	62	84	89	89	0	123	43	120	115	40	

What does the computer see?



What do I see

Computer Vision Applications

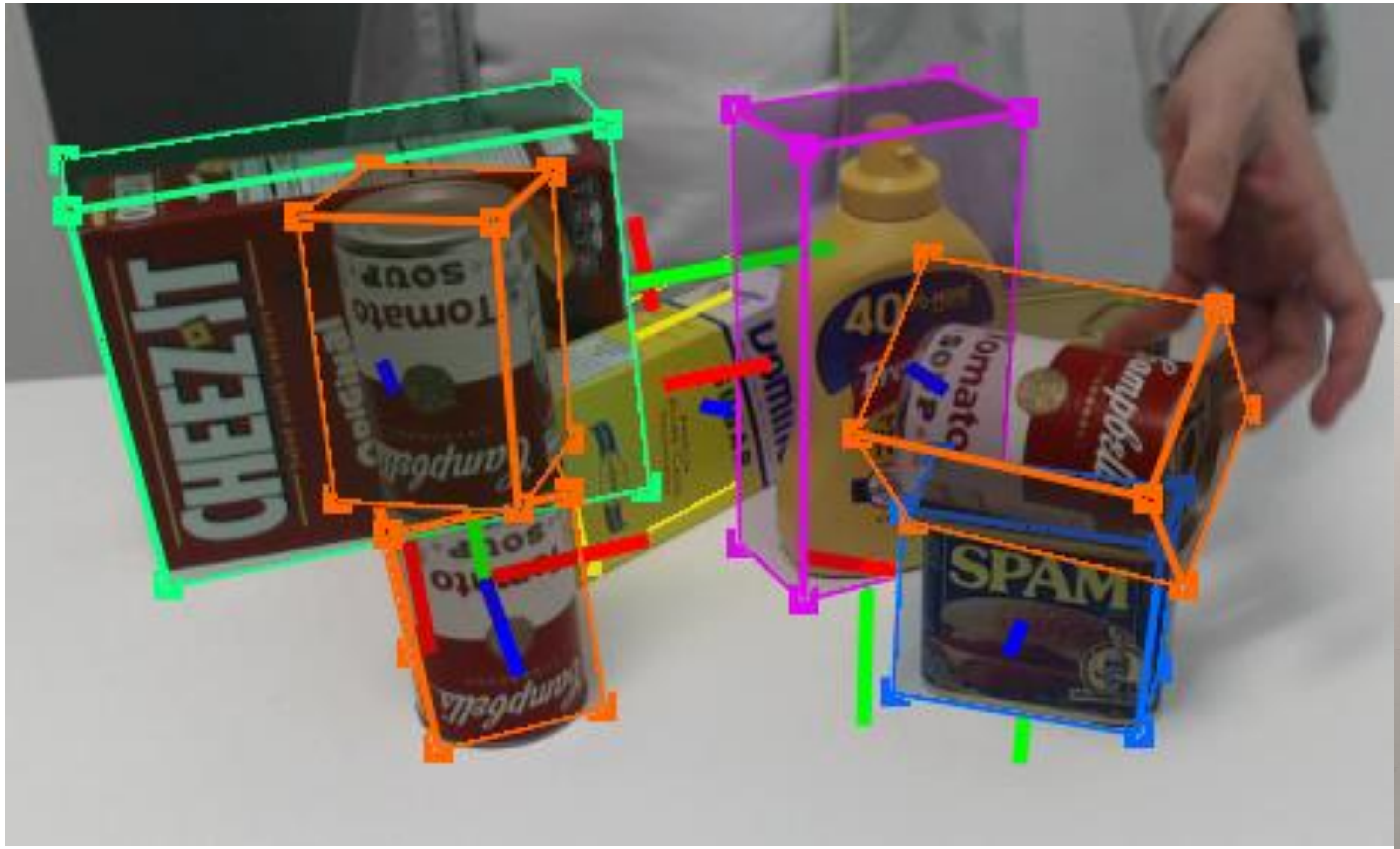
Image segmentation

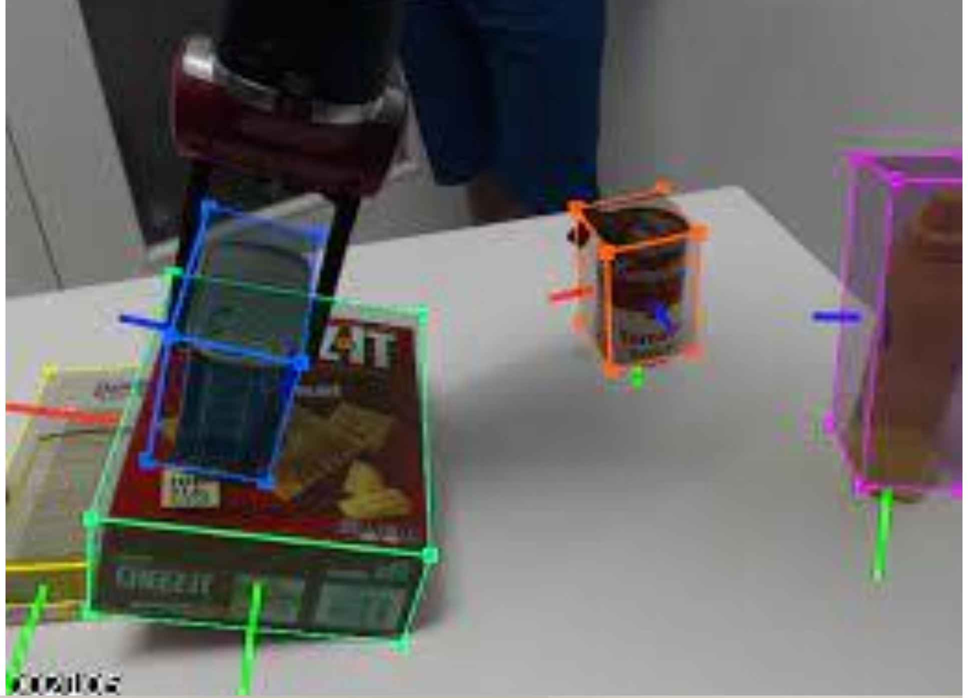
https://segment-anything.com/

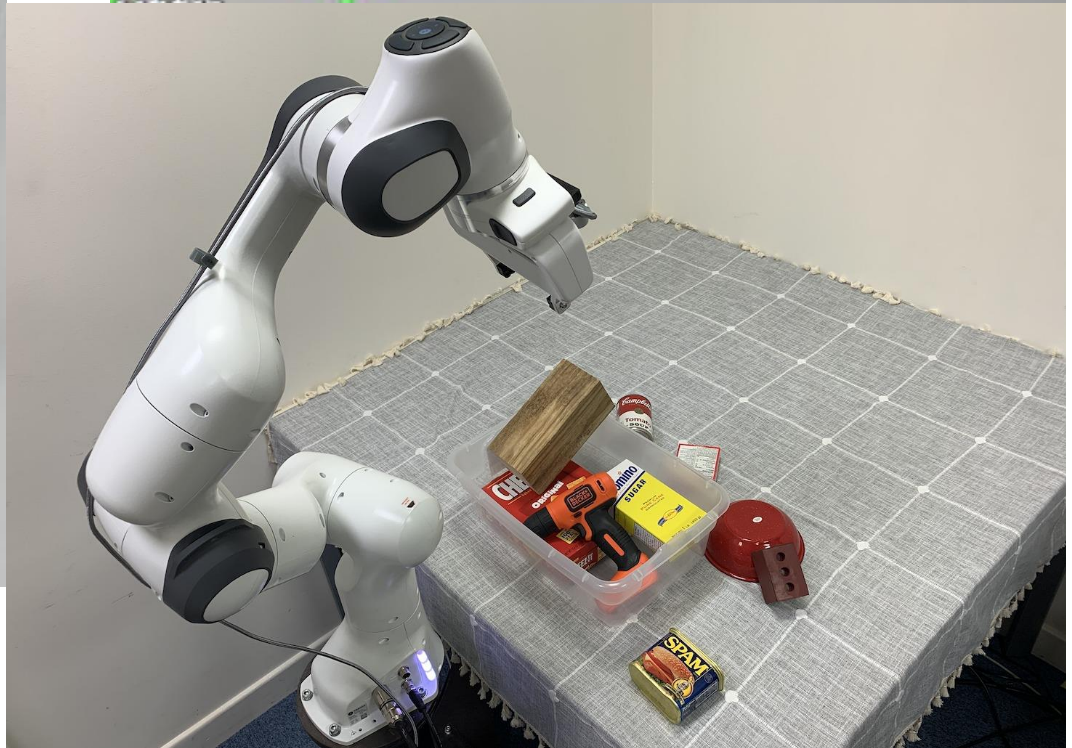
Human pose estimation

https://github.com/CMU-Perceptual-Computing-Lab/openpose

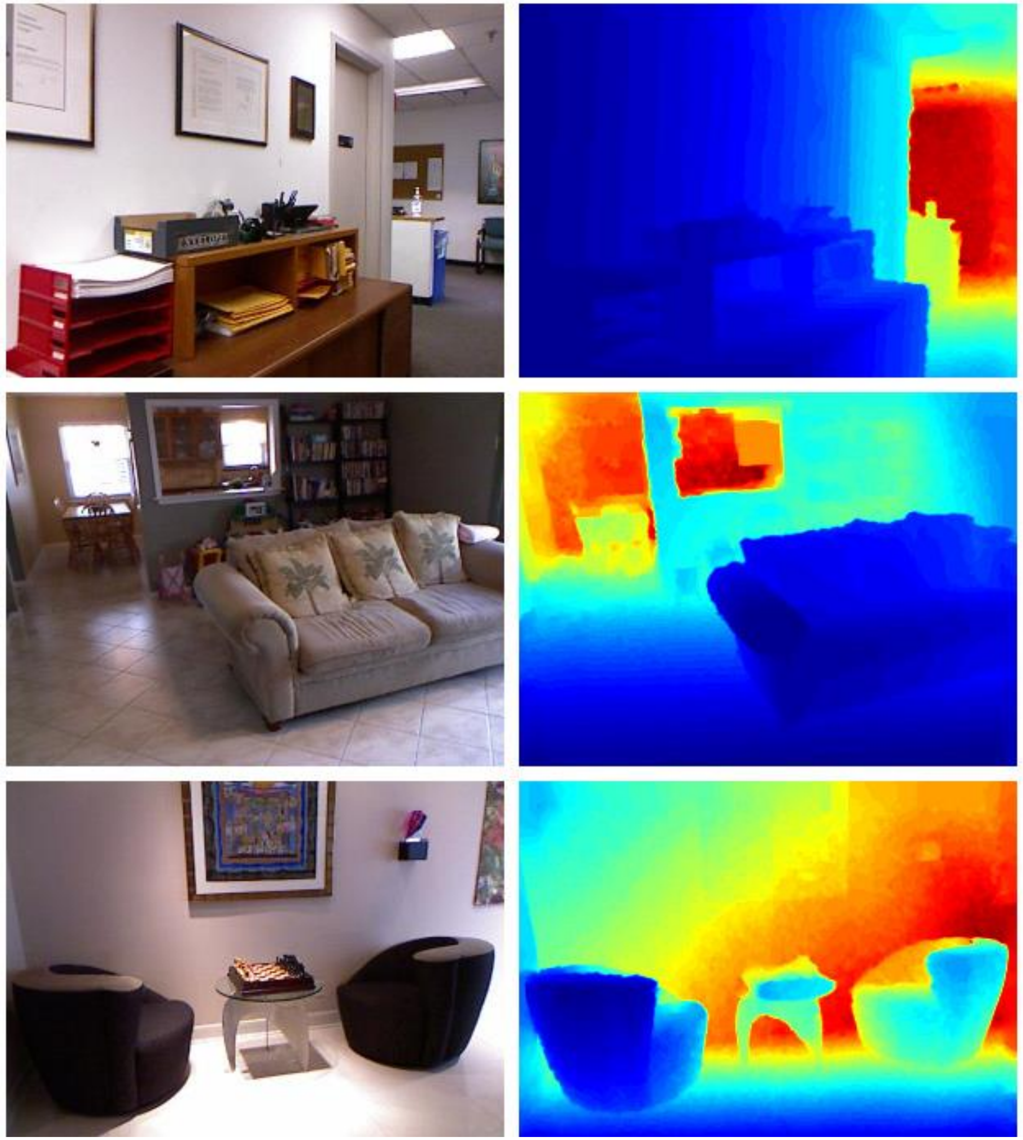
Object 6D pose estimation

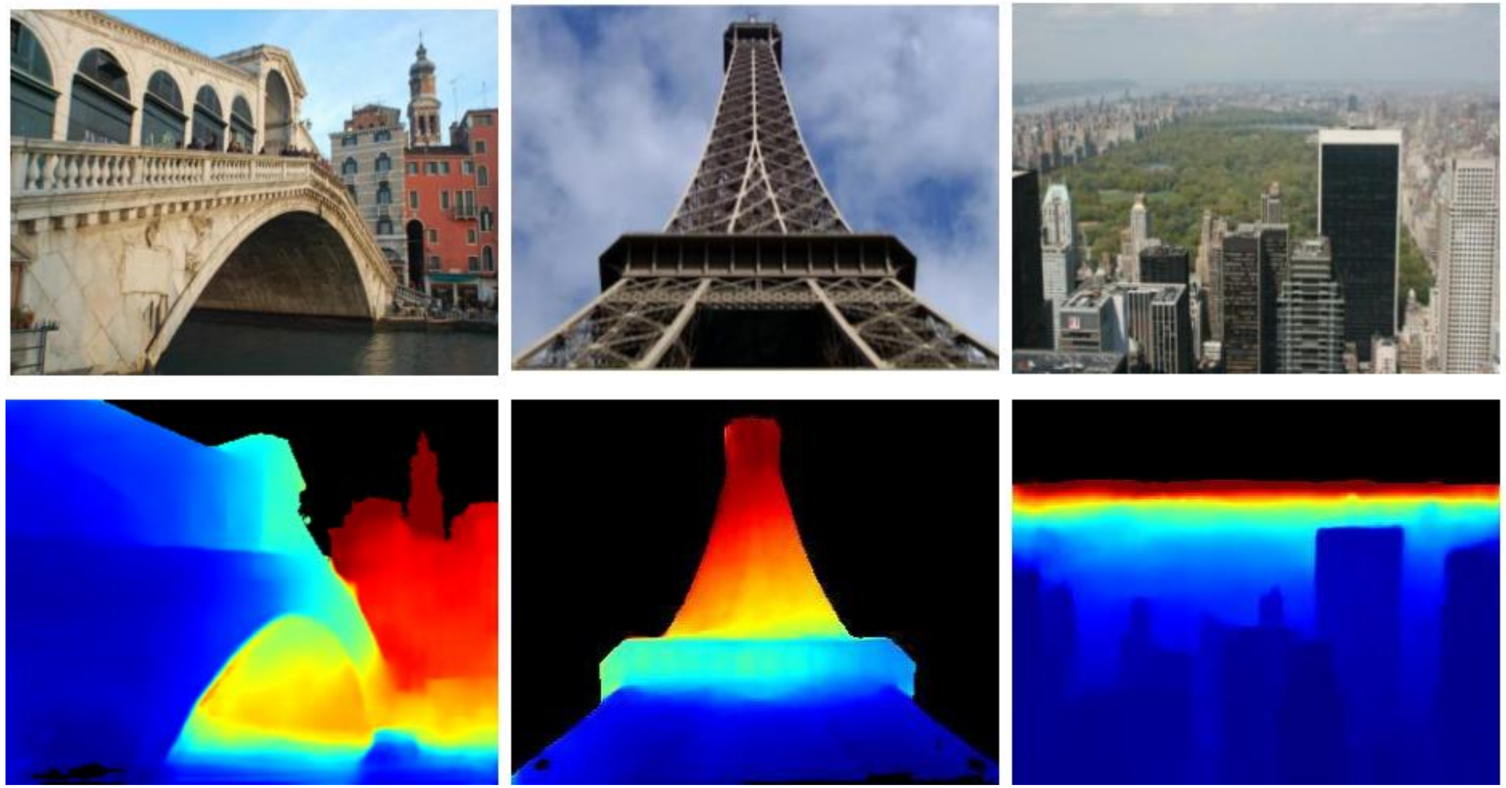






Depth estimation





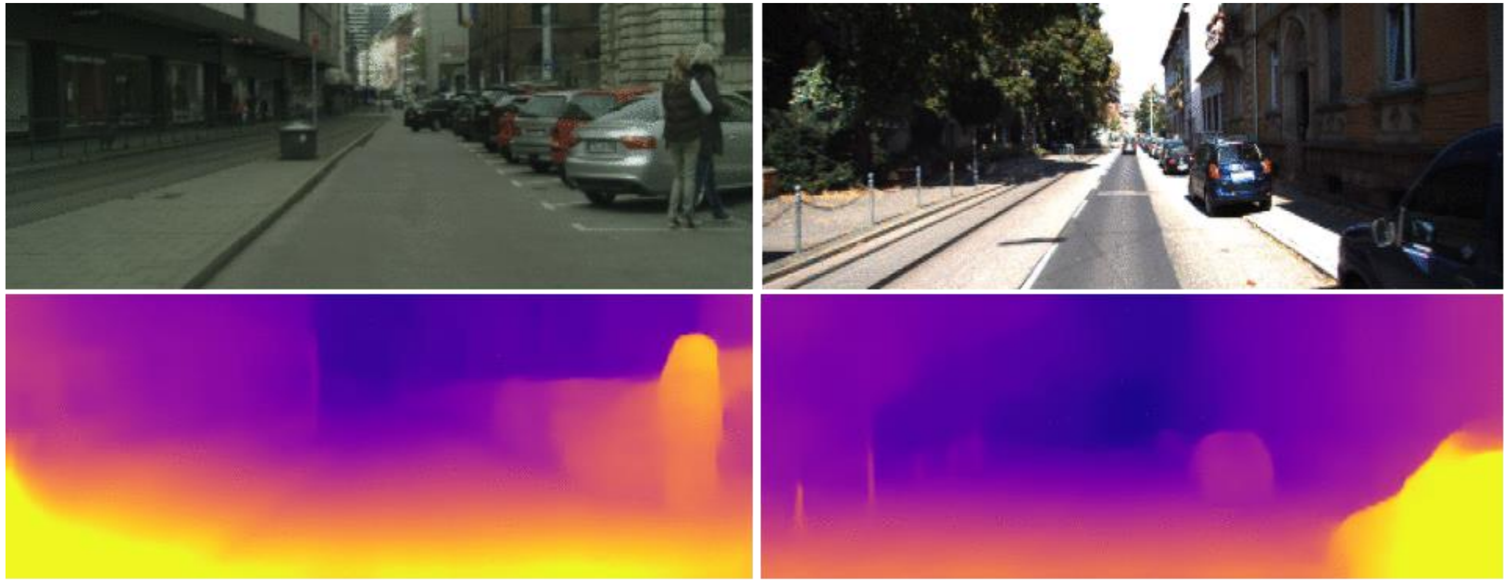
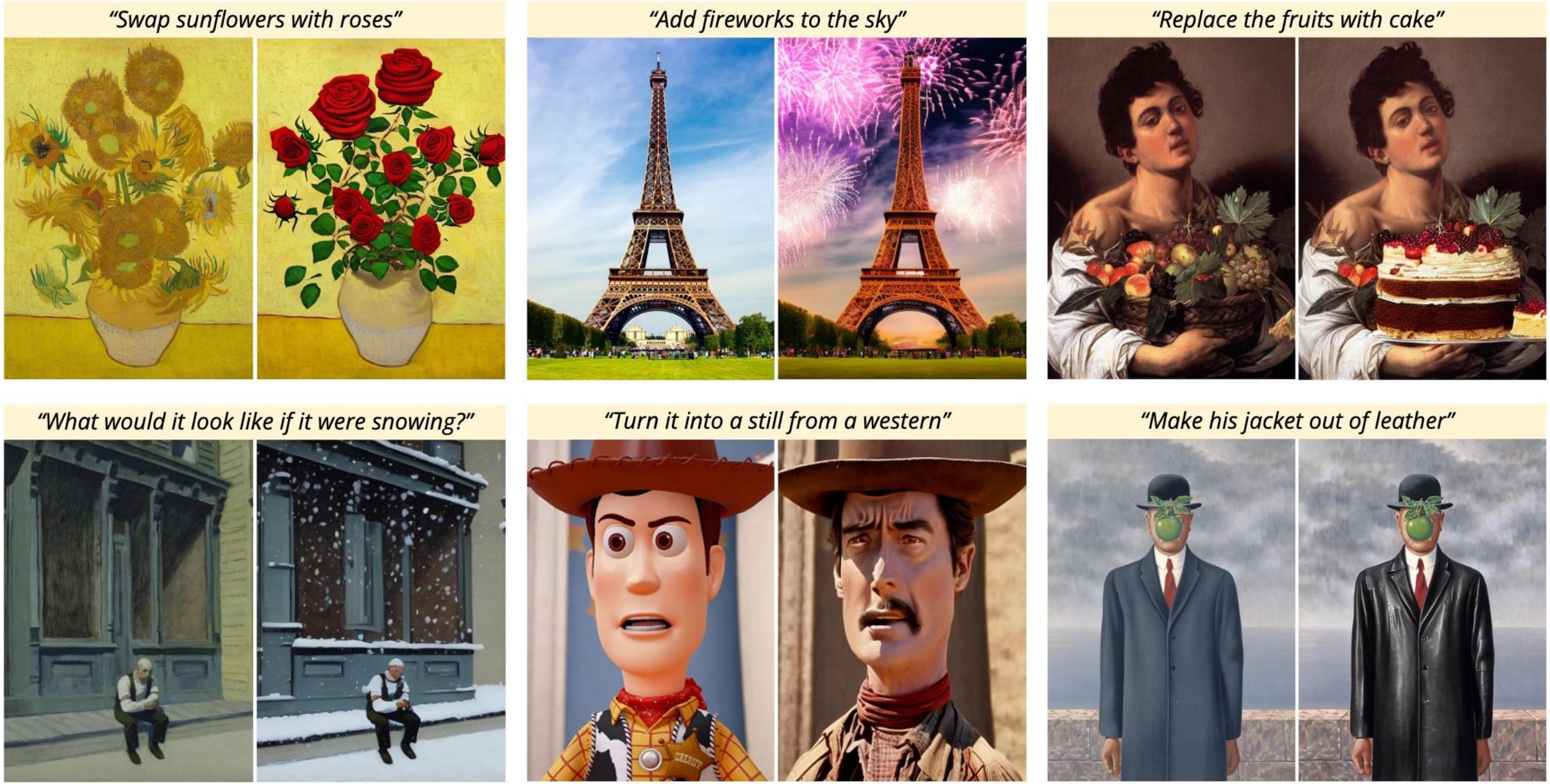


Image generation and editing

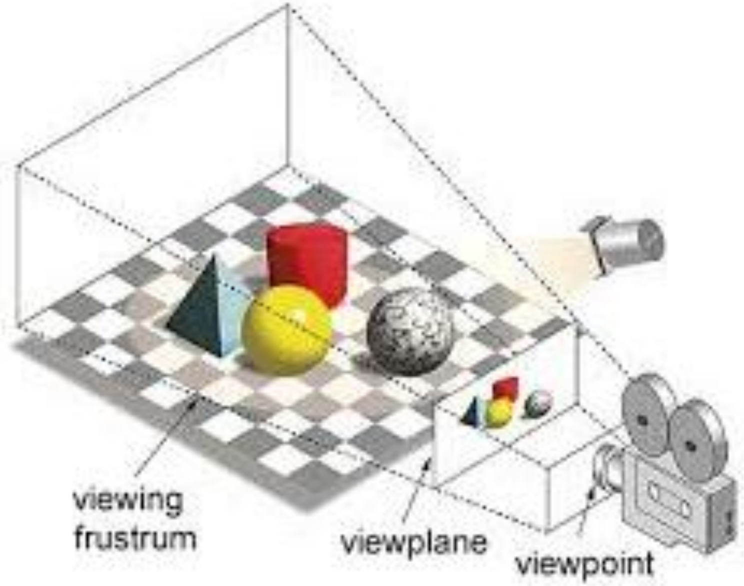


https://www.timothybrooks.com/instruct-pix2pix

Between 2D and 3D

Rendering (a Graphics problem)

- Given a 3D model of the scene (3D mesh, materials, lighting), and a camera, obtain an image

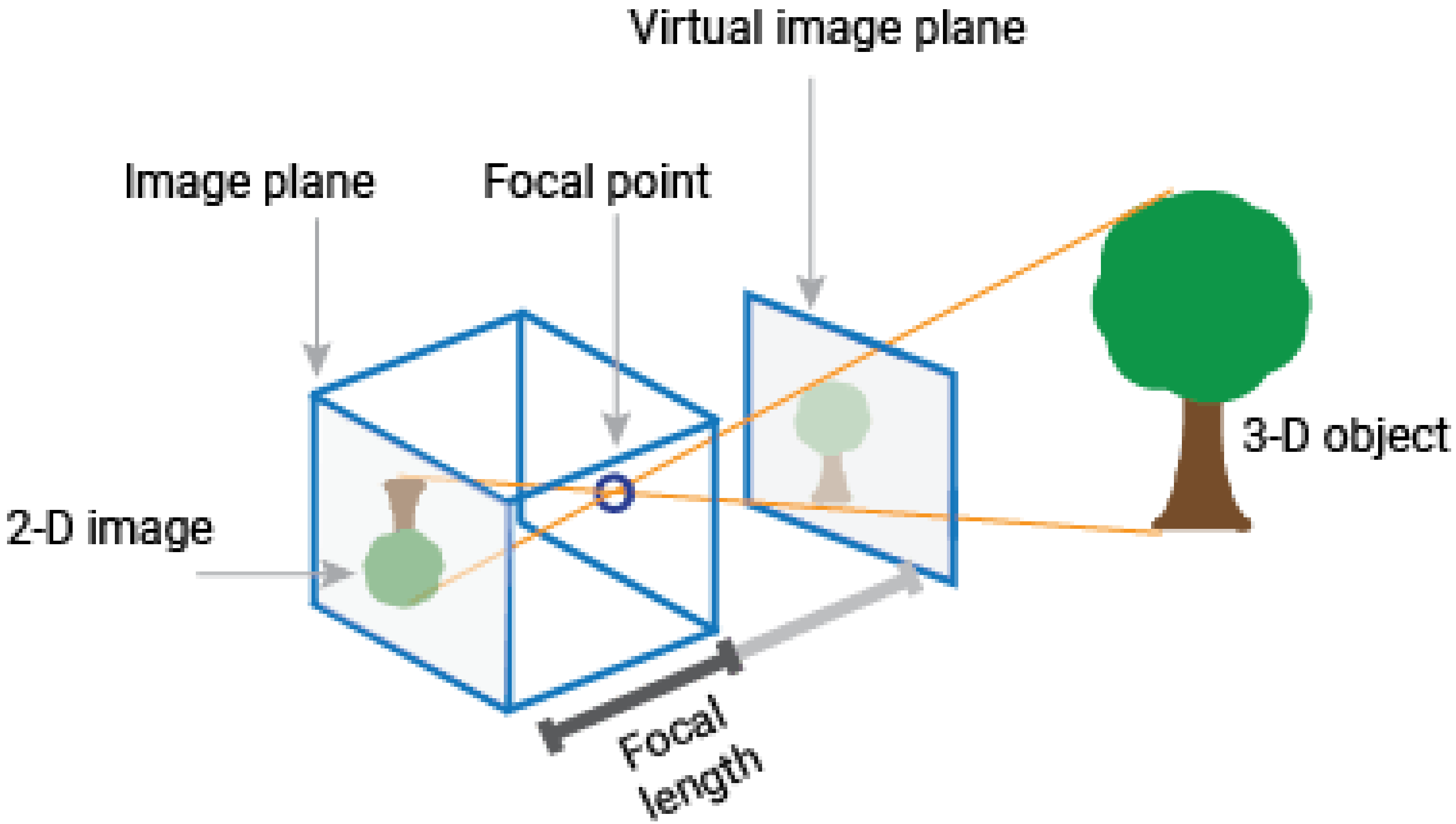


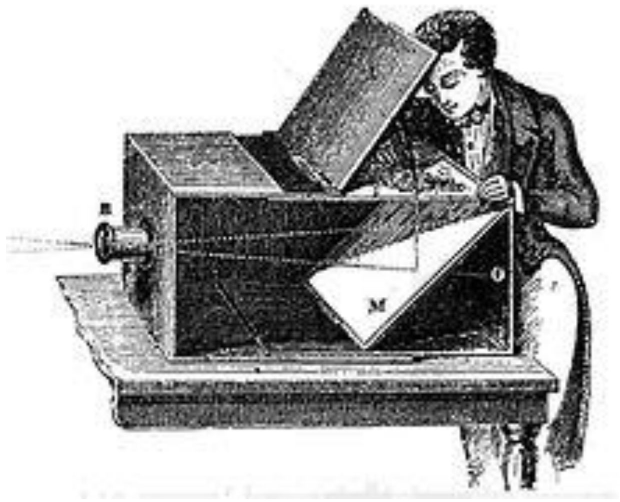
Inverse Rendering (a Vision problem)

- Given an image of a scene, infer the 3D model
- Under-constrained, ill-posed

Computer Vision: Pinhole camera model

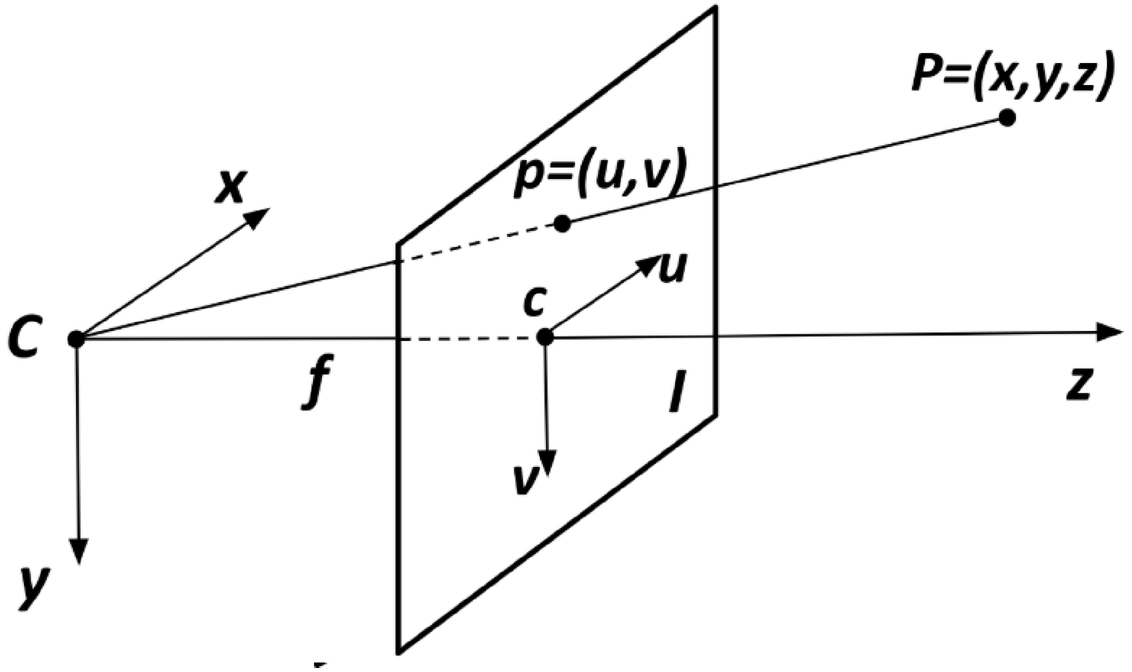
Image formation

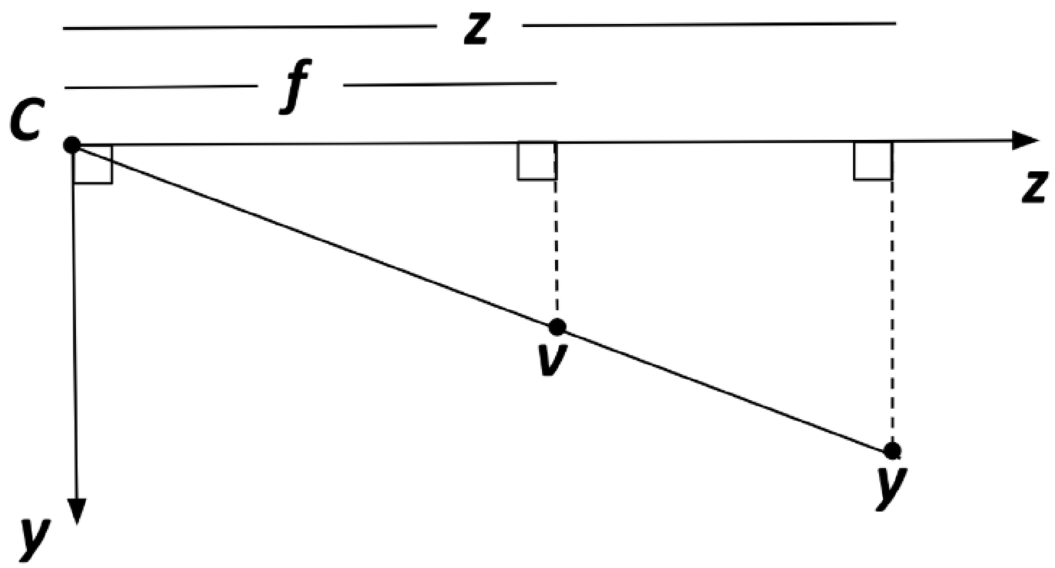




Computer Vision

The pinhole camera model - Image formation



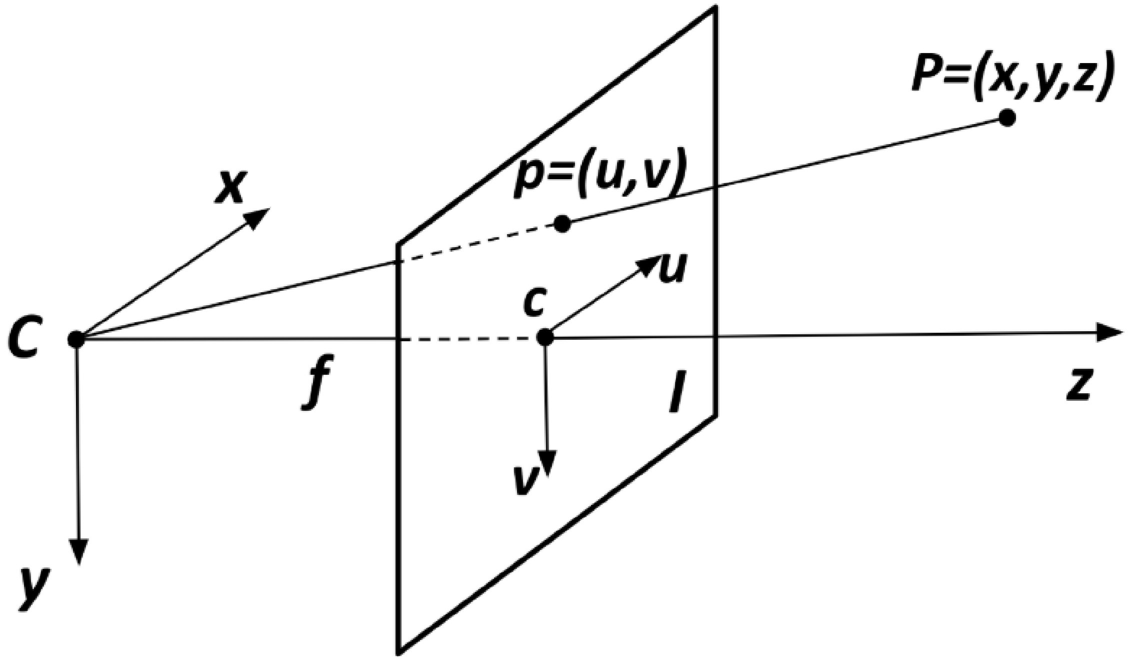


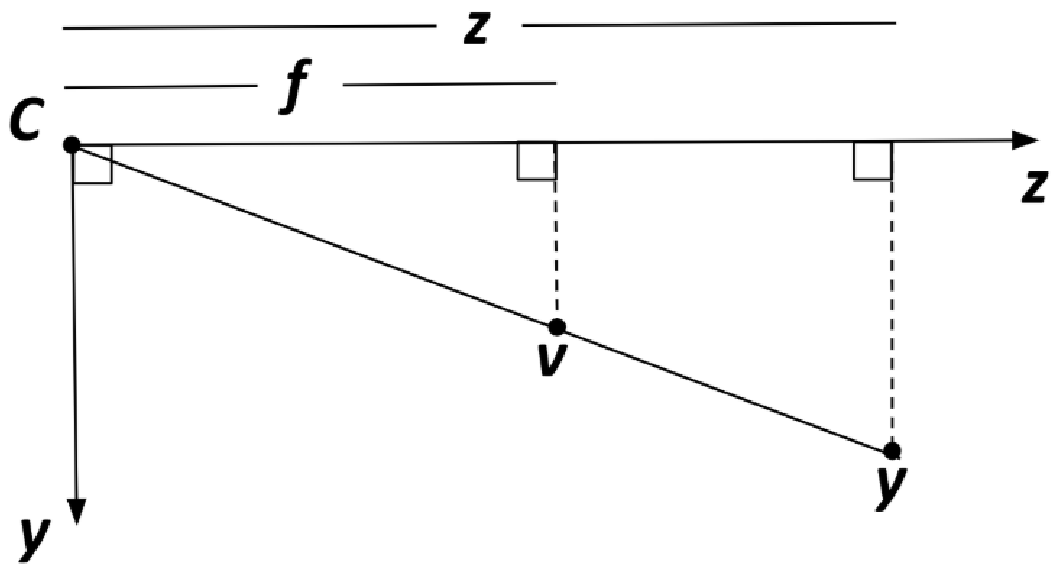
f - Focal length
z - depth

$$\frac{v}{f} = \frac{y}{z}, \frac{u}{f} = \frac{x}{z}$$

Computer Vision

The pinhole camera model - Image formation



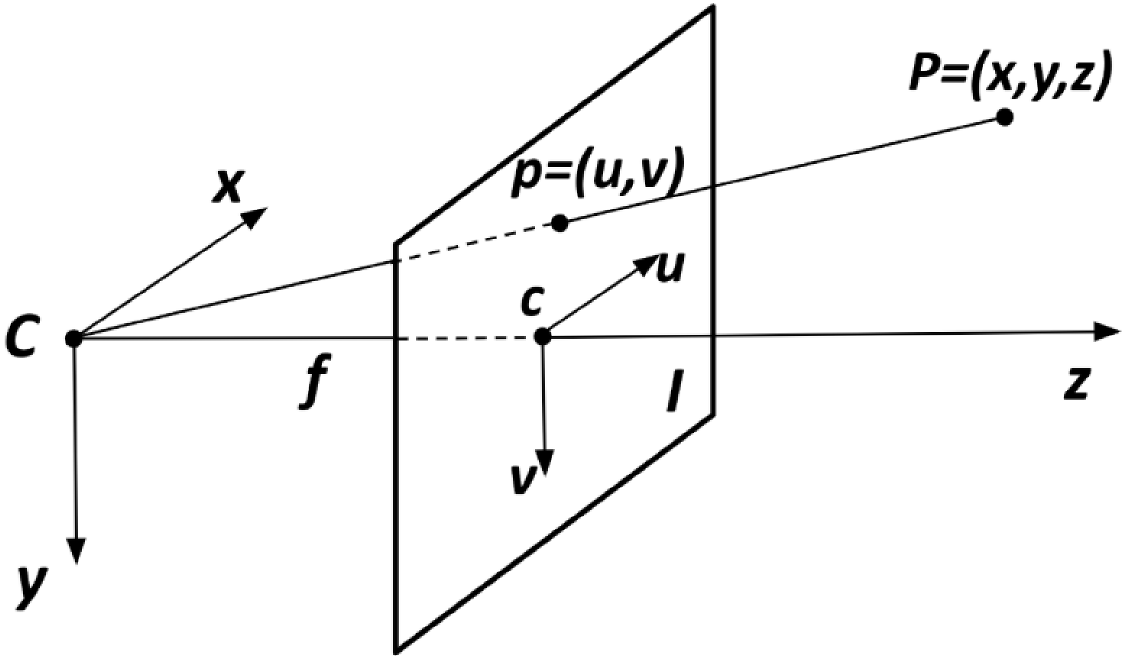


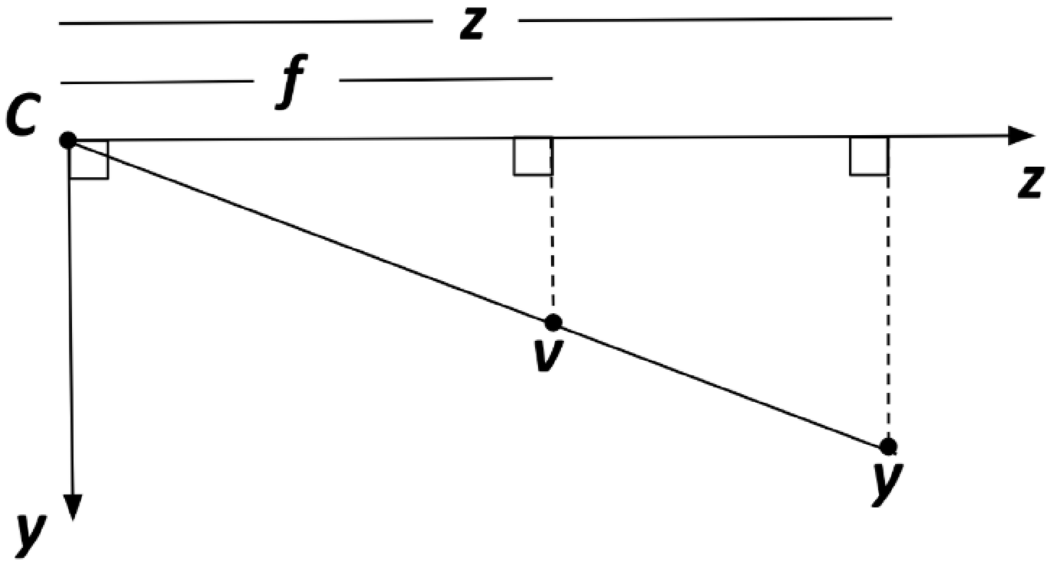
$$v = f\frac{y}{z} \qquad \text{and } u = f\frac{x}{z}$$

f - Focal length
z - depth
$c = (c_x, c_y)$ – optical center

Computer Vision

The pinhole camera model - Image formation

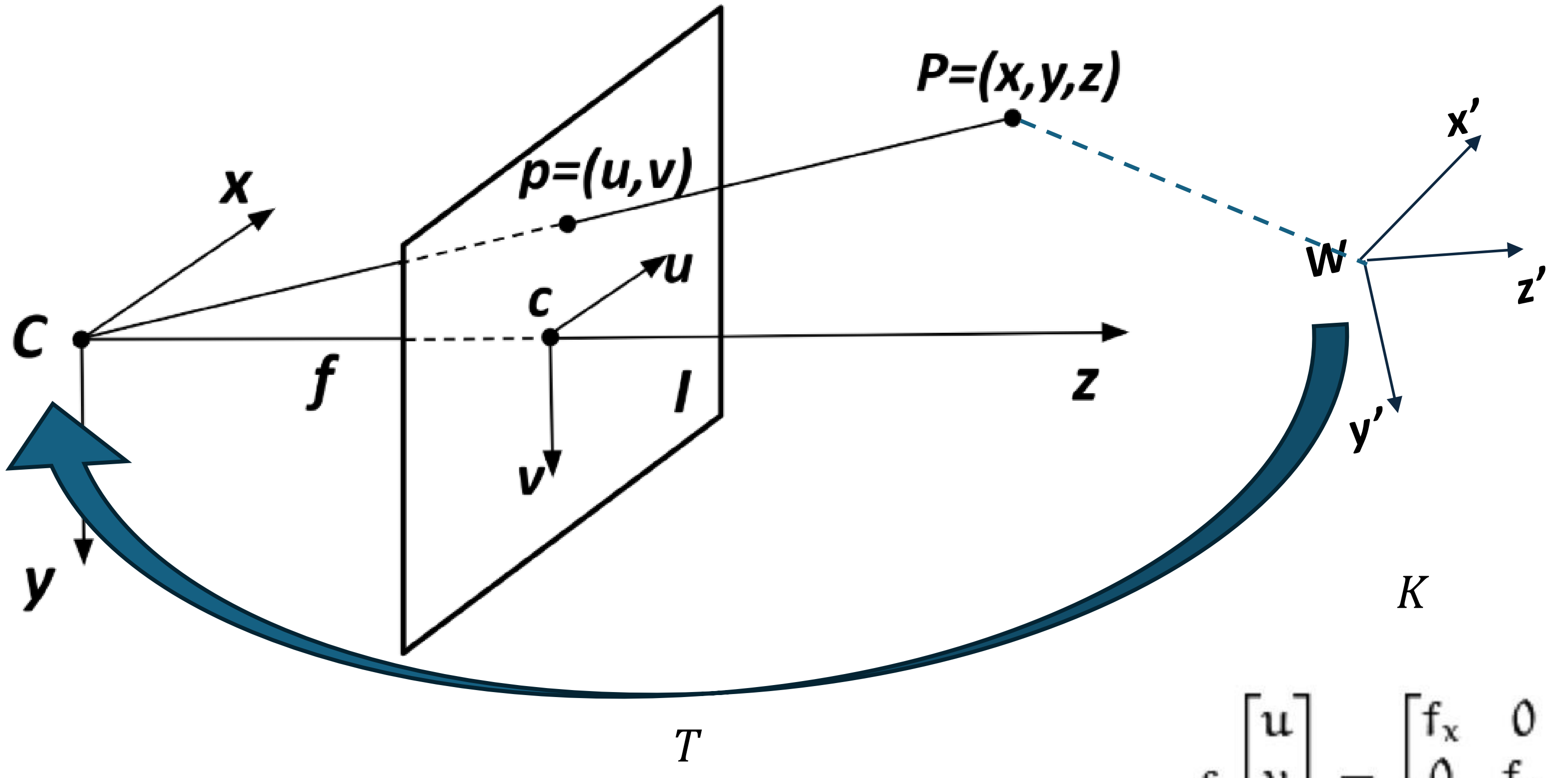




$$Z\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \overset{K}{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

f - Focal length
z - depth
$c = (c_x, c_y)$ – optical center

Pinhole camera model



$$
s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \overset{K}{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}} \overset{\boldsymbol{R}}{\boxed{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}}} \overset{\boldsymbol{T}}{\boxed{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}}
$$

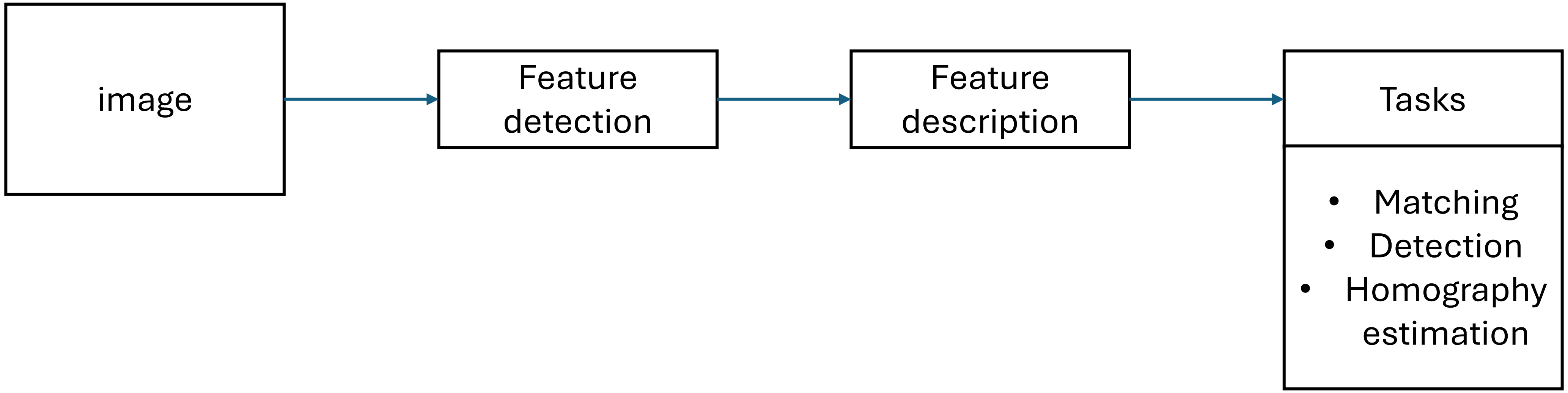
2D Image Coordinates

Intrinsic properties (Optical Centre, scaling)

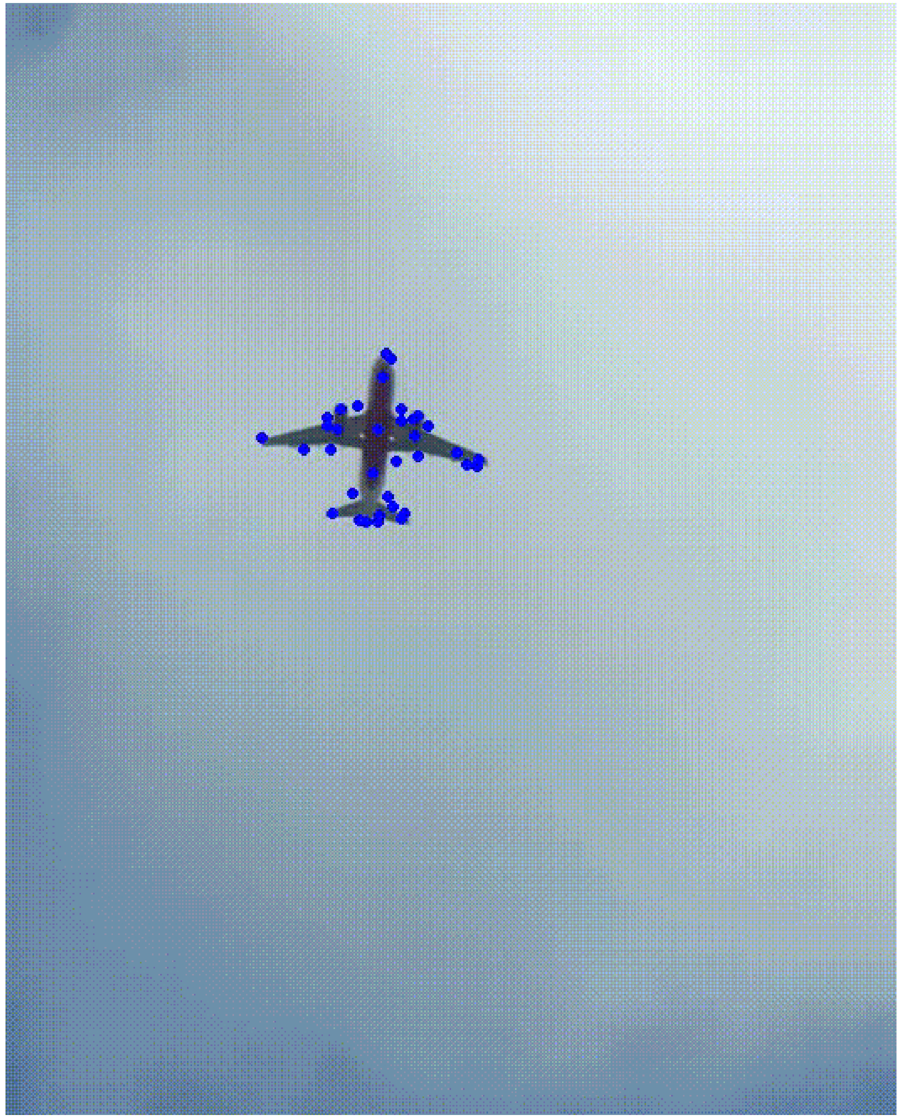
Extrinsic properties (Camera Rotation and translation)

3D World Coordinates

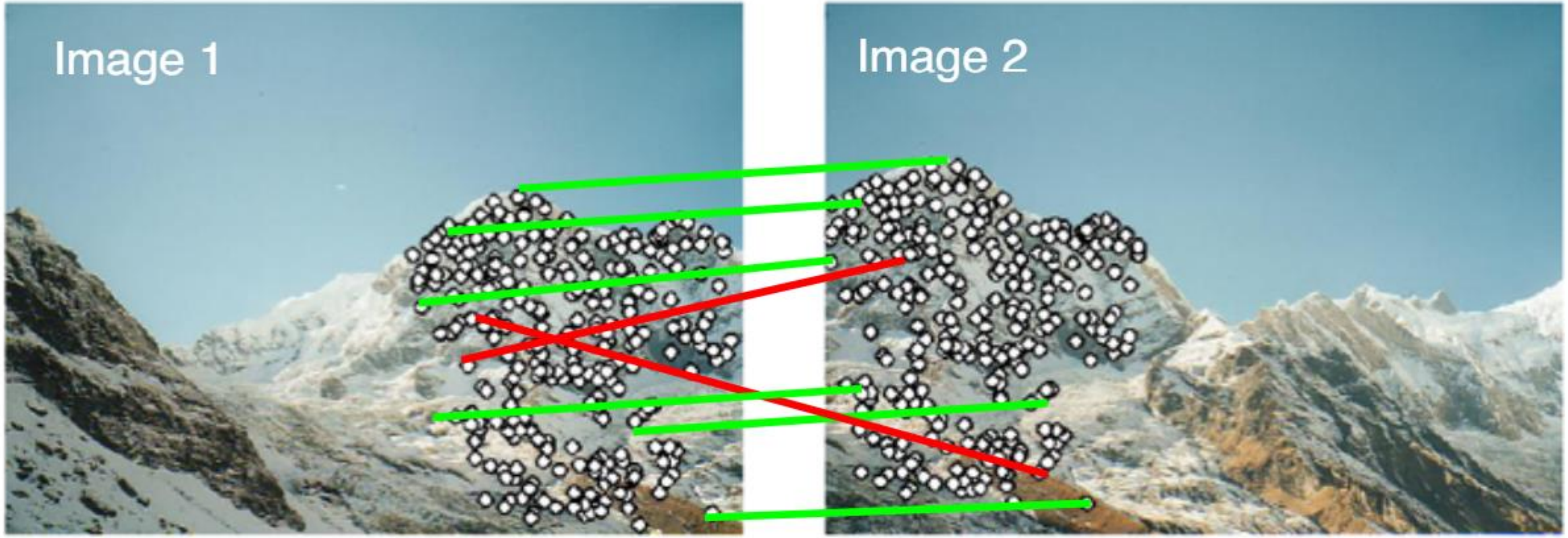
Classic understanding of images



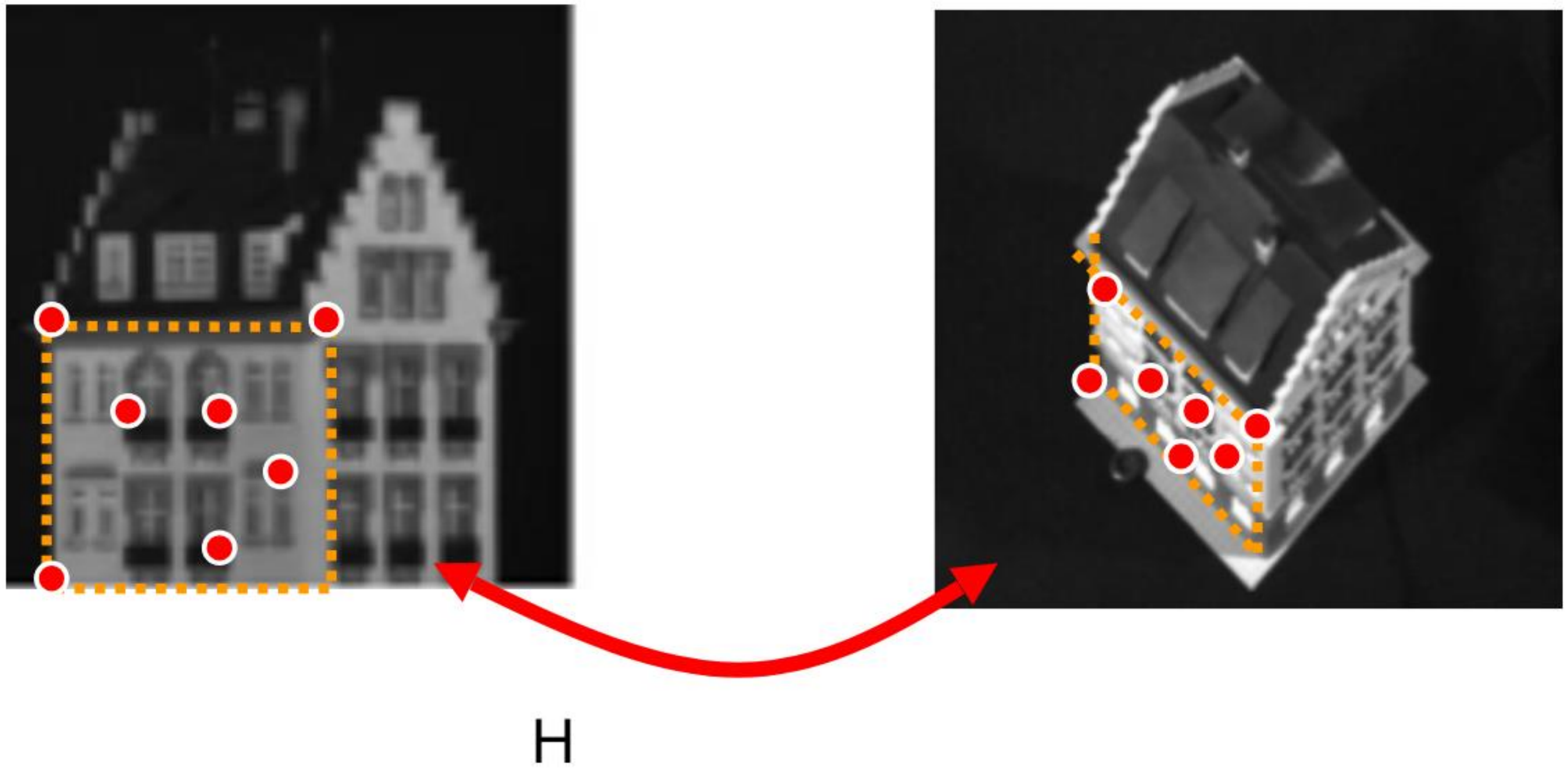
Object detection and tracking



Matching for image stitching



Homography estimation

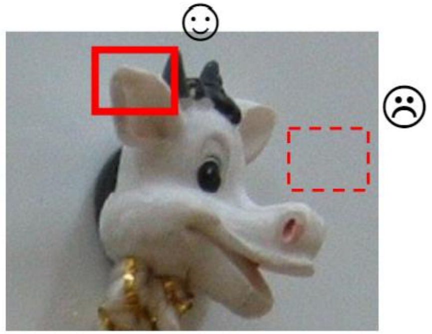


Edge detection

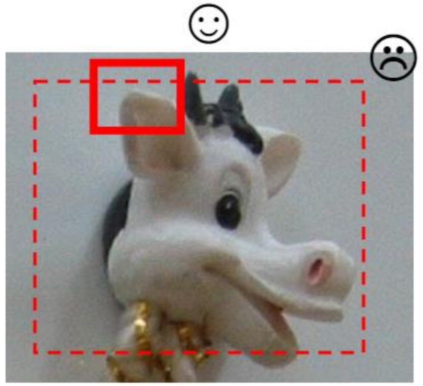
- Use derivatives (in x and y direction) to obtain pixels with high gradient
- Need smoothing to reduce noise prior to taking derivative
- E.g. Canny Edge Detector

Corner Detection

- **Repeatability** – The same feature can be found in several images despite geometric and photometric transformations



- **Saliency** – Each feature is found at an "interesting" region of the image

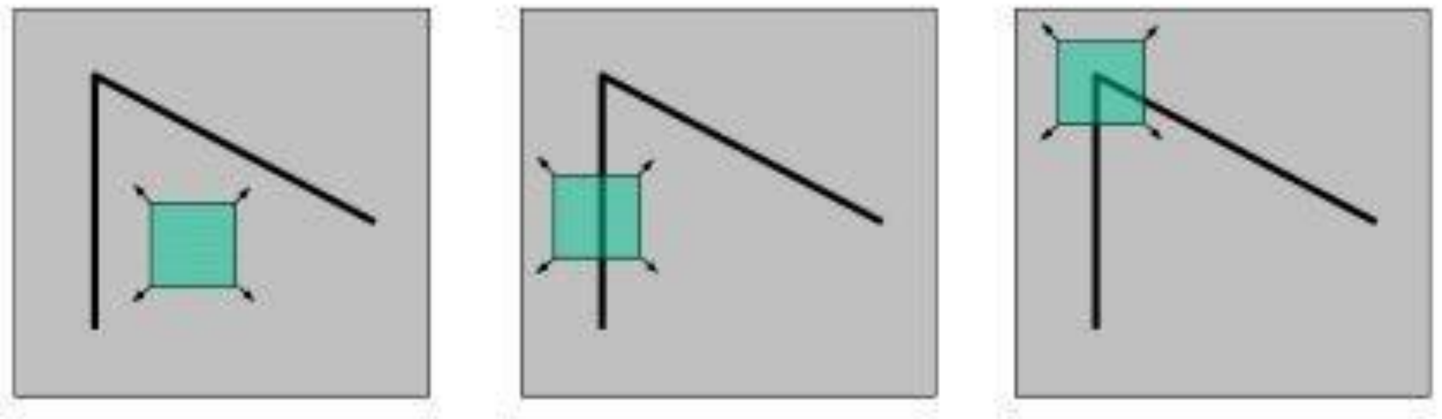


- **Locality** – A feature occupies a "relatively small" area of the image

Source: https://cvgl.stanford.edu/teaching/cs231a_winter1415/lecture/lecture10_detector_descriptors_2015.pdf

Harris Corner Detector

Explore intensity changes within a window as the window changes location



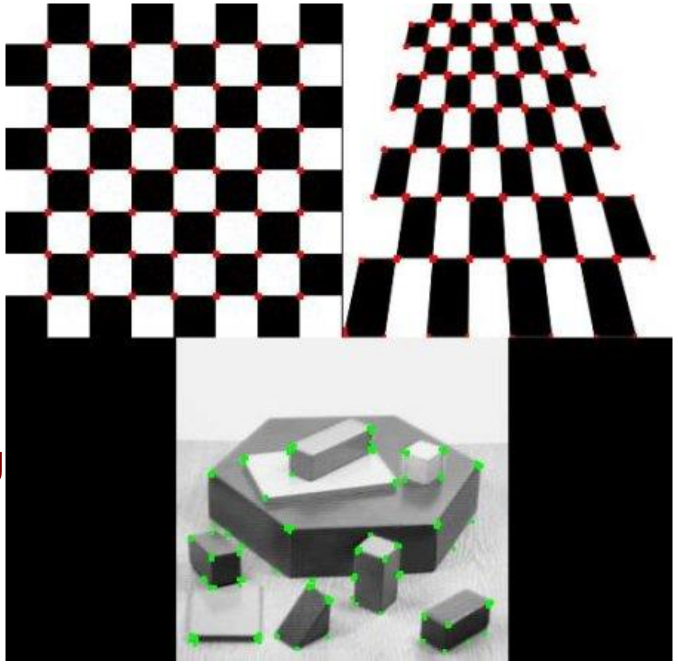
"flat" region:
no change in all
directions

"edge":
no change along the
edge direction

"corner":
significant change in
all directions

Harris Corner Detector – Code example OpenCV

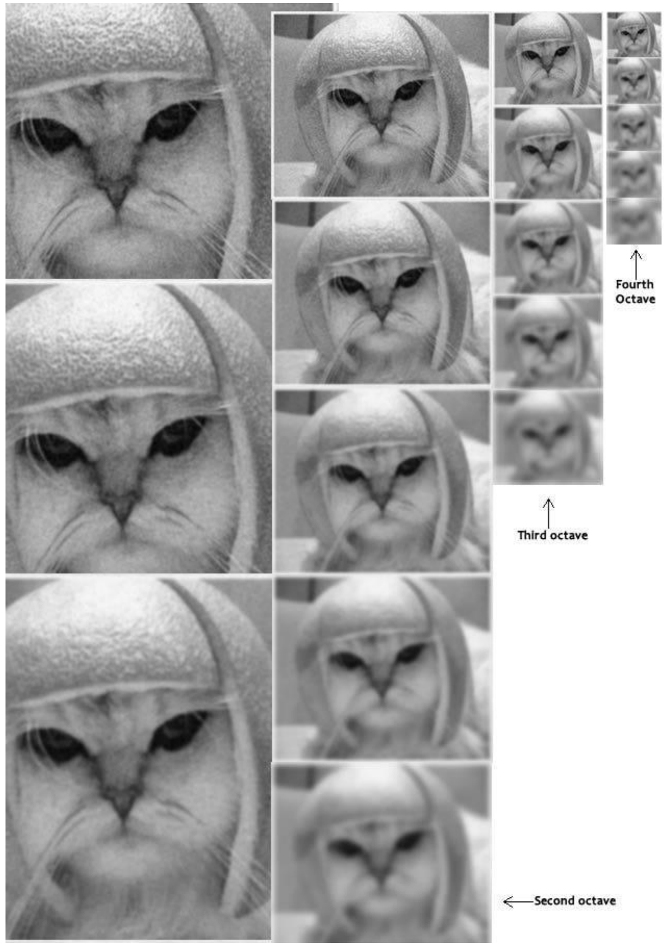
```
import numpy as np
import cv2 as cv
filename = 'chessboard.png'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)
#result is dilated for marking the corners, not important
dst = cv.dilate(dst,None)
# Threshold for an optimal value, it may vary depending on the imag
img[dst>0.01*dst.max()]=[0,0,255]
cv.imshow('dst',img)
if cv.waitKey(0) & 0xff == 27:
    cv.destroyAllWindows()
```

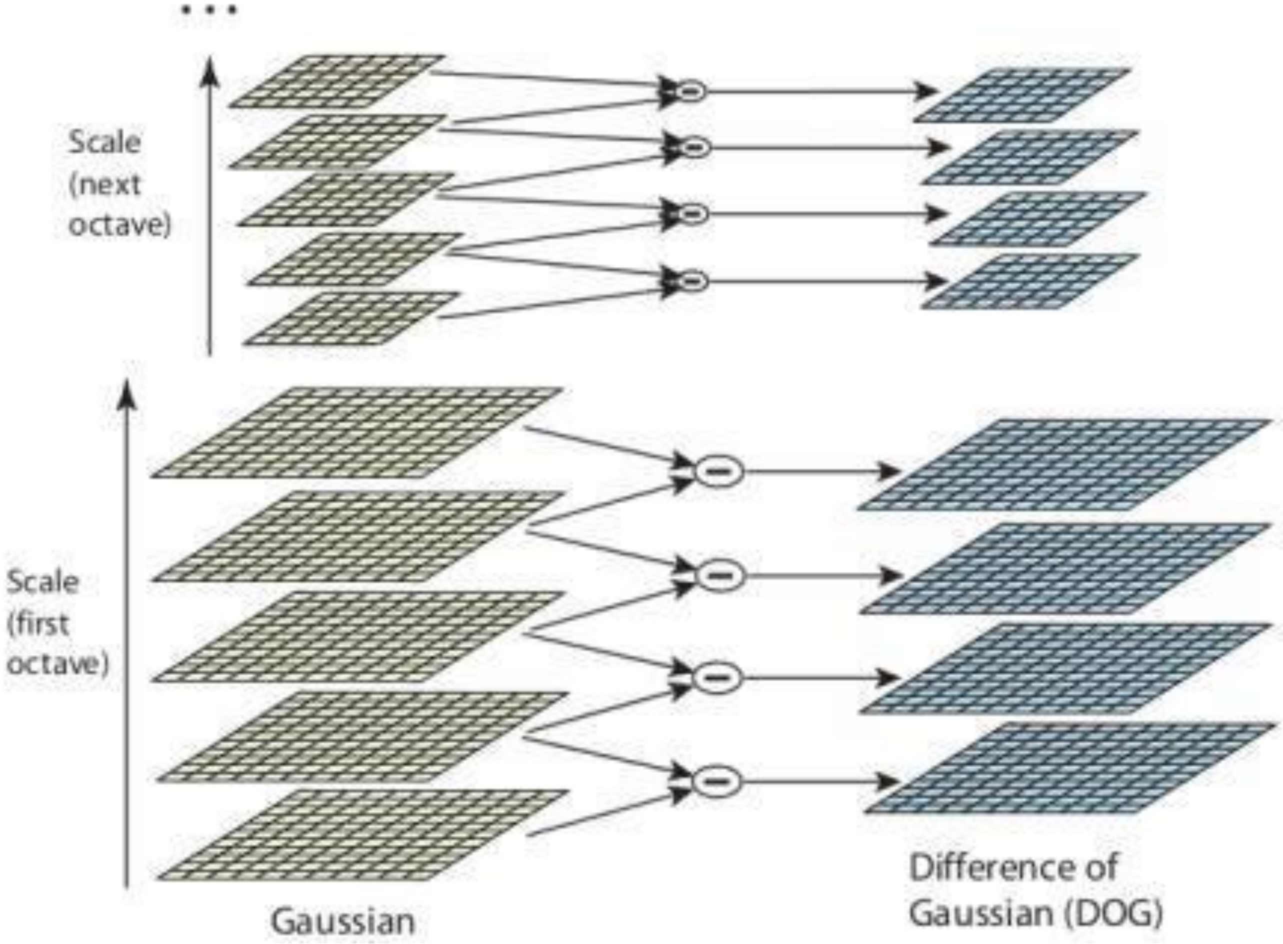


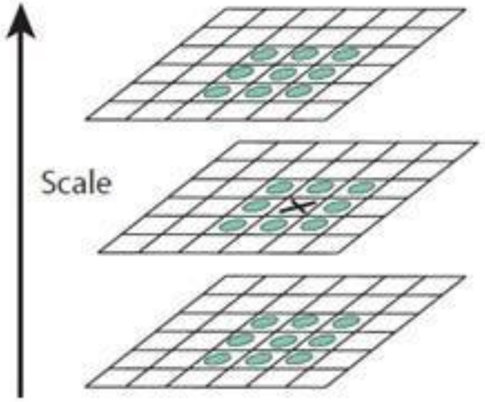
https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html

Difference of Gaussians (DoG)

Obtain Image in different scales and different amount of Gaussian blur
Keypoints obtained by computing difference of Gaussians in each scale
Choose best scale to represent that keypoint - the scale that contains a spatial gradient maxima







Feature Descriptors

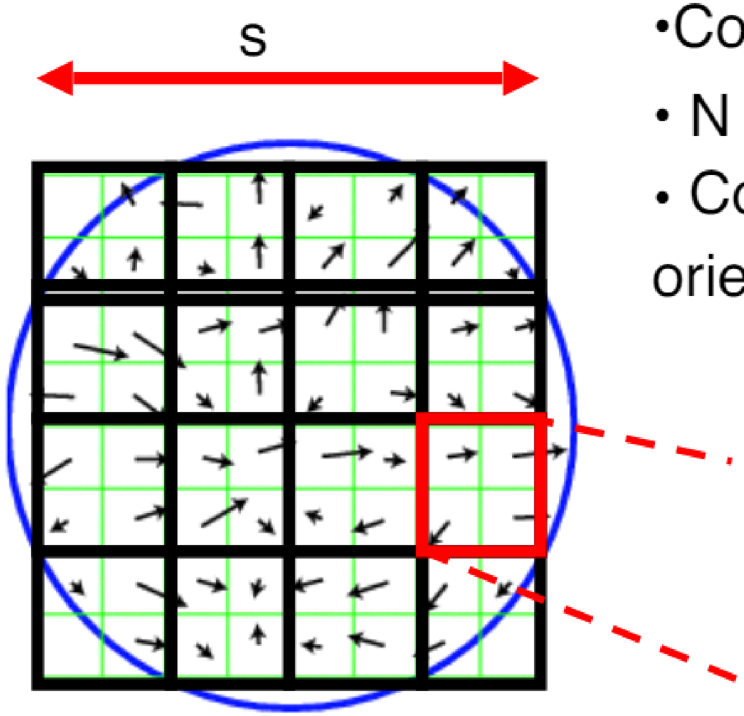
*Why do we need them? T*o match relevant/corresponding image parts based on their feature similarity

Properties

1. Information that is **invariant** w.r.t: illumination, pose, scale, intraclass variability
2. Highly **distinctive**: allows for finding the correct match with a good probability

SIFT Descriptor

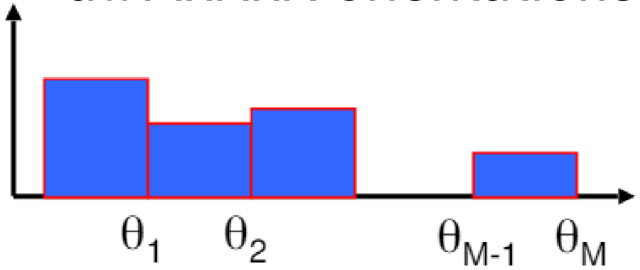
- Based on DoG keypoints
- Location and characteristic scale s given by DoG detector (**scale invariant**)



•Compute gradient at each pixel

- N x N spatial bins
- Compute an histogram h_i of M orientations for each bin i

Obtain feature vector by concatenating all MxNxN orientations



- **Rotation invariant**: Find dominant orientation by building a orientation histogram. Rotate all orientations by the dominant orientation

David G. Lowe. "Distinctive image features from scale-invariant keypoints." IJCV 60 (2), 04

Other feature descriptors

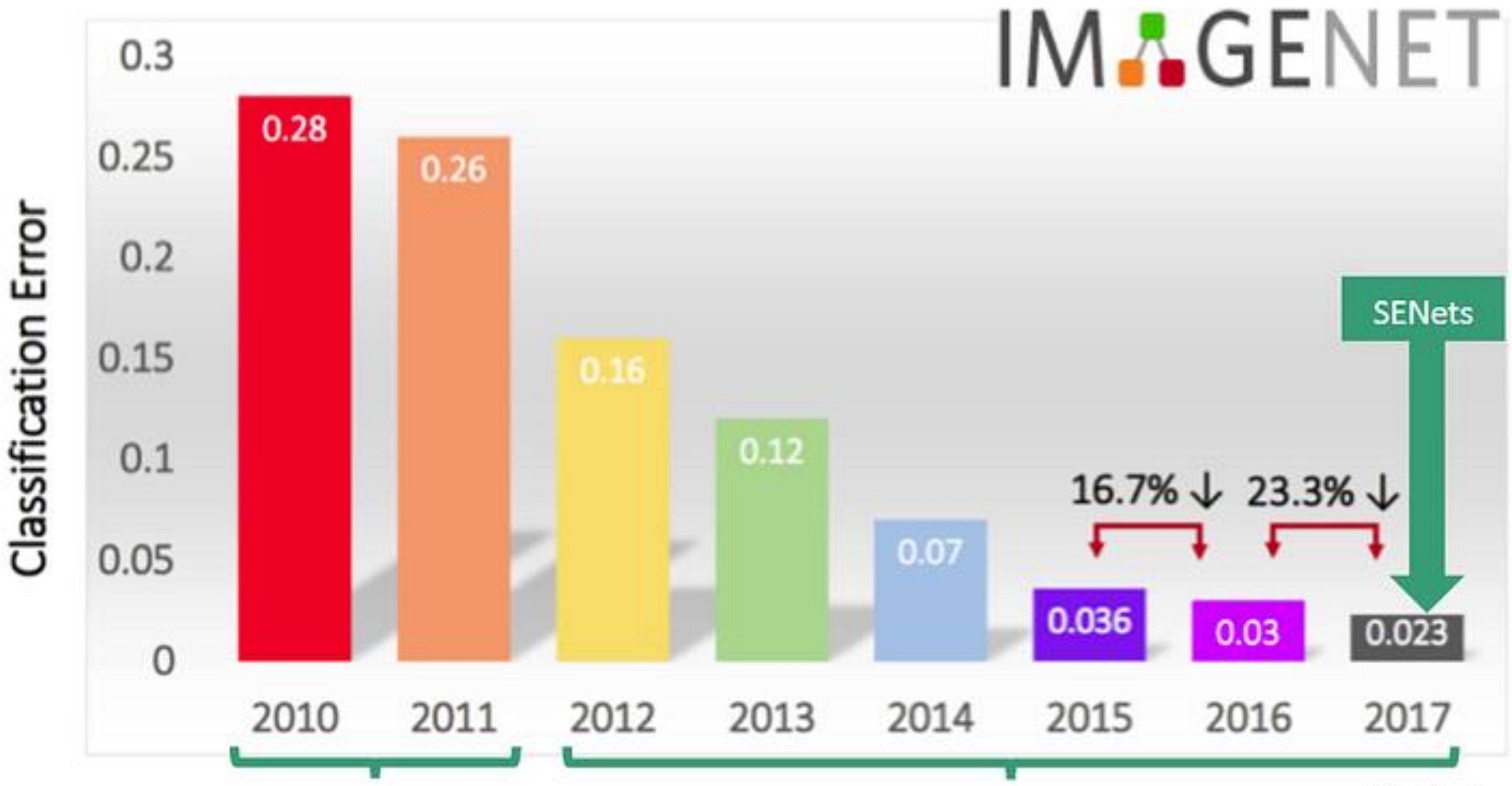
- HoG (Histogram of oriented gradients)
- SURF (Speeded Up Robust Features)
- ORB (an efficient alternative to SIFT or SURF)
- FREAK (Fast Retina Keypoint)

Summary of feature detectors/descriptors

- Based on spatial derivatives and local smoothing filters
- Based on expert knowledge
- Require some heuristic thresholds and filtering steps (not all keypoints are relevant)
- They are handcrafted to support desired properties, i.e. scale, rotation, illumination invariance.
- Hard to come up with rules that generalize well to all scenarios!

- What’s next?

Image classification performance



[Statistics provided by ILSVRC]

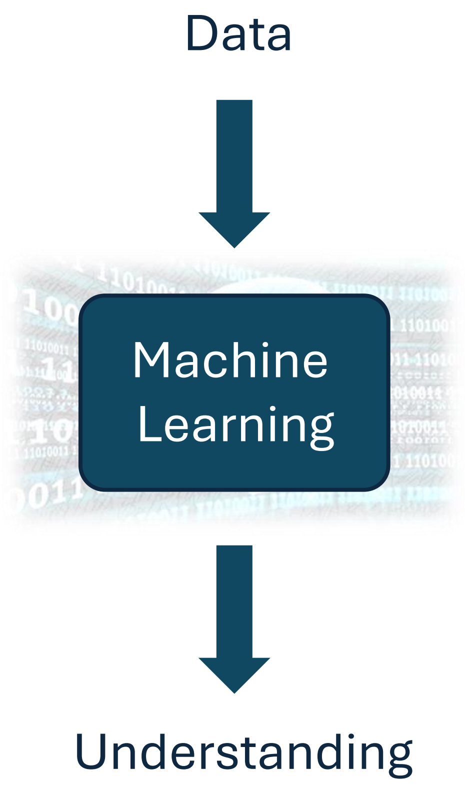
Deep Learning

A type of machine learning based on artificial neural networks in which multiple layers of processing are used to extract progressively higher level features from data.

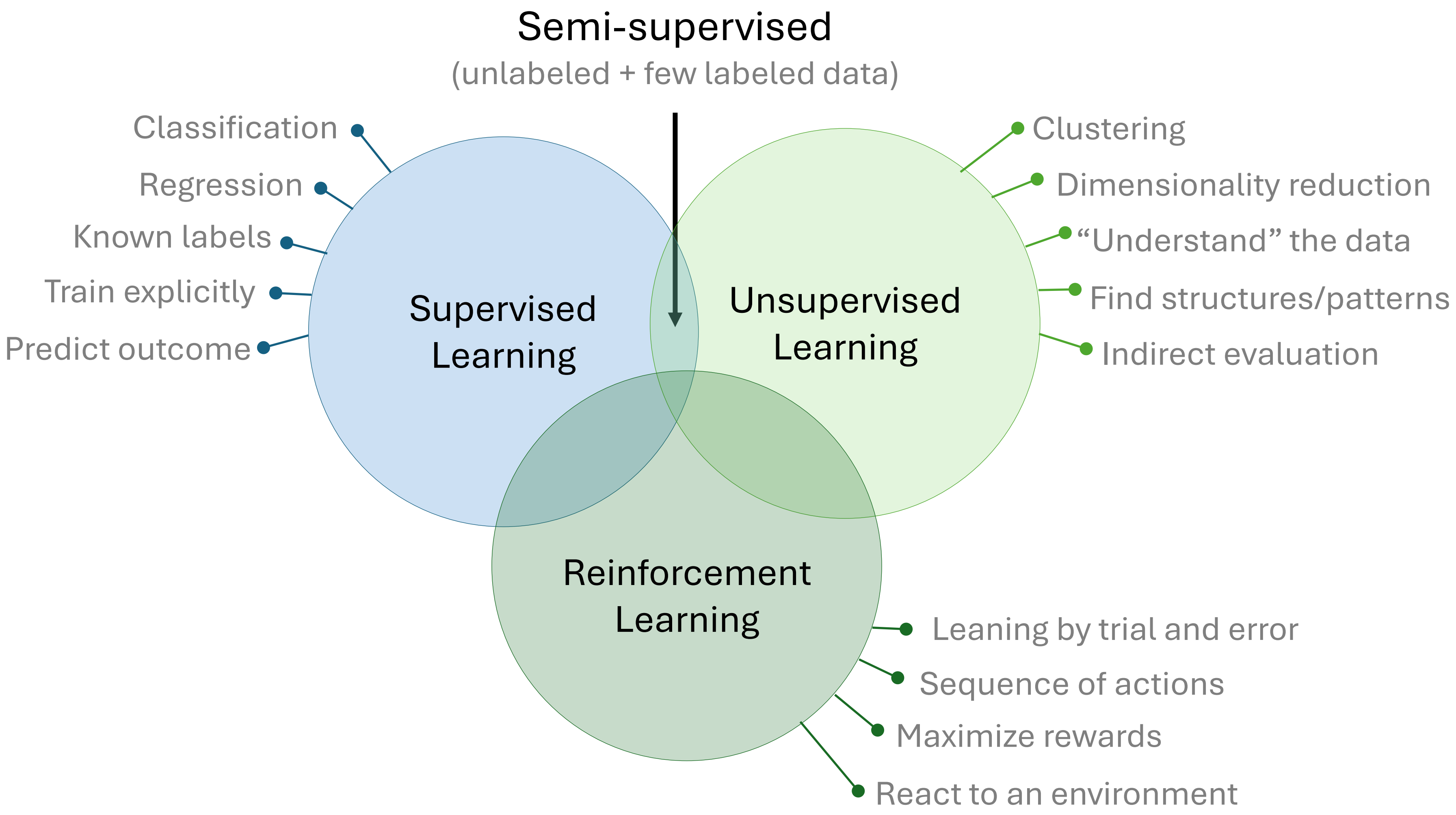
Machine Learning

- [Arthur Samuel, 1959]
  - Field of study that gives computers the ability to learn without being explicitly programmed
- [Kevin Murphy]
  Algorithms that
  - automatically *detect patterns* in data
  - use the uncovered patterns to *predict* future data or other outcomes of interest
- [Tom Mitchell]
  Algorithms that
  - learn from experience (E)
  - with respect to some class of tasks (T)
  - to improve their performance (P)

Slide Credit (edited): Dhruv Batra



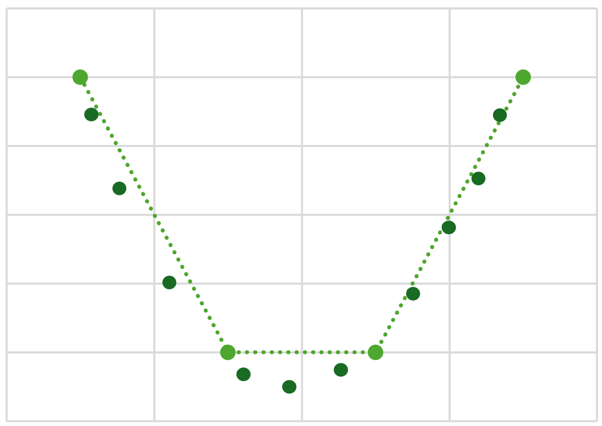
Machine Learning

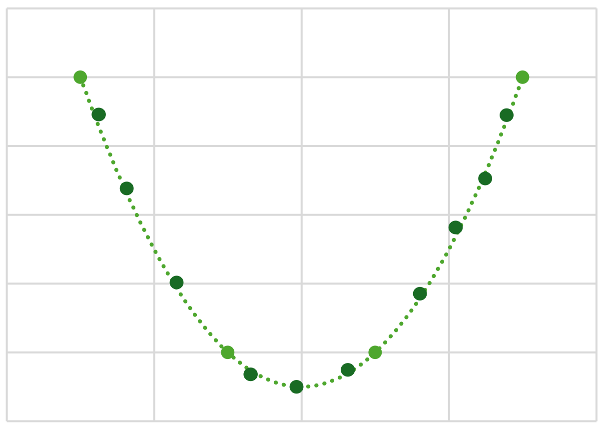


Supervised Learning

Learning from Examples

- Training set of N samples $\left(x^{(i)}, y^{(i)}\right)$
- Generated by unknown function f s.t. $f\left(x^{(i)}\right) = y^{(i)} \ \forall i$
- $x^{(i)}$: input, $y^{(i)}$: expected outcome
- Discover/Learn f^* that approximates f
- Given a **new** $x^{(j)}$ **with unknown** $y^{(j)}$ compute $y^{(j)}$ as $y^{(j)} = f^*(x^{(j)})$

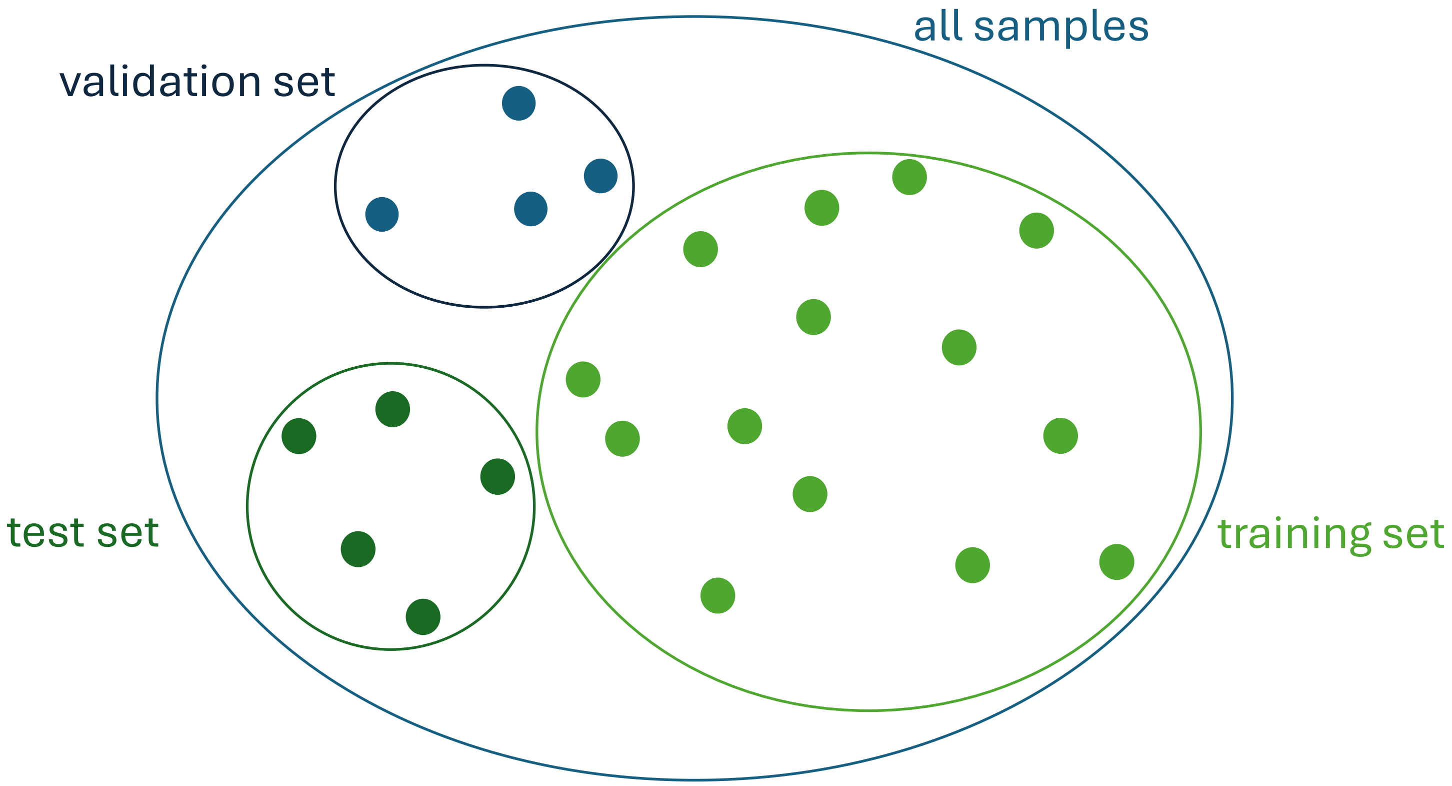




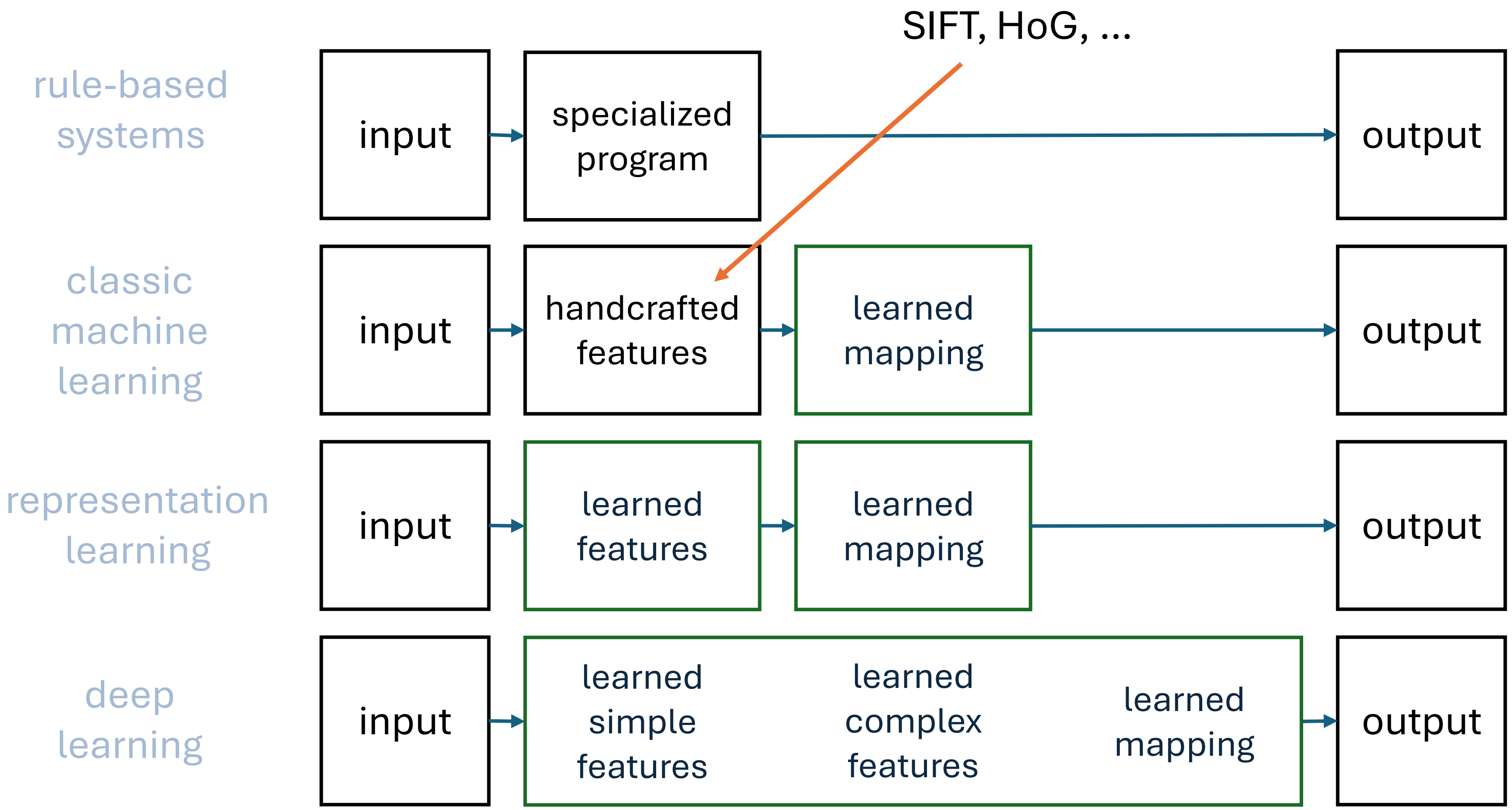
How can we assess the quality of the learned f?

Training – Validation – Testing

- Use only a subset of the samples for learning f (training set)
- The rest is for testing the quality of the predicted f^* (test set)
- If the learning process has parameters: split the training set again and tune the parameters on left-out subset (validation set)



Why is deep learning attractive?



Artificial Neural Networks

"...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs."

Dr. Robert Hecht-Nielsen in "Neural Network Primer: Part I" by Maureen Caudill, Feb. 1989

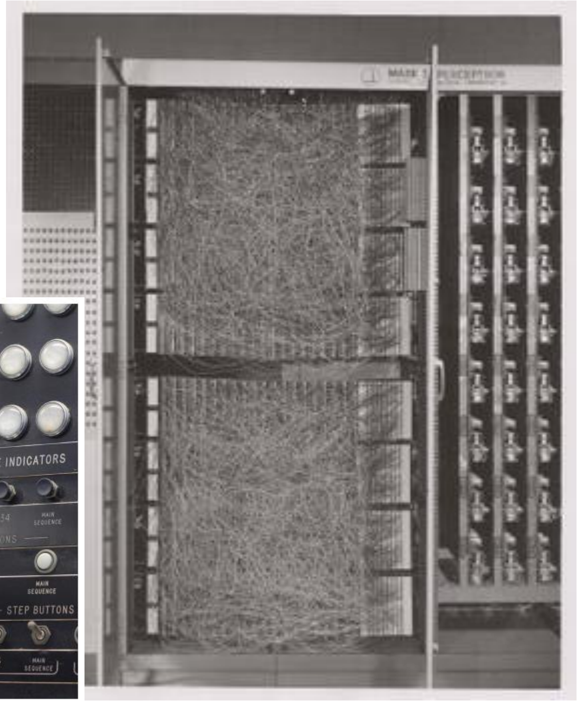
The Perceptron

“the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.”

Frank Rosenblatt acc. to New York Times from
Mikel Olazaran (1996). "A Sociological Study of the Official History of the Perceptrons Controversy". *Social Studies of Science* 26 (3): 611–659

Mark I Perceptron

- Frank Rosenblatt (1957)
- Image recognition
- 20 x 20 photo cells
- Learning with motors attached to potentiometers



The Perceptron

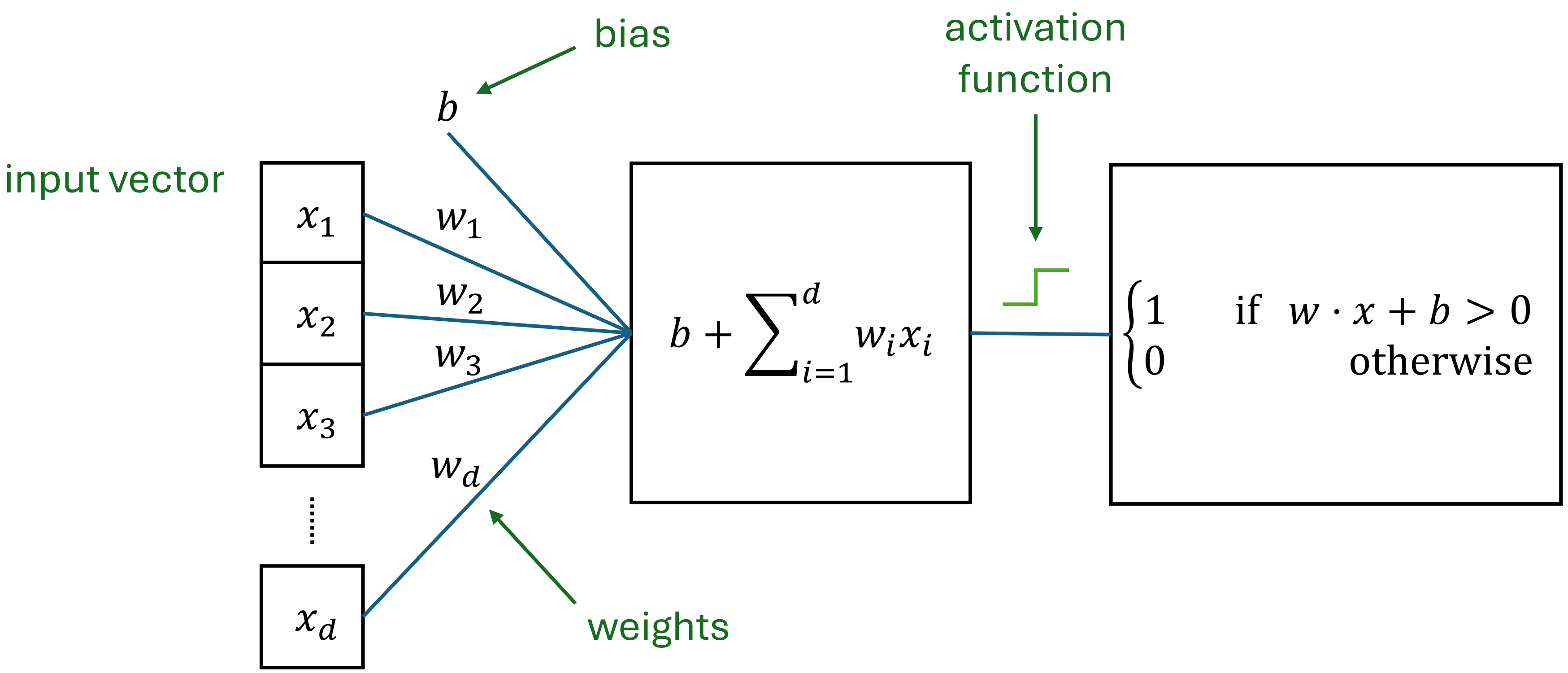
$$f\colon \mathbb{R}^d \to \{1,0\}$$

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

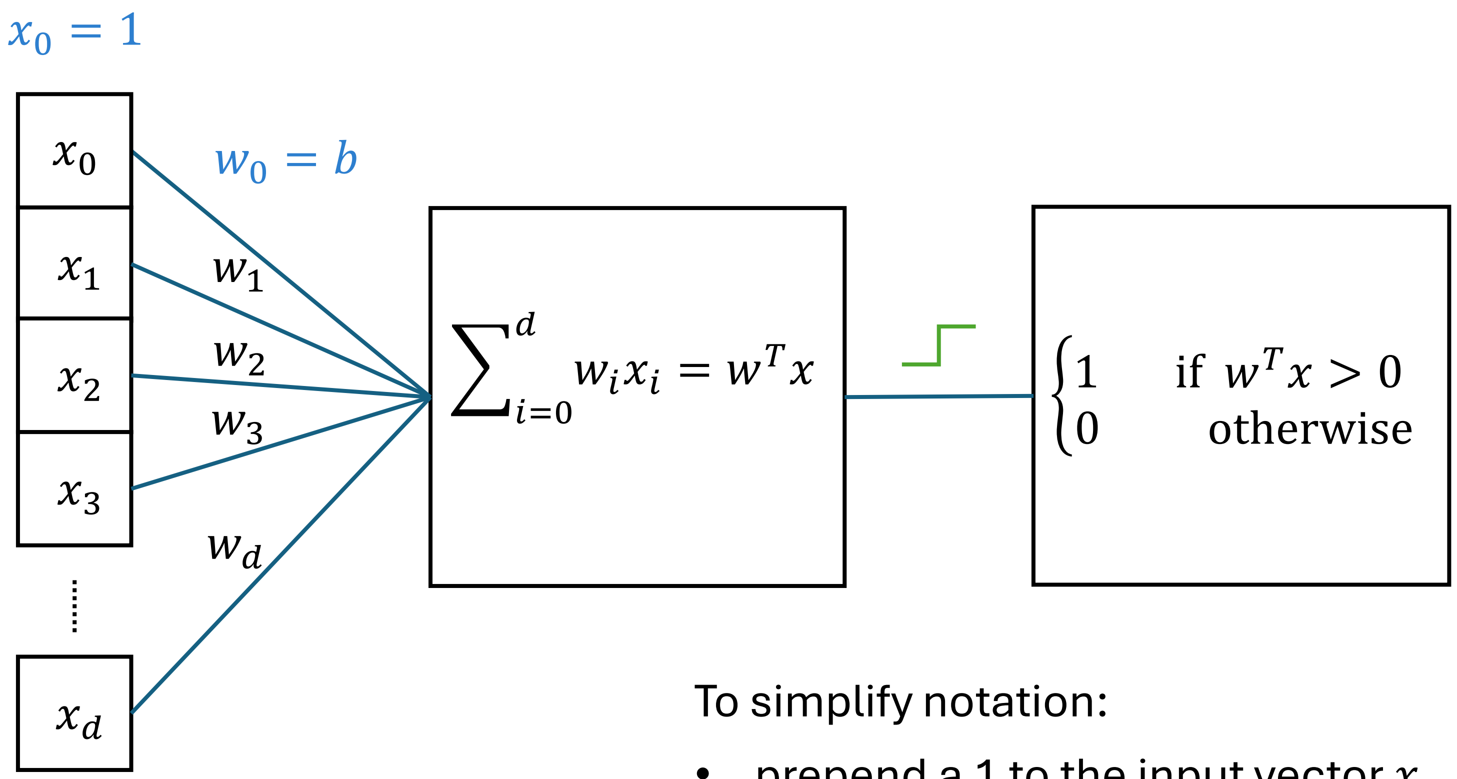
$$x, w \in \mathbb{R}^d \qquad b \in \mathbb{R}$$

- Linear Classifier
- Only works well on linearly separable problem

The Perceptron



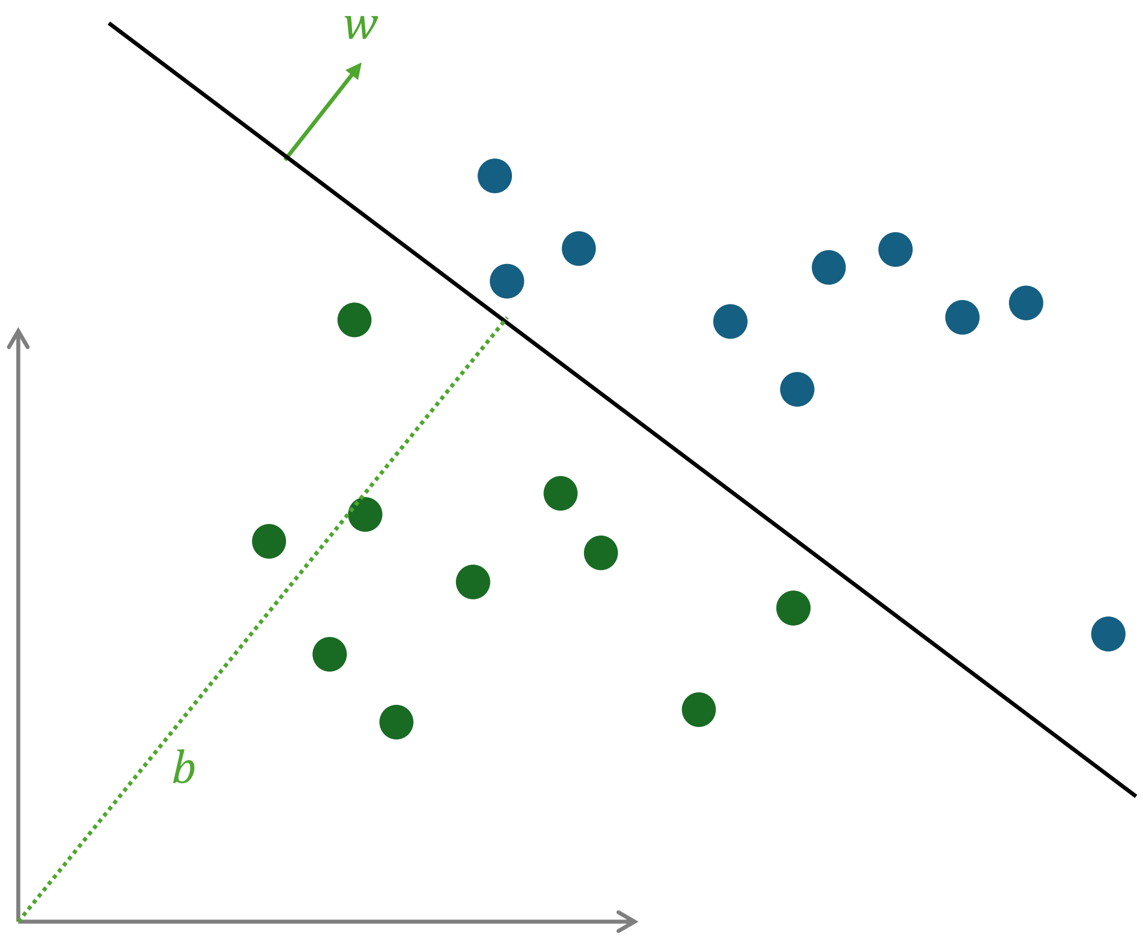
The Perceptron



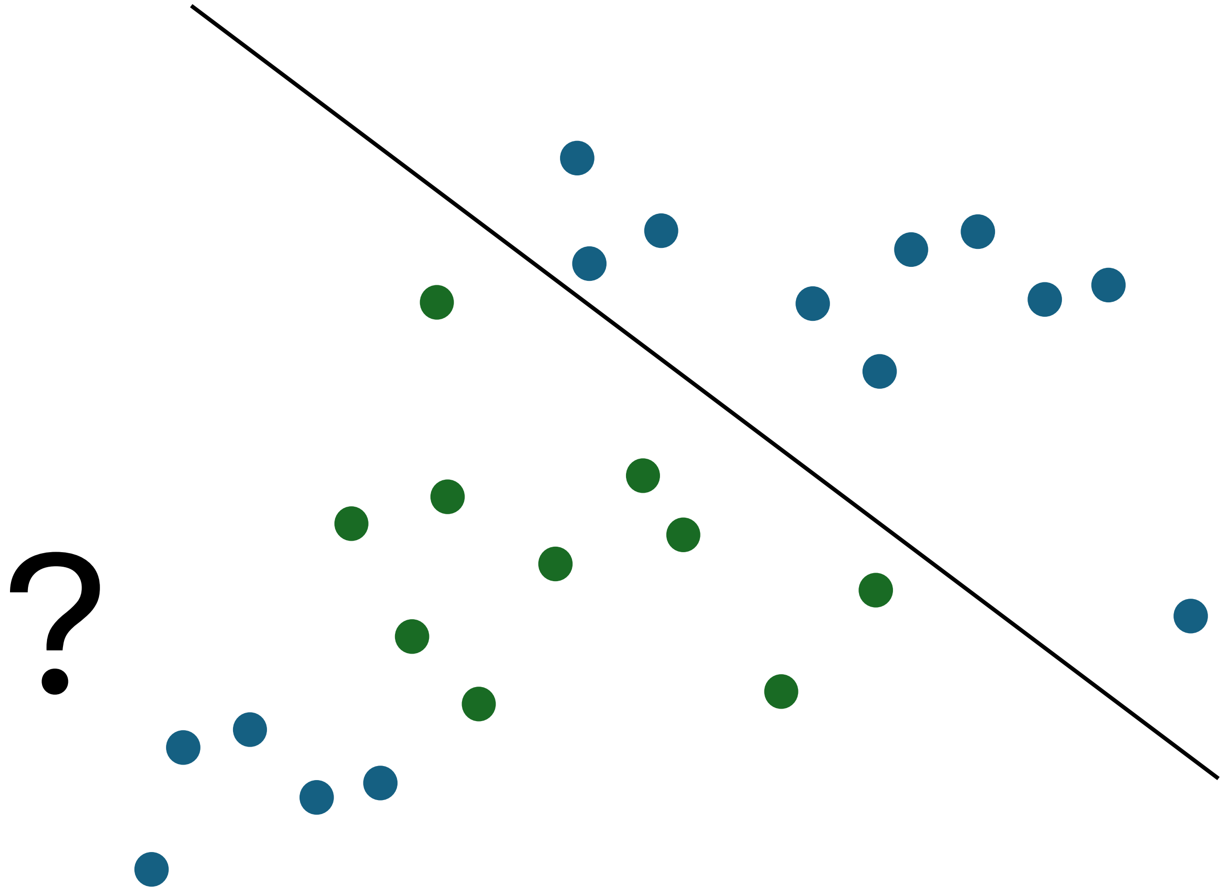
To simplify notation:

- prepend a 1 to the input vector x
- include the bias into the weights
- write everything as an inner product

The Perceptron

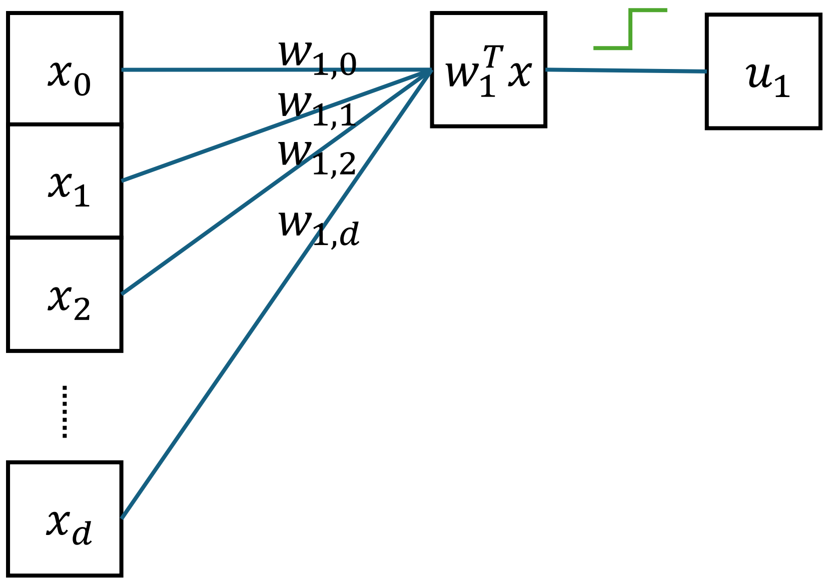


The Perceptron

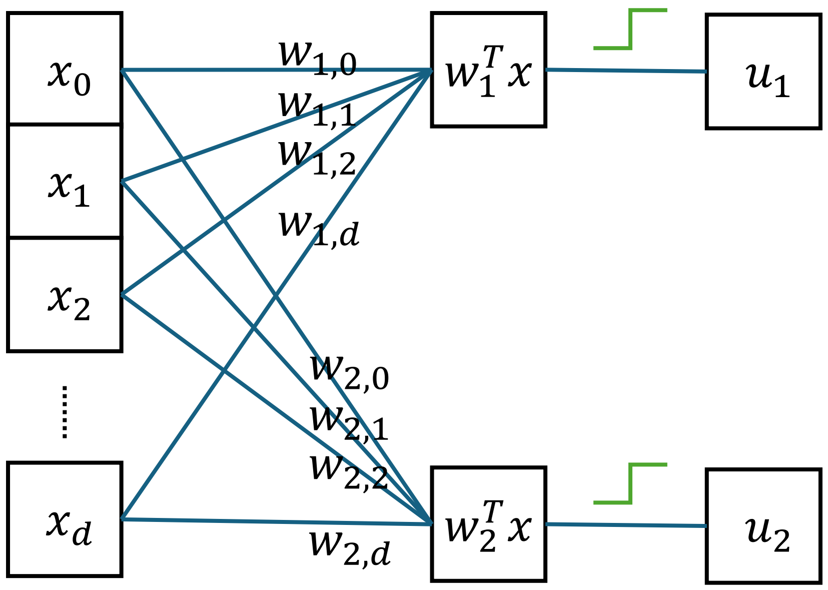


What to do when the problem is not linearly separable?

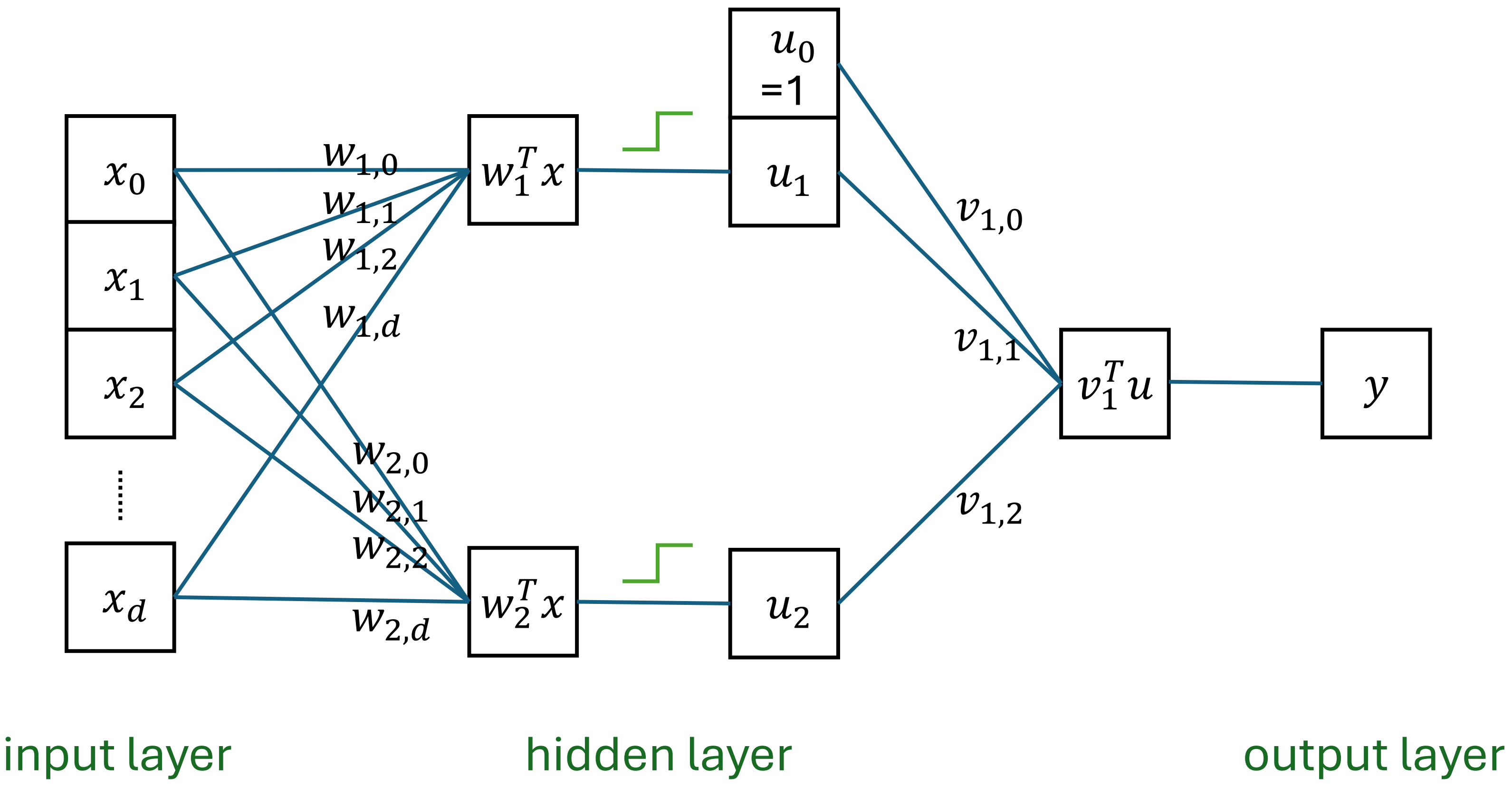
More Layers



More Layers

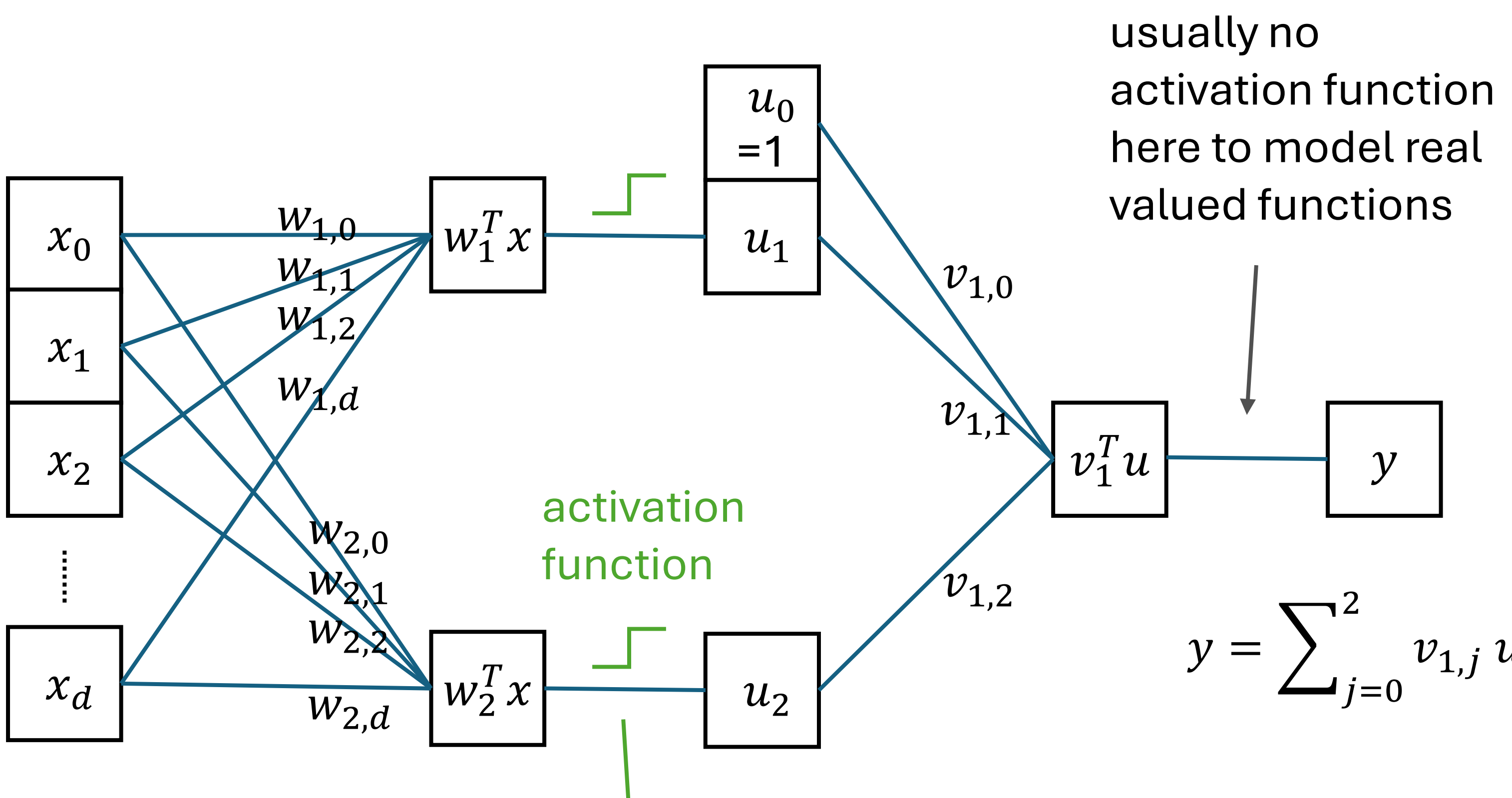


More Layers



x_0
x_1
x_2
x_d
$w_{1,0}$
$w_{1,1}$
$w_{1,2}$
$w_{1,d}$
$w_{2,0}$
$w_{2,1}$
$w_{2,2}$
$w_{2,d}$
$w_1^T x$
$w_2^T x$
u_0 =1
u_1
u_2
$v_{1,0}$
$v_{1,1}$
$v_{1,2}$
$v_1^T u$
y
input layer
hidden layer
output layer

More Layers



$$y = \sum_{j=0}^{2} v_{1,j}\, u_j$$

$$u_j = \sigma\left(\sum_{i=0}^{d} w_{j,i}\, x_i\right) = \sigma(w_j, x)$$

More Layers

- Matrix notation greatly simplifies writing down the computations in the layers, e.g.

$$u_j = \sigma(\sum_{i=0}^{d} w_{j,i}\, x_i) = \sigma(w_j\, x)$$

$$u = \sigma(wx)$$

$$y = uv = v\sigma(wx)$$

- Activation Function: Do we need one?

$$y = \sigma\big(v\sigma(wx)\big)$$

Assume $\sigma(z) = z$:

$$y = vwx = (vw)x$$

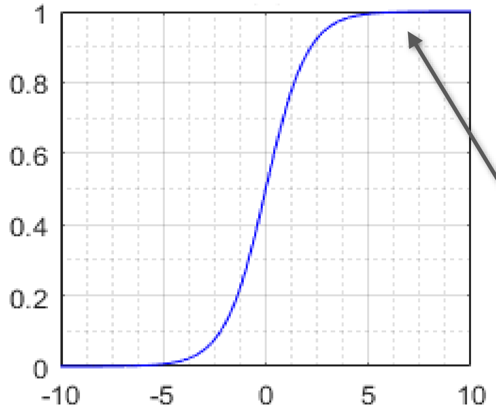
Yes, otherwise we still compute a linear function and nothing is gained by stacking the second layer!

Activation function

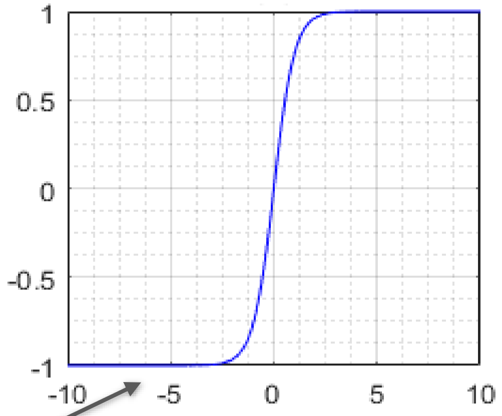
Layers are typically followed by **non-linear** activation functions, that act per neuron.

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$

Tanh: $\tanh(x) = 2\sigma(2x) - 1$



Output range: [0,1]



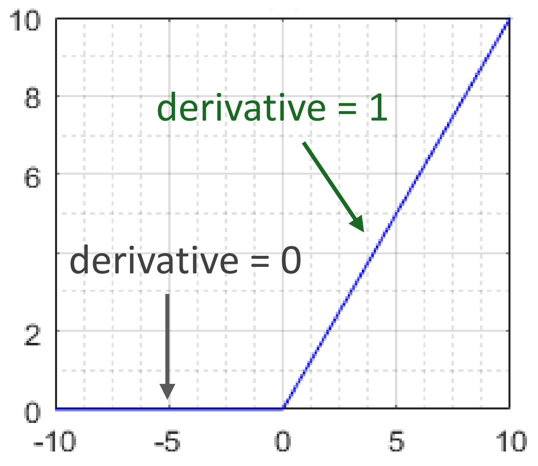
Output range: [-1,1]

Activation function

Layers are typically followed by **non-linear** activation functions, that act per neuron.

Rectified Linear Unit (ReLU)

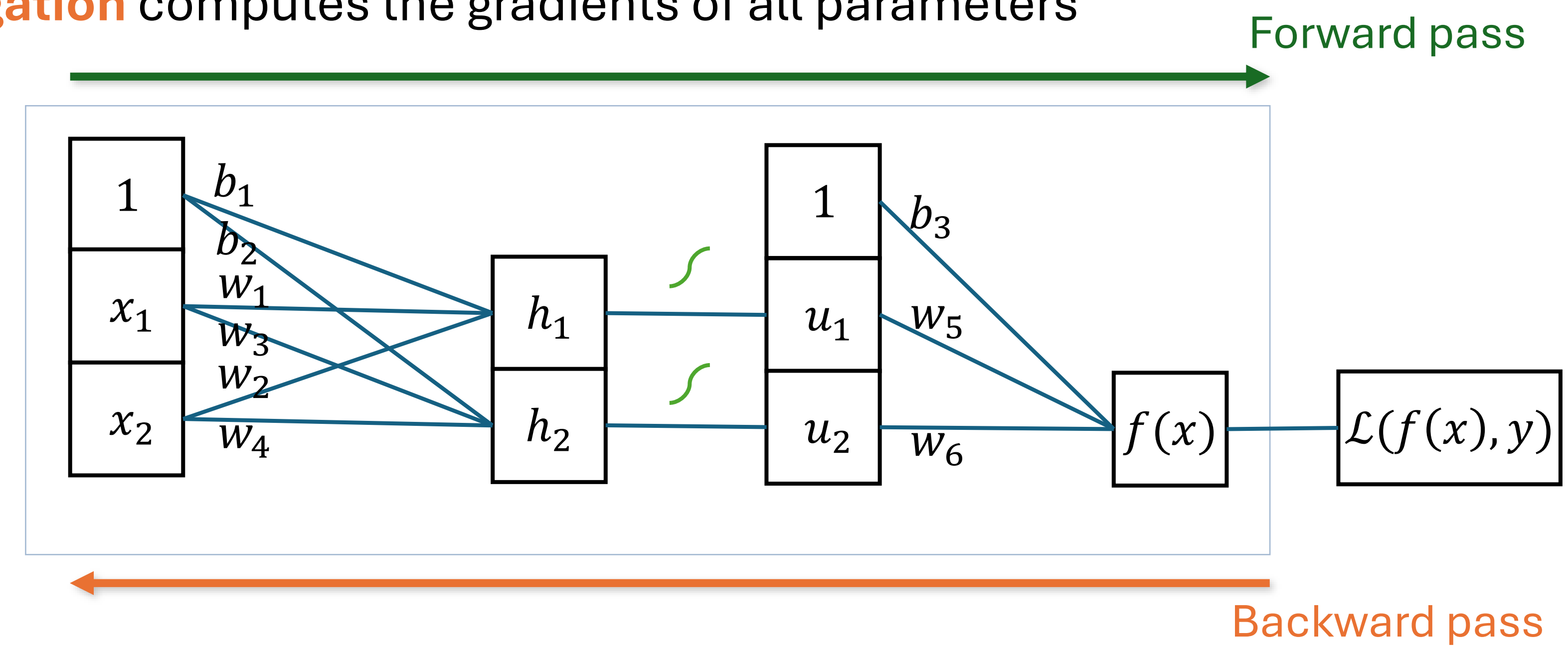
$$f(x) = \max(0, x)$$



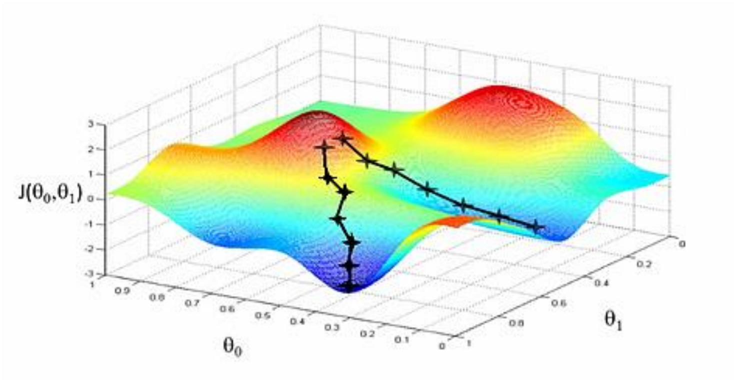
- Simply thresholds at zero
- Sparse activation
- Computationally efficient
- Non-saturating → speeds up convergence

How to train your network

- Set of N samples $\left(x^{(i)}, y^{(i)}\right)$
- Define loss $\mathcal{L}(f(x), y) \in \mathbb{R}$ to measure error *for a sample*
- *E.g. for regression task: mean square error (MSE) or mean absolute error (MAE) are common*
- **Training:** find weights that minimize $\sum_i \mathcal{L}\left(f\left(x^{(i)}\right), y^{(i)}\right)$ for the samples
- Often uses simple gradient descent methods
- **Backpropagation** computes the gradients of all parameters



Gradient Descent



How to use derivatives?

- Gradient descent to minimize error function

$$w^{(t+1)} = w^{(t)} - \lambda \frac{\partial \mathcal{L}}{\partial w}$$

  - Update the weights in every iteration of training with a small gradient step
  - Learning rate λ adjusts the step size

  - Error is defined over the **whole** training set: $\sum_i \mathcal{L}(x^{(i)})$
  - Need to compute and sum the derivatives of all samples before *one* gradient step
  - Slow but accurate updates

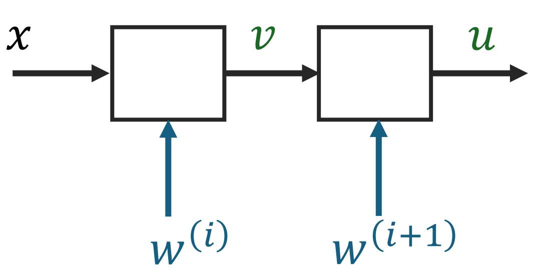
- Stochastic gradient descent (SGD)
  - Approximate derivative from small, random subset ([mini]batch) of training set
  - Noisy but faster
  - Usually: make sure to see every sample the same amount of times (epochs)

Backpropagation

- Backpropagation is an efficient way to compute derivatives of $\mathcal{L}$ w.r.t. *all* parameters

- Using (stored) activations from the forward pass

- Backpropagating information from layer $i+1$ to i and reusing already known (previously computed) derivatives



- Possible through chain rule $f\left(g\left(h(x)\right)\right)$

Layer 1

$$\frac{\partial}{\partial x} f\left(g\left(h(x)\right)\right) = \boxed{\frac{\partial f(u)}{\partial u}\left(g\left(h(x)\right)\right)} \frac{\partial g(v)}{\partial v}\left(h(x)\right) \frac{\partial h(x)}{\partial x}(x)$$

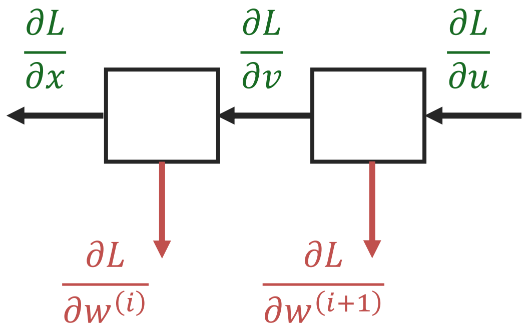
⋮

Layer 3

$$\frac{\partial}{\partial u} f\left(g\left(h(x)\right)\right) = \boxed{\frac{\partial f(u)}{\partial u}\left(g\left(h(x)\right)\right)}$$

simpler (Leibnitz's notation)

$$t = f(u), \qquad u = g(v), \qquad v = h(x) \quad \rightarrow \quad \frac{\partial t}{\partial x} = \frac{\partial t}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$



Break ☺

Deep Learning Frameworks

theano

MatConvNet

Chainer

PyTorch

MindSpore

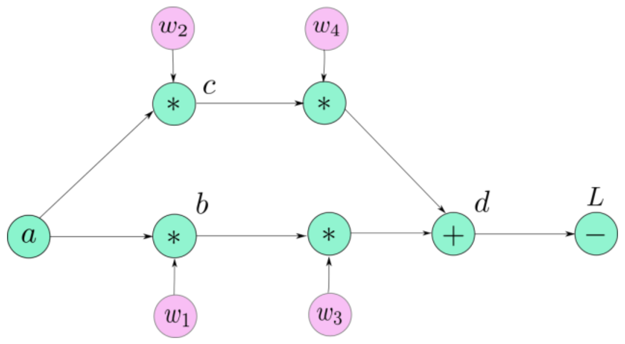
TensorFlow

Caffe2

PyTorch

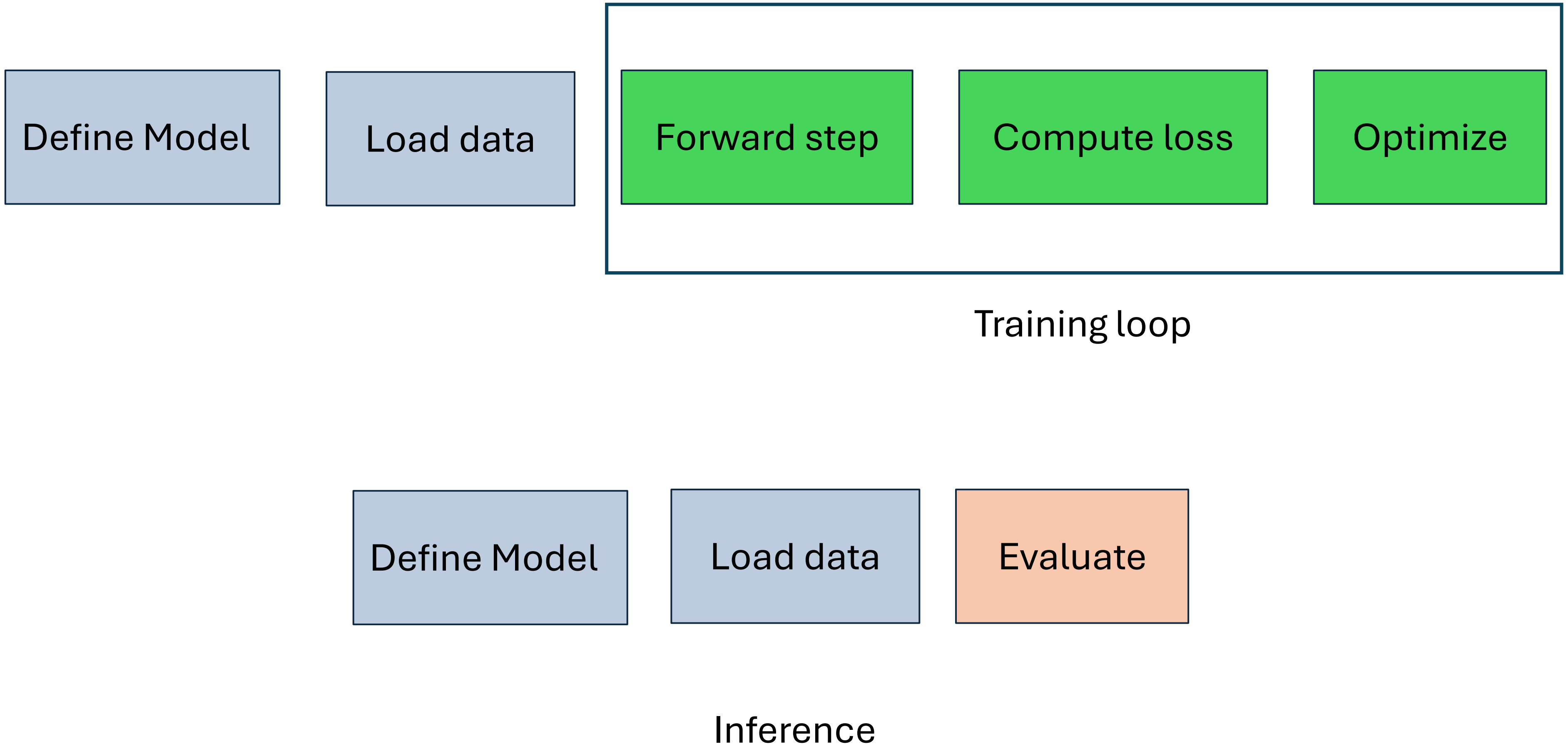
PyTorch Overview

- **Open source** machine learning library based on the Torch library
- Operates on multi-dimensional vectors (**Tensors**)
- Can execute on the CPU, GPU, distributed systems, etc.
- **Dynamic** computational graph (can change on runtime)
  - Nodes: Tensors
  - Edges: mathematical operations



- Performs **automatic differentiation**
- Python and C++ interface
- **Torchvision**: package that implements many important vision algorithms

PyTorch Overview a of Deep Learning Pipeline



PyTorch

`torch.nn.Module`
- Base class for a neural network module
- Can contain sub-modules
- Inherit this class when creating your neural network

`torch.nn.Parameter`
- Learnable tensor

`torch.nn.functional`
- A set of operations such as convolution, activation, etc.

PyTorch Define a Neural Network Model

- Extend the `Module` class of `torch.nn`
- Implement the constructor and the forward member function

```
import torch.nn as nn

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()

        ...


    def forward(self, x):
        ...
```

- Optional: implement own `backward()` for custom back-propagation

PyTorch Layers vs. Functions

Layers

- Defined as **classes** in `torch.nn`
- Has attributes, like weights and bias
- Internally calls the functional API
- Use whenever possible, i.e. for standard layers
- In general good coding style

Functions

- Defined as **functions** in `torch.nn.functional`
- **Only** provides **operation,** you need to pass your own weight and bias
- Learnable parameters need to be declared in `__init__()`, otherwise it will not learn
- Use in case you need to customize a layer

```
nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1,
groups=1, bias=True, padding_mode='zeros')
```

```
nn.functional.conv1d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)
```

PyTorch Data Loader

Use python utilities: `torch.utils.data.DataLoader` class

- Represents a Python iterable over a dataset (pass a dataset as argument)
- Automatic batching in standard cases, use own `collate_fn()` to customize

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

dataset = MyDataset()
dataloader = DataLoader(dataset, batch_size, ...)

class MyDataset(Dataset):
    def __init__(self, ...):
    ...

    def __getitem__(self, index):
        ...

    def __len__(self):
        ...
```

PyTorch Data Loader

Use python utilities: `torch.utils.data.DataLoader` class

- Represents a Python iterable over a dataset (pass a dataset as argument)
- Automatic batching in standard cases, use own `collate_fn()` to customize

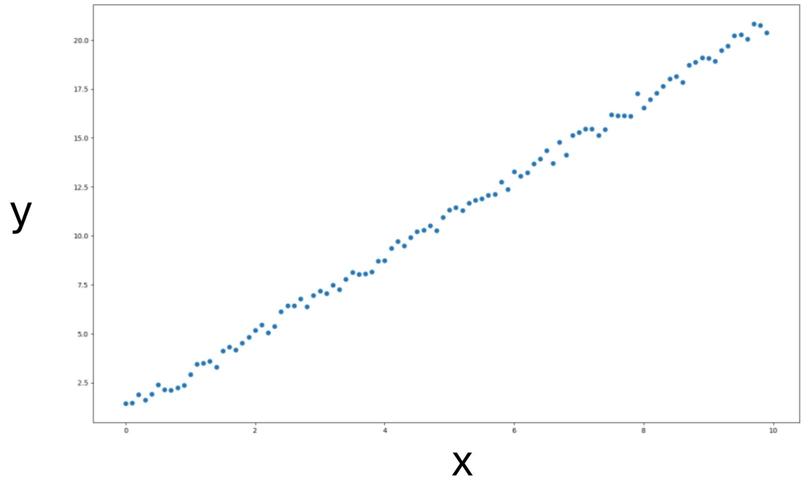
```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, data_dir):
        # get list of all image paths in the data_dir
        self.image_list = glob.glob(data_dir)

    def __getitem__(self, index):
        image, label = load_data(self.image_list[index])
        # normalization, augmentation, etc
        image, label = do_some_preprocessing(image, label)
        return image, label

    def __len__(self):
        return len(self.image_list)
```

PyTorch Linear Regression

Given pairs of x and y data, learn w and b

Optimize using SGD $y = wx + b$

Prepare toy data to train the model:




```python
# in the dataset class

def __init__(self):

    # create toy data for training
    x_values = [0.1*i for i in range(100)]
    x_train = np.array(x_values, dtype=np.float32).reshape(-1, 1)

    y_values = [2*i + 1 + random.random()-0.5 for i in x_values]
    y_train = np.array(y_values, dtype=np.float32).reshape(-1, 1)

    # from numpy to torch tensors

    self.x_train = torch.from_numpy(x_train)
    self.y_train = torch.from_numpy(y_train)


def __getitem__(self, index):

    return self.x_train[index], self.y_train[index]
```

Load data

PyTorch Linear Regression

We need the linear layer from **`torch.nn:`**

`torch.nn.Linear(in_features, out_features, bias=True)`

```
class LinearRegression(torch.nn.Module):
    def __init__(self, inputSize, outputSize):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(inputSize, outputSize)

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

Define Model

PyTorch Linear Regression

We need the linear layer from **`torch.nn`**:

`torch.nn.Linear(in_features, out_features, bias=True)`

Or create our own **w** and **b** parameters

```
class LinearRegression(torch.nn.Module):
    def __init__(self, inputSize, outputSize):
        super(LinearRegression, self).__init__()
        # self.linear = torch.nn.Linear(inputSize, outputSize)
        self.w = nn.Parameter(torch.ones([inputSize, outputSize]))

        self.b = nn.Parameter(torch.zeros([outputSize]))
    def forward(self, x):
        # out = self.linear(x)

        out = self.w * x + self.b
        return out
```

Define Model

PyTorch Linear Regression

```
#define data loader
dataset = MyDataset()
dataloader = DataLoader(dataset, batch_size=4)
# define model, loss function and optimizer
model = LinearRegression(inputSize=1, outputSize=1)
model.train()

loss_fn = torch.nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# train loop
for i in range(n_epochs):
    for input, target in dataloader:
        optimizer.zero_grad()
        # forward step
        output = model(input)
        # compute loss
        loss = loss_fn(output, target)
        # optimize: compute gradients and apply
        loss.backward()
        optimizer.step()
# evaluate by printing w and b

print("slope: ", model.w.data.numpy(), "\t offset: ",
model.b.data.numpy()) # slope: 1.9622   offset:  1.1127834
```

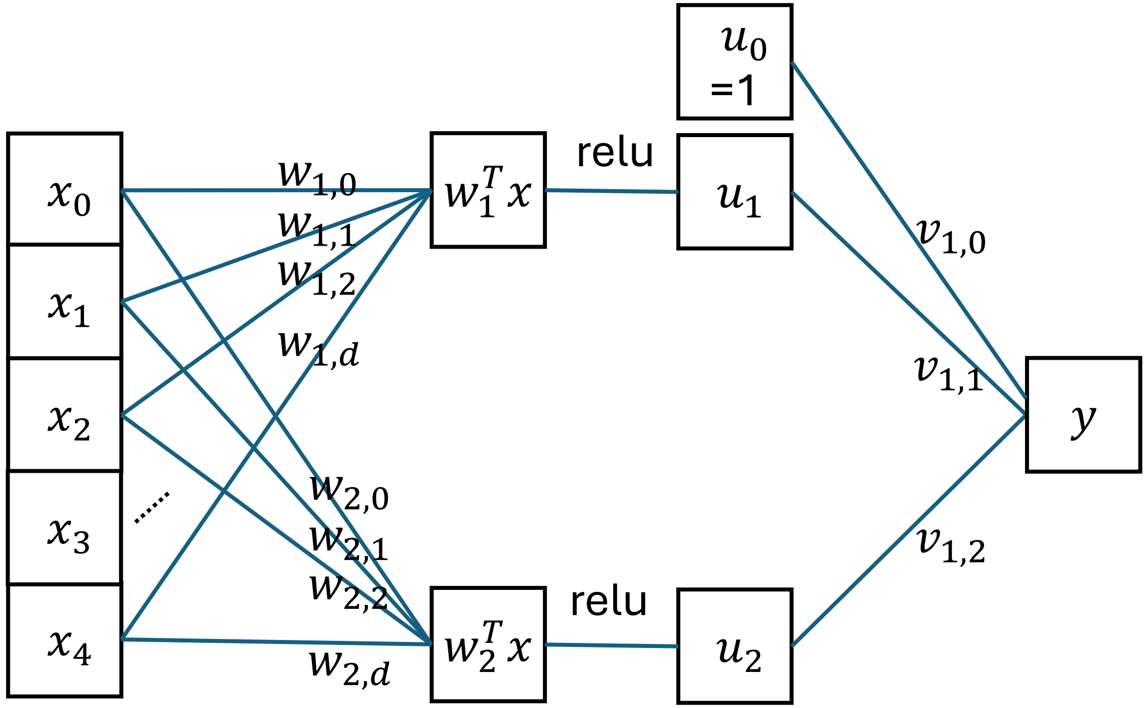
Forward step

Compute loss

Optimize

Evaluate

How does the PyTorch model look like for this network?




```
class MyNet(torch.nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        ...

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

Google Colab

- A Jupiter notebook stored on Google drive
- It runs online on Google resources (no need to have your own GPU)
- We will use it for practical coding exercises
- Contains coding cells and text cells

https://colab.research.google.com/drive/16pBJQePbqkz3QFV54L4NIkOn1kwpuRrj

Coding exercise

Given a set of inputs x and a set of outputs y, learn the function y = f(x) by a neural network

Develop and experiment with 3 different network models and see how they compare

Make a copy of this Google Colab: https://colab.research.google.com/drive/15wuKbpHuJmS8-FcsW2rRjuYmJh9aWIUb?usp=sharing

Follow instructions and complete TODO list

Once you are done, share the link with me per email (dl4cv.eci24@gmail.com), using the **Share** button on the top right corner of Google Colab.

Run the code cells, and preferably let the running outputs there for me to see.

However, you can expect that I will try to run your coding cells myself to see if the output can be reproduced.

References

Stanford CNN class notes:

https://cs231n.github.io/

General Machine Learning concepts explained simply:

https://www.youtube.com/@statquest

PyTorch tutorials:

https://pytorch.org/tutorials/beginner/introyt.html